ASTR 575

Fall 2015

1 Course Overview

2 Introduction to computing hardware and NMSU/Astronomy computing

3 Working in a Unix environment

4 Presentation/communication

5 Programming

6 Plotting

7 Algorithms

7.1 Writing a Program

Simple suggestions:

1. Make sure you fully understand the question/problem before starting to write the code. Outline the methodology you will use in words, diagrams, or figures, before getting caught up in the syntax of coding.
2. Generate the code in pieces. Consider writing out all the comments before the actual coding. Include some debugging statements, considering the possibility of building in a "verbosity" level from the start. Consider building in error trapping from the start.
3. Test code.

4. Clean up code.
5. Fully comment code.

7.2 Speed and scaling

7.3 Lists and list matching

7.4 Random number generation

(see NR chapter 7).

A random distribution of numbers is useful for simulating data sets to test analysis routines. Computer generated random numbers generally start with a seed number (usually the current clock time) if user doesn't specify one. Users should record whichever seed they use in case repeatability is desired.

Lowest level rnadom number generators give *uniform deviates*, i.e., equal probability of results in some range (usually 0 to 1 for floats). Python stuff:

```
>>> random.random
>>> random.seed
>>> numpy.random.random
>>> numby.random.seed
```

To generate random numbers for some other distribution, e.g. Gaussian, Poisson, luminosity funciton, mass function, etc. use the **Transformation Method**: Consider a cumulative distribution of the desired function (whose values range between zero and one). Generate a uniform random deviate between 0 and 1; these are your y-values. Solve your function for x in terms of y, and calculate all the values of x that correspond to your y-values (the random numbers). This does require that you can integrate your function, then invert the integral. (see NR, figure 7.3.1)

In-class exercise: generate random deviates for a "triangular" distribution: $p(x) \propto x$. What does the constant of proportionality need to be in order to make the integral equal to 1 (aka: a probability distribution)? Use the relation to generate random deviates, and plot them with a histogram.

$$y = 2x$$
$$F = \int y \, \mathrm{d}x = x^2$$
$$x(F) = \sqrt{(F)}$$

If you can't integrate and invert your function, use the **Rejection method**: choose a function that you *can* integrate and invert that is always higher than your desired distribution.

As before, choose a random deviate and find the corresponding values. Calculate the value of both the desired function and comparison function. Choose a uniform deviate between 0 and c(x). If it is larger than f(x), reject your value and start again. This requires two random deviates for each attempt, and the number of attempts before you get a deviate in your desired distribution depends on how close your comparison function is to your desired function (see NR figure 7.3.2).

Understand how to use and implement the transformation method for getting deviates from any function that is integrable and invertible.

See NR for several ways to generate deviates in desired functions.

Gaussian distribution: Used for, e.g. maxwellian speed distribution, and in general as a reasonable approximation for a large mean. `>>> numpy.random.normal`

$$P(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Poisson distribution: Used for counting statistics. `>>> numpy.random.poisson`

$$P(x, \mu) = \frac{\mu^x e^{-\mu}}{x!}$$

In class: simulate some data, e.g. a linear relation (x=1,10, y=x) with a Gaussian (mean=0, sigma=varius) scatter, and plot it. Alternate: CMD from isochrones with Poisson scatter in colors and magnitudes.

Know how to use canned functions for generating uniform deviates, Gaussian deviates, and Poisson deviates.

7.5 Interpolation

(see NR chapter 3)

Interpolation can be used if you have a tabulated or measured set of data, and want to estimate the values at intermediate locations in your data set, e.g. inverting a function, resampling data, etc.

- Linear interpolation: value $f(x)$ at some intermediate location, x, between two points, x_i and x_{i+1}:

$$f(x) = Ay_i + By_{i+1}$$
$$A = \frac{x_{i+1} - x}{x_{i+1} - x_i}$$
$$B = \frac{x - x_i}{x_{i+1} - x_i}$$

- Polynomial interpolation: Lagrange formula gives value polynomial of degree k going through k+1 points at arbitrary position, x, as a function of the tabulated data values:

$$L(x) = \sum_{j=0}^{k} y_j l_j(x)$$

$$l_j(x) = \prod_{0 \le m \le k, m \ne j} \frac{x - x_m}{x_j - x_m}$$

  Note that this provides the values of the interpolating polynomial at any location without providing the coefficients of the polynomial itself. The coefficients are determined such that the polynomial fit goes exactly through the tabulated data points.

Be aware that higher order is not necessarily better, e.g.:

$$f(x) = \frac{1}{(1 + 25x^2)}$$

There are also issues with even orders because this leads to using more tabulated points on one side of the desired value than the other (since even order means odd number of data points). At high order, the polynomial can go wildly off, especially near data edges.

Another problem with piecewise polynomial interpolation, even at low order, is that the input data values that are used to interpolate to a desired location change as you cross each data point. This leads to abrupt changes in derivatives (infinite second derivative) of the interpolated function. For some applications, this can cause problems, e.g., if you are trying to fit an observed data set to some tabulated series of models, if you use derivatives to get the best fit.

This issue can be overcome using *spline interpolation*, which gives interpolated values that go through the data points but also provides continuous second derivatives. Doing so, however, requires the use of non-local interpolation, i.e. the interpolating function includes values from all tabulated data points. Generally, people use cubic fits to the second derivatives, leading to *cubic spline interpolation*. To do this, specify boundary conditions at the ends of the data range. Usually, a *natural spline* is adopted, with zero second derivatives at the ends, but it is also possible to specify a pair of first derivatives at the ends.

Cubic splines are probably the most common form of interpolation, which doesn't necessarily mean the best in all circumstances.

Python implementation: `scipy.interpolate.interp1d` calculates interpolating coefficients for linear and spline interpolation, and can then be called to interpolate to desired position(s):

```
from scipy import interpolate
intfunc = interpolate.interp1d(xdata,ydata,kind='linear'|'slinear'|'quadratic'|'cubic')
intfunc(x) # returns interpolated value(s) at x
```

Exercise:

1. Define some function, $f(x)$
   ```
   def f(x):
   ...
   ```

2. For some given domain, plot your function at high sampling, e.g.:

   ```
   # get a high sampled array of independent variable
   x=numpy.linspace(xmin,xmax,1000)
   # plot the relation as a smooth line
   plt.plot(x,f(c))
   ```

3. For some given domain, sample your function with N points (evenly spaced, or maybe random), e.g.:

   ```
   numpy.linspace(xmin,xmax,npts) # provides npts xdata points in your your
   domain
   plt.plot(xdata,func(data),'o') # plot data points as points
   ```

4. Determine the interpolating polynomial for linear and cubic spline, and overplot the interpolated function.

   ```
   from scipy import interpolate
   # get the interpolating function (you need to choose which type)
   intfunc = interpolate.interp1d(xdata,func(xdata), kind='linear'|'slinear'|'quadrati
   # plot interpolated data at high sampling
   plt.plot(x,intfunc(x))
   ```

7.6 Fourier analysis basics and sinc interpolation

(see NR chapter 12)

Consider a function in two representations: values in "physical" space (e.g., time or location) vs. values in "frequency" space (e.g., temporal frequency or wavenumber). The two are related by Fourier transforms:

$$H(f) = \int h(x) e^{-2\pi i f x} \, \mathrm{d}x$$
$$h(x) = \int H(f) e^{-2\pi i f x} \, \mathrm{d}f$$

note that different implementations use different sign conventions, for example: *angular* frequency ($\omega = 2\pi f$).

The physical interpretation is that a function can be decomposed into the sum of a series of sine waves, with different amplitudes and phases at each wavelength/frequency. Because we have amplitude and phase, the Fourier transform is, in general, a complex function.

Fourier transforms can be determined for discrete (as oppposed to continuous) streams of data using the discrete Fourier transform, which replaces the integrals with sums. Algorithms

have been devoloped to compute the discrete Fourier transform quickly by means of the *Fast Fourier Transform (FFT)*; generally, this is done for data sets that are padded to a length of a power of 2. However, the FFT algorithm requires equally spaced data points. For unequally spaced points, the full discrete Fourier transform is required.

```
numpy.fft, scipy.fftpack
>>> import matplotlib.pyplot as plt
>>> import numpy as np
# generate sine wave
x = np.linspace(0, 10000., 8192)
y = np.sin(100*x)
plt.plot(x,y)
# fft f = np.fft.fft(y)
# plot amplitude, try to get frequencies right...
plt.plot(np.abs(f))
plt.plot(np.fft.fftfreq(8192), np.abs(f))
plt.plot(np.fft.fftfreq(8192,100./8192), np.abs(f))
```

Two particular operations involving pairs of functions are of general interest: convolution and cross-correlation.

- Convolution
$$g(x) * h(x) = \int g(x')h(x - x')\mathrm{d}x'$$
Usually, convolution is seen in the context of *smoothing*, where $h(x)$ is a normalized function (with a sum of unity), centered on zero; convolution is the process of running this function across an input function to produce a smoothed version. Note that the process of convolution is computationally expensive; at each point in a data series, you have to loop over all of the points (or at least those that contribute to the convolution in the Fourier domain, because of the *convolution theorem*, which states the covolving two functions in physical space is eequivalent to multiplying the transforms of the functions in Fourier space. Multiplication of two functions is faster than convolution). `numpy.convolve`
- Cross-correlation
$$g(x) \star h(x) = \int g(x')h(x + x')\mathrm{d}x'$$
Cross-correlation is the same as convolution if h is a symmetric function, but is usually used quite differently. It is generally considered a function of the *lag, x*. Multiply two functions, calculate the sum, then shift one of the functions and do it again. The sums are a function of the shift. For two similar functions, the cross-correlation will be a maximum when the two functions "line up", so this is useful for determining shifts between two functions (e.g., spatial shifts of images, or spectral shifts from velocity).

Back to interpolation... See Bracewell chapter 10 for sinc interpolation and the sampling theorem.

When we sample a function at finite locations, we are multiplying by a sampling function, which is just a discrete set of impulses spaced by Δx. In the Fourier domain, this is convolving by the Fourier transform of the sampling function, which is a discrete set of frequencies spaced by $1/\Delta x$.

An important and useful result for interpolation is called the *sampling theorem* which arises in the case of a *band-limited* function, i.e. a function that has no power above some critical frequency known as the *Nyquist frequency*. In this case, if the function is sampled at sufficient frequency, (twice the Nyquist frequency), then it is possible to recover the entire function. Figure 10.3 from Bracewell shows the idea graphically.

What to do:

1. Fourier transform the sampled function.
2. Multiply by a box function.
3. Fourier transform back.

Alternatively, convolve the sampled function with the Fourier transform of a box function, which is the sinc function:

$$\mathrm{sinc}(x) = \frac{\sin(x)}{x}$$

This leads to *sinc interpolation*, which is the ideal interpolation in the case of a band-limited function.

Interpolation and Uncertainties

Generally all interpolation schemes can be considered as a sum of data values multiplied by a set of interpolation coefficients at each point:

$$y(x) = \sum_{i=0}^{n} y_i P_i(x)$$

So far, we have been considering interpolation in perfect data. If there are uncertainties on the data points, there will be uncertainties in the interpolated values. These can be derived by error propagation, if the errors on the individual data points are uncorrelated:

$$\sigma(y(x))^2 = \sum \sigma(y(x_i))^2 P_i(x)^2$$

However, the interpolated errors are now correlated, so using them properly requires you to use the *full covariance matrix*.

$$= \sigma_u^2 \left(\frac{\partial x}{\partial u}\right)^2 + \sigma_v^2 \left(\frac{\partial x}{\partial v}\right)^2 + 2\sigma_{uv}^2 \frac{\partial x}{\partial u}\frac{\partial x}{\partial v}$$

$$\sigma_{uv}^2 = \lim_{n\to\infty} \frac{1}{N} \sum (u_i - < u >)(v_i - < v >)$$

In the presence of noise (uncertainties), it is often advantageous to *model* the data by fitting an underlying function to a tabulated set of data points with some merit function criteria

for determing what function provides the best fit. The fit can then be used to predict values at arbitrary location.

Beware of mixing and matching interpolation and fitting: doing fits on data sets with uncorrelated errors is substantially easier than doing them on data sets with correlated errors.

7.7 Differentiation and integration

Integration: simple rules, extended rules, reaching a desired accuracy.

Monte Carlo integration

7.8 Differential equations

Runge-Kutta, boundary value problems.

8 Fitting

8.1 Overview: frequentism vs Bayesian

Given a set of observations/data, one often wants to summarize and get at underlying physics by fitting some sort of model to the data. The model might be an empirical model or it might be motivated by some underlying theory. In many cases, the model is parametric: there is some number of parameters that specify a particular model out of a given class.

The general scheme for doing this is to define some merit function that is used to determine the quality of a particular model fit, and choose the parameters that provide the best match, e.g. a minimum deviation between model and data, or a maximum probability that a given set of parameters matches the data.

When doing this, one also wants to understand something about how reliable the derived parameters are, and also about how good the model fit actually is, i.e. to what extent it is consistent with your understanding of uncertainties on the data.

There are two different "schools" about model fitting: Bayesian and frequentism.

- **Frequentism** (also called the classical approach)
  Consider how *frequently* a data set might be observed given some underlying model. $P(D|M)$ – *probability of observing a data set given a model.* The model that produces the observed data most frequently is viewed as the correct underlying model, and as such, gives the best parameters, along with some estimate of their uncertainty.

- **Bayesian**
  $P(M|D)$ – *probability that a model is correct given a data set.* It allows for the possibility that external information may prefer one model over another, and this is incorporated into the analysis as a *prior*. It considers the probability of different models, and hence, the probability distribution functions of parameters. Examples of priors: fitting a Hess diagram with a combination of SSPs, with external constraints on allowed ages; fitting UTR data for low count rates in the presence of readout noise.

The frequentist paradigm has been mostly used in astronomy up until fairly recently, but the Bayesian paradigm has become increasingly widespread. In many cases they can give the same result, but with somewhat different interpretation. In some cases, results can differ. The basic underpinning of Bayesian analysis comes from **Bayes theorem** of probabilites:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

where $P(A|B)$ is the conditional probability, i.e. the probability of A given that B has occurred. Imagine there is some joint probability distribution function of two variables, $p(x, y)$ (see ML 3.2; this could be a distribution of stars as a function of effective temperature and luminosity). Think about slices in each direction to get to the probability at a given point, and we have:

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x)$$

which gives:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

which is Bayes theorem. Bayes theorem also relates the conditional probability distribution to the *marginal* probability distribution:

$$p(x) = \int p(x|y)p(y)\ \mathrm{d}y$$

$$p(y) = \int p(y|x)p(x)\ \mathrm{d}y$$

$$p(x|y) = \frac{p(y|x)p(x)}{\int p(y|x)p(x)\ \mathrm{d}x}$$

This is all fairly standard statistics, but the Bayesian paradigm extends this to the idea that the quantity A or B can also represent a hypothesis, or model, i.e. the relative probability of different models.

In the context of Bayesian analysis, we can talk about the probability of models, and we have:

$$P(M|D) = \frac{P(D|M)P(M)}{P(D)}$$

In this case, P(D) is a normalization constant, P(M) is the prior on the model (which, in the absense of any additional information, is equivalent for all models: a *noninformative prior*. A noninformative prior can itself be a prior; a uniform prior is not necessarily invarient to a change in variables, e.g. uniform in the logarithm of a variable is not uniform in the variable). In the noninformative case, the Bayesian result is the same as the frequentist maximum likelihood. However, in the Bayesian analysis we'd want to calculate the full probability distribution function for the model parameter, μ.

In practice, frequentist analysis yields parameters and their uncertainties, while Bayesian analysis yields probability distribution functions of parameters. The latter is often more comutationally intensive to calculate. Bayesian analysis includes explicit priors.

For some more detailed discussion, see http://arxiv.org/abs/1411.5018

Understand the basic conceptual differences between a frequentist and a Bayesian analysis. Know Bayes' theorem. Understand the practical differences.

Starting with a frequentist analysis (which often provides a component of the Bayesian analysis): Given a set of data and some model (with parameters), consider the probability that the data will be observed if the model is correct, and then choose the set of parameters that maximizes this probability.

For example, when fitting a straight line through the data, the best fit is determined by the method of *least squares*. Consider that observed data points have some measurement uncertainty, σ. For now, all points have homoschedastic (equal) uncertainties that are distributed according to a Gaussian.

1. What is the probability is of observing a given data value from a known Gaussian distribution?
2. What is the probability of observing a series of two independently drawn data values?

Suppose we are fitting N data points (x_i, y_i), where $i = 0, \ldots, N-1$. that has M adjustable parameters a_j, where $j = 0, \ldots, M-1$. The model predicts a functional relationship between the measured independent and dependent variables:

$$y(x) = y(x|a_0 \ldots a_{M-1})$$

where the notation indicates dependence on the parameters explicitly on the right-hand side, following the vertical bar.

So given some model $y(x_i|a_j)$, the probability of observing a series of data points is:

$$P(D|M) \propto \prod_{i=0}^{N-1} \left\{ \exp\left[-\frac{1}{2} \left[\frac{y_i - y(x_i|a_j)}{\sigma} \right]^2 \right] \Delta y \right\}$$

where a_j is the jth parameter of the model, and Δy is some small range in y.

Maximizing the probability is the same thing as maximizing the *logarithm* of the probability, which is the same thing as *minimizing* the *negative* of the logarithm:

$$-\log P(D|M) = \sum_{i=0}^{N-1} \left\{ \frac{1}{2} \left[\frac{y_i - y(x_i)}{\sigma} \right]^2 \right\} - N \log(\Delta y) + const$$

If σ is the same for all points, then this is equivalent to minimizing

$$\sum_{i=1}^{N-1} [y_i - y(x_i|a_j)]^2$$

which is the least squares fit.

Understand how least squares minimization can be derived from a maximum likelihood consideration in the case of normally-distributed uncertainties.

Consider a simple application, multiple measurements of some quantity, so the model is $y(x_i) = \mu$ where we want to determine the most probable value of the quantity, i.e. μ.

Minimizing gives:

$$\frac{\mathrm{d}(-\log P(D|M))}{\mathrm{d}\mu} = 2 \sum_{i=0}^{N-1} (y_i - \mu) = 0$$

$$\mu = \sum \frac{y_i}{N}$$

which is a familiar result. To calculate the uncertainty on the *mean*, use error propagation, for $x = f(u, v, \ldots)$,

$$\sigma(\mu)^2 = \sigma_{y_0}^2 \left(\frac{\partial \mu}{\partial y_0}\right)^2 + \sigma_{y_1}^2 \left(\frac{\partial \mu}{\partial y_1}\right)^2 + \ldots$$

$$\sigma_\mu^2 = \sum \frac{\sigma^2}{N^2} = \frac{\sigma^2}{N}$$

$$\sigma_\mu = \frac{\sigma}{\sqrt{N}}$$

For *heteroschedastic* (unequal) uncertainties on the data points:

$$-\log P(D|M) = \sum_{i=0}^{N-1} \left[\frac{(y_i - y(x_i))^2}{\sigma_i^2}\right] + const \equiv \chi^2$$

Here, we are minimizing a quantity called χ^2. An important thing about χ^2 is that the probability of a given value of χ^2 given N data points and M parameters can be calculated *analytically*. This is called the χ^2 *distribution for* $\nu = N - M$ *degrees of freedom*; see, e.g. `scipy.stats.chi2`, e.g. `cumulative density function (cdf)`, which you want to be in some range, e.g. 0.05 0.95.

A way to qualitatively judge the quality of a fit is by the *reduced* χ^2, aka. χ^2 per degree of freedom:

$$\chi_\nu^2 = \frac{\chi^2}{N - M} = \frac{\chi^2}{\nu}$$

which is expected to have a value near unity. However, the *probability* of χ^2 is what should be calculated since the spread in χ_ν^2 depends on ν (the standard deviation is $\sqrt{2\nu}$).

It is important to recognize that this analysis depends on the assumption that the uncertainties are distributed according to a normal (Gaussian) distribution.

χ^2 can be used as a method for checking uncertainties, or even determining them (at the expense of being able to say whether your model is a good representation of the data). See ML 4.1.

Know specifically what χ^2 is and how to use it. Know what reduced χ^2 is, and understand degrees of freedom.

For our simple model of measuring a *single quantity*, we have:

$$\frac{\mathrm{d}(-\log P(D|M))}{\mathrm{d}\mu} = 2 \sum \frac{(y_i - \mu)}{\sigma_i^2} = 0$$

$$\sum \frac{y_i}{\sigma_i^2} - \mu \sum \frac{1}{\sigma_i^2} = 0$$

$$\mu = \frac{\sum \frac{y_i}{\sigma_i^2}}{\sum \frac{1}{\sigma_i^2}}$$

i.e., a weighted mean. Again, you can use error propagation to get (work not shown):

$$\sigma_\mu = \left(\sum \frac{1}{\sigma_i^2} \right)^{-1/2}$$

A more complicated example: fitting a straight line to a set of data. Here, the model is

$$y(x_i) = a + bx_i$$

and χ^2 is:

$$\begin{aligned} \chi^2 &= \sum_{i=0}^{N} \left[\frac{[y_i - (a + bx_i)]^2}{\sigma_i^2} \right] \\ &= \sum_{i=0}^{N} \left[\frac{[y_i - a - bx_i]^2}{\sigma_i^2} \right] \end{aligned}$$

Again, we want to minimize χ^2, so we take the derivatives with respect to each of the parameters and set them to zero:

$$\frac{\partial \chi^2}{\partial a} = 0 = -2 \sum \frac{(y_i - a - bx_i)}{\sigma_i^2}$$

$$\frac{\partial \chi^2}{\partial b} = 0 = -2 \sum \frac{x_i (y_i - a - bx_i)}{\sigma_i^2}$$

Separating out the sums, we have:

$$\sum \frac{y_i}{\sigma_i^2} - a \sum \frac{1}{\sigma_i^2} - b \sum \frac{x_i}{\sigma_i^2} = 0$$

$$\sum \frac{x_i y_i}{\sigma_i^2} - a \sum \frac{x_i}{\sigma_i^2} - b \sum \frac{x_i^2}{\sigma_i^2} = 0$$

or

$$aS + bS_x = S_y$$

$$aS_x + bS_{xx} = S_{xy}$$

where the various S are a shorthand for the sums:

$$S = \sum \frac{1}{\sigma_i^2}$$

$$S_x = \sum \frac{x_i}{\sigma_i^2}$$

$$S_{xy} = \sum \frac{x_i y_i}{\sigma_i^2}$$

$$S_{xx} = \sum \frac{x_i y_i}{\sigma_i^2}$$

$$S_y = \sum \frac{y_i}{\sigma_i^2}$$

which is a set of two equations with two unknowns. Solving, you get:

$$a = \frac{S_{xx}S_y - S_x S_{xy}}{SS_{xx} - S_x^2}$$

$$b = \frac{SS_{xy} - S_x S_y}{SS_{xx} - S_x^2}$$

Know how to derive the least squares solution for a fit to a straight line.

We also want the uncertainties in the parameters. Again, use propagation of errors to get (work not shown):

$$\sigma_a^2 = \frac{S_{xx}}{SS_{xx} - S_x^2}$$

$$\sigma_b^2 = \frac{S}{SS_{xx} - S_x^2}$$

Finally, we will want to calculate the probability of getting χ^2 for our fit, in an effort to understand

1. if our uncertainties are not properly calculated and normally distributed
2. if our model is a poor model

Let's do it. Simulate a data set, fit a line, and output parameters, parameter uncertainties, χ^2, and probability of χ^2.

8.2 General linear fits

We can generalize the least squares idea to any model that is some linear combination of terms that are a function of the independent variable, e.g.

$$y = a_0 + a_1 f_1(x) + a_2 f_2(x) + a_3 f_3(x) + \ldots$$

such a model is called a *linear* model because it is linear in the parameters (but not necessarily linear in the independent variable). The model could be a polynomial of arbitrary order, but could also include trigonometric functions, etc. We write the model in simple form:

$$y(x) = \sum_{k=0}^{M-1} a_k X_k(x)$$

where there are M parameters, N data points, and $N > M$. The χ^2 merit function can be written as:

$$\chi^2 = \sum_{i=0}^{N-1} \left[\frac{(y_i - \sum_{k=0}^{M-1} a_k X_k(x_i))^2}{\sigma_i^2} \right]$$

Minimizing χ^2 leads to the set of M equations:

$$0 = \sum_{i=1}^{N-1} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=0}^{M-1} a_j X_j(x_i) \right] X_k(x_i)$$

where $k = 0, \ldots, M-1$.

Separating the terms and interchanging the order of the sums gives:

$$\sum_{i=0}^{N-1} \frac{y_i X_k(x_i)}{\sigma_i^2} = \sum_{j=0}^{M-1} \sum_{i=0}^{N-1} \frac{a_j X_j(x_i) X_k(x_i)}{\sigma_i^2}$$

Define:

$$\alpha_{jk} = \sum_{i=0}^{N-1} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2}$$

$$\beta_j = \sum_{i=0}^{N-1} \frac{X_j(x_i) y_i}{\sigma_i^2}$$

then we have the set of equations:

$$\alpha_{jk} \alpha_j = \beta_k$$

for $k = 0, \ldots, M-1$.

Know the equations for a general linear least squares problem, and how they are derived.

Sometimes these equations are cast in terms of the *design matrix*, A, which consists of N measurements of M terms:

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i}$$

with N rows and M columns. Along with the definition:

$$b_i = \frac{y_i}{\sigma_i}$$

we have:

$$\alpha = A^T \cdot A$$

$$\beta = A^T \cdot b$$

where the dots are for the matrix operation that produces the sums. This is just another notation for the same thing, introduced here in case you run across this language or formalism.

For a given data set, α and β can be calculated either by doing the sums or by setting up the disign matrix and using matrix arithmetic. Then solve the set of equations for α_k.

Note that this formulation applies to problems with multiple independent variables, e.g., fitting a surface to a set of points; simply treat x as a vector of data, and the formulation is exactly the same.

8.3 Solving linear equations

This is just a linear algebra problem, and there are well-developed techniques (see NR chapter 2). For the most simple case, invert the matrix α to get

$$\alpha_k = \alpha_{jk}^{-1}\beta_k$$

A simple algorithm for inverting a matrix is called a *Gauss-Jordan elimination*. In particular, NR recommends the use of singular value decomposition for solving all but the simplest least squares problems; this is especially important if your problem is nearly singular, i.e. where two or more of the equations may not be totally independent of each other: fully singular problems should be recognized and redefined, but it is possible to have non-singular problems encounter singularity under some conditions depending on how the data are sampled. See NR 15.4.2 and chapter 2 (in these notes?).

As an aside, note that it is possible to solve a linear set of equations significantly faster than inverting the matrix through various types of matrix decomposition, e.g., LU decomposition and Cholesky decomposition (see NR Chapter 2). There may be linear algebra problems that you encounter that only require the solution of the equations and not the inverse matrix. However, for the least squares problem, we often do want the inverse matrix $C = \alpha^{-1}$, because its elements have meaning. Propagation of errors gives:

$$\sigma^2(a_k) = C_{jj}$$

i.e., the diagonal elements of the inverse matrix. The off-diagonal elements give the covariances between the parameters, and the inverse matrix, C, is called the *covariance matrix*.

A Python implementation of matrix inversion and linear algebra in general can be found in scipy.linalg. Here is the summed matrix implementation for a straight line (but the functional form only comes in through the derivative function.):

```
def deriv(x) :
    # if x is numpy array, then we can return vectors of derivatives
    try :
        return [np.ones(len(x)),x]
    except :
        return [1.,x]

# data points in (x,y)
# loop over parameters, sums over data points (y) are done with vector arithmetic
for k in np.arange(npar) :
    beta[k] = np.sum(deriv(x)[k]*y/sigma**2)
    for j in np.arange(npar) :
        alpha[k,j] = np.sum(deriv(x)[k]*deriv(x)[j]/sigma**2)
c=np.linalg.inv(alpha)
print np.dot(c,beta)
```

Here is the design matrix approach to the sums:

```
A = np.vander(x, 2)  # Take a look at the documentation to see what this function do
ATA = np.dot(A.T, A / yerr[:, None]**2)
w = np.linalg.solve(ATA, np.dot(A.T, y / yerr**2))
V = np.linalg.inv(ATA)
```

Linear least squares is also implemented in `astropy` in the modeling module using the LinearLSQFitter class (which uses `numpy.linalg`); `astropy` has a number of standard models to use, or you can provide your own custom model.

```
from astropy.modeling import models, fitting
fit_p = fitting.LinearLSQLSQFitter()
p_init = models.Polynomial1D(degree=degree)
pfit = fit_p(p_init, x, data)
```

Astropy has a number of common models, but you can also define your own model using models.custom_model, by supplying a function that returns values, and a function that returns derivatives with respect to the parameters.

Be able to computationally implement a linear least squares solution to a problem.

Note that for problems with large numbers of parameters, the linear algebra can become very computationally expensive. In many cases, however, the matrix that represents these problems may only be sparsely populated, i.e. so-called sparse matrices (see Numerical Recipes 2.7, Figure 2.7.1).

Example: spectral extraction

In such cases, there are methods that allow the equations to be solved significantly more efficiently.

However, not all fits are linear in the parameters. Which of the following are not?

- $y = a_0 x + a_1 e^{-x}$
- $y = a_0(1 + a_1 x)$
- $y = a_0 \sin(x - a_1)$
- $y = a_0 \sin^2 x$
- $y = a_0 e^{(-(x-a_1)^2/a_2^2)}$

8.4 Nonlinear fits

In a linear fit, the χ^2 surface is parabolic in parameter space, with a single minimum, and linear least squares can be used to determine the location of the minimum. In the nonlinear case, however, the χ^2 surface can be considerably more complex, and there is a (good) possibility that there are multiple minima, so one needs to be concerned about finding the global minimum and not just a local minimum. Because of the complexity of the surface, the best fit is found by an iterative approach.

A conceptually simple approach would be a *grid search*, where one simply tries all combinations of parameters and finds the one with the lowest χ^2. Obviously, this is extremely computationally inefficient, especially for problems with more than a few parameters. One is also forced to decide on a step size in the grid, although you might imagine a successively refined grid as you proceed. But in general, this method is not recommended, apart from occaisionally trying it to try to ensure that your more efficient solution is not landing in a local minimum.

Better approaches attempt to use the χ^2 values to find the minimum more efficiently. They can generally be split into two classes: those that use the derivative of the function and those that don't. If you can calculate derivatives of χ^2 with respect to your parameters, then this provides information about how far you can move in parameter space towards the minimum. If you can't calculate derivates, you can evaluate χ^2 at several different locations, and use these values to try to work your way towards the minimum.

With derivatives, the approach has a fairly close parallel to the linear least squares problem. Around the final minimum, the χ^2 surface can be approximated as a parabola, and it is possible to correct the solution to the minimum solution if one can arrive at a set of parameters near to the final minimum. This is acheived via the set of equations:

$$\sum_{i=0}^{M-1} \alpha_{kl} \delta a_1 = \beta_k$$

where

$$\beta_k = -\frac{1}{2}\frac{\partial \chi^2}{\partial a_k} = \sum_{i=0}^{N-1} \frac{(y_i - y(x_i|a))}{\sigma_i^2} \frac{\partial (y(x_i|a))}{\partial a_k}$$

$$\alpha_{kl} = \frac{1}{2}\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} = \sum_{i=0}^{N-1} \left[\frac{1}{\sigma_i^2} \frac{\partial y(x_i|a)}{\partial a_k} \frac{\partial y(x_i|a)}{\partial a_l} - (y_i - y(x_i|a)) \frac{\partial^2 y(x_i|a)}{\partial a_k \partial a_l} \right]$$

The matrix α_{kl} is known as the *curvature matrix*. In most cases, it is advisable to drop the second derivative term in this matrix (see NR 15.5.1 for a partial explanation). The standard approach uses:

$$\alpha_{kl} = \sum_{i=0}^{N-1} \left[\frac{1}{\sigma_i^2} \frac{\partial y(x_i|a)}{\partial a_k} \frac{\partial y(x_i|a)}{\partial a_l} \right]$$

To implement this, you choose a starting guess of parameters, solve for the corrections δa_l to be applied to a current set of parameters, and iterates until one arrives at a solution that does not change significantly.

Far from the solution, this method can be significantly off in providing a good correction, and, in fact, can even move parameters away from the correct solution. In this case, it may be advisable to simply head in the the steepest downhill direction in χ^2 space, which is known as the method of steepest descent. Note that while this sounds like a reasonable

thing to do in all cases, it can be very inefficient in finding the final solution (see, e.g. NR Figure 10.8.1).

A common algorithm switches between the method of steepest descent and the parabolic approximation and is known as the Levenberg-Marquardt method. This is done by using a modified curvature matrix:

$$\sum_{i=0}^{M-1} \alpha'_{kl} \delta a_1 = \beta_k$$

where

$$\alpha'_{jj} = \alpha_{jj}(1 + \lambda)$$
$$\alpha'_{jk} = \alpha_{jk} \text{ (for } j \neq k)$$

When λ is large, this gives a *steepest descent* method. When λ is small, this gives a *parabolic* method. This leads to the following recipe:

1. Choose a starting guess of parameter vector (a) and calculate $\chi^2(a)$.
2. Calculate the correction using model λ, e.g. $\lambda = 0.001$ and evaluate χ^2 at the new point.
3. If $\chi^2(a + \delta a) > \chi^2(a)$, increase λ and try again.
4. If $\chi^2(a + \delta a) < \chi^2(a)$, decrease λ, update a, and start the next step.
5. Stop when the convergence criteria are met.
6. Invert the curvature matrix to get parameter uncertainties.

Two key issues in nonlinear least squares is finding a good starting guess and a good convergence criterion. You may want to consider multiple starting guesses to verify that you're not converging to a local minimum. For convergence, you can look at the amplitude of the change in parameters or the amplitude in the change of χ^2.

This method is implemented in `astropy` in the modeling module using the LevMarLSQFitter which is used identically to the linear least squares fitter described above.

There are certainly other approaches to non-linear least squares, but this provides an introduction.

8.4.1 A nonlinear fitter without derivatives

A common minimization method that does not use derivatives is the "downhill simple" or *Nelder-Mead* algorithm. For this technique, χ^2 is initially evaluated at $M + 1$ points (where M is the number of parameters). This makes an M-dimensional figure, called a *simplex*. The largest value is found, and a trial evalaution is made at a value reflected through the volume of the simplex. If this is smaller than the next-highest value of the original simplex, a value twice the distance is chosen and tested; if not, then a value half the distance is tested, until a better value is found. This point replaces the initial point, and the simplex has moved a step; if a better value can't be found, the entire simplex is contracted around the best point,

and the process tries again. The process in then repeated until a convergence criterion is met. See NR Figure 10.5.1 and section 10.5. The simplex works its way down the χ^2 surface, expanding when it can take larger steps, and contracting when it needs to take smaller steps. Because of this behavior, it is also known as the "amoeba" algorithm.

Python/scipy implementation of Nelder-Mead

Python/scipy summary of optimization routines in general

Understand the concepts of how a nonlinear least squares fit is performed, and the issues with global vs local minima, starting guesses, and convergence criteria. Know how to computationally imiplement a nonlinear least squares solution.

8.5 Parameter uncertainties and confidence limits

Fitting provides us with a set of best-fit parameters, but, because of uncertainties and limited number of data points, these will not necessarily be the true parameters. One generally wants to understand how different the derived parameters might be from the true ones.

The covariance matrix gives information about the uncertainties on the derived parameters. In the case of well-understood uncertainties that are strictly distributed according to a Gaussian, these can be used to provide *confidence levels* on your parameters; see NR 15.6.5. However, if the uncertainties are not well-understood, numerical techniques can be used to derive parameter uncertainties. A straightforward technique if you have a good understanding of your *measurement* uncertainties is the *Monte Carlo simulation*. In this case, you simulate your data set multiple times, derive parameters for each simulated data set, and look at the range of fit parameters as compared with the input parameters. To be completely representative of your uncertainties, you would need to draw the simulated data set from the true distribution, but you don't know what that is (it's what you're trying to derive). So we take the best-fit from the actual data as representative of the true distribution, and hope that the *difference* between the derived parameters from the simulated sets and the input parameters is representative of the difference between the actual data set and the true parameters.

If you don't have a solid understanding of your *data* uncertainties, then *Monte Carlo* will not give an accurate representation of your parameter uncertainties. In this case, you can use multiple samples of your own data to get some estimate of the parameter uncertainties. A common technique is the *bootstrap* technique, where, if you have N data points, you make multiple simulations using the same data, drawing N data points from the original set **with replacement** (i.e. the same data point can be drawn more than once), derive parameters from multiple simulations, and look at the distribution of these parameters.

However, you may need to be careful about the interpretation of the confidence intervals determined by any of these techniques, which are based on a frequentist interpretation of the data. For these calculations, note that confidence intervals will change for different data sets. The frequentist interpretation is that the true value of the parameter will fall within the confidence levels at the frequency specified by your confidence interval. If you happen

to have taken an unusual (*in*frequent) data set, the true parameters may not fall within the confidence levels derived from this data set.

Understand how uncertainties can be estimated from least-squares fitting via the covariance matrix, Monte Carlo simulation, and the bootstrap method, and how these techniques work.

8.6 Bayesian analysis

Review material above, also see astroML, NR and http://arxiv.org/abs/1411.5018

As discussed above and in additional references, Bayesian analysis has several significant differences from the frequentist/likelihood approach. First, Bayesian analysis calculates the probability of model parameters given data, rather than the probability of data given a model.

$$P(M|D) = \frac{P(D|M)P(M)}{P(D)}$$

Practically, this allows for the possibility of specifying an explicit prior on the model parameters. The Bayesian analysis may give the same result as the frequentist analysis for some choices of the prior, but it makes the prior explicit, rather than hidden.

Second, the Bayesian analysis produces a joint probability distribution function of the model parameters. The frequentist analysis produces a most probable set of parameters and a set of confidence intervals.

The Bayesian analysis is computationally more challenging because you actually have to compute the probability for different model parameters. This is essentially performing a grid search of the full parameter space, which can be computationally intractable for large numbers of parameters. Fortunately, computational techniques have been developed to search through the full parameter space concentrating only on the regions with non-negligible probability; as discussed below, Markov Chain Monte Carlo is foremost among such techniques.

8.6.1 The prior

Priors in Bayesian analysis reflect external knowledge about parameters that exist independent of the data being analyzed. If there is no external knowledge, then you want to use a *noninformative* prior. Often, this is just a statement that all parameter values are equally likely. In many cases, this makes a Bayesian analysis equivalent to a frequentist analysis. However, beware that model parameters could be expressed with a transformation of variables, and a flat distribution in one variable may not be a flat distribution under a variable transformation, and could lead to different results. Also, a prior that appears to be unbiased may not be so, e.g., consider the fitting parameters for a straight line, in particular the slope.

8.6.2 Marginalization

For many problems, only some of the parameters are of interest, while others fall into the category of so-called *nuisance* parameters, which may be important for specifying an accurate model (which is fundamentally important), even though they may not be of interest for the scientific question you are trying to answer. In such cases, Bayesian analysis uses the concept of *marginalization*, where one integrates over the dimensions of the nuisance parameters to provide probability distribution functions of only the parameter(s) of interest.

Marginalization is also used to give probability distribution functions of individual parameters, i.e. without regard to the values of other parameters. However, one needs to be aware that parameter values may be correlated with each other, and marginalization hides such correlations.

Marginalization can also be used to derive the probability distribution of some desired quantity given measurements of other quantities if there is some relation between the measured and desired quantity. An example is in determining ages of stars given a set of observations, e.g. of spectroscopic parameters and/or apparent magnitudes. Here, stellar evolution gives predictions for observed quantities as a function of age, mass, and metallicity; in conjunction with an initial mass function, you get predictions for number of stars for each set of parameters. Given some priors on age, mass, and/or metallicity, you could compute the probability distribution of a given quantity by marginalizing over all other parameters. Observational constraints would modify this probability distribution.

Understand what marginalization means, and why and when it can be used.

8.6.3 Markov Chain Monte Carlo (MCMC)

Calculating marginal probability distributions is basically a big integration problem. If the problem has many parameters, the multi-dimensional integral can be very intensive to calculate. One technique for multi-dimensional integration is ***Monte Carlo*** integration. Choose a (large) number of points at random within some specified volume (limits in multiple dimensions), sample the value of your function at these points and estimate the integral as

$$\int f \mathrm{d}V = V < f >$$

where

$$< f >= \frac{1}{N} \sum f(x_i)$$

We could use this to calculate marginal probability distribution functions, but it is likely to be very inefficient if the probability is small over most of the volume being integrated over. Also, for most Bayesian problems, we do not have a proper probability density function because of an unknown normalizing constant; all we have is the relative probability at different points in parameter space.

To overcome these problems, we would instead like to place points in a volume proportional

to the probability distribution at that point; we can then calculate the integral by summing up the number of points. This is achieved by a *Markov Chain Monte Carlo* analysis. Here, unlike Monte Carlo, the points we choose are not statistically independent, but are chosen such that they sample the (unnormalized) probability distribution function in proportion to its value. This is acheived by setting up a Markov Chain, a process where the value of a sampled point depends only on the value of the previous point. To get the Markov Chain to sample the (unnormalized) PDF, the transistion probability has to satisfy:

$$\pi(x_1)p(x_2|x_1) = \pi(x_2)p(x_1|x_2)$$

where p is a transition probability to go from one point to another.

To do the integral, sum the number of points in the chain at the different parameter values.

There are multiple schemes to generate such a transition probability. A common one is called the *Metropolis-Hastings* algorithm: from a given point, generate a candidate step from some proposal distribution, $q(x2|x1)$, that is broad enough to move around in the probability distribution $q(x2|x1)$ in steps that are not too large or too small. If the candidate step falls at a larger value of the probability function, accept it, and start again. If it falls at a smaller value, accept it with probability proportional to

$$\pi(x_c)q(x_1|x_{2c})\pi(x_1)q(x_{2c}|x_1)$$

For a symmetric Gaussian proposal definition, note that the q values cancel out.

Note also that you need to choose a starting point. If this is far from the maximum of the probability distribution, it may take some time to get to the equilibrium distribution. This leads to the requirement of a *burn-in* period for MCMC algorithms.

In practice, you have to be careful about choosing an appropriate proposal distribution, and an appropriate burn-in period.

Implementation: see emcee and pymc, among others. Also this.

Note that `emcee` uses an algorithm that is different from Metropolis-Hastings, that is designed to function more efficiently in a case where the PDF may be very narrow, leading to very low acceptance probabilities with M-H. Partly this is accomplished by using a number of *walkers*, i.e. multiple different Markov chains, but these are not totally independent: knowledge from other walkers are used (somehow) to help explore the parameter space more efficiently. Note that since the walkers are not independent, the chain from a single walker does not sample the PDF proportionally; the combined chain from all of the walkers is what needs to be used.