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
```

7.6 Fourier analysis basics and sinc interpolation

(see NR chapter 12)

In Fourier analysis, 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, and sometimes angular frequency ($\omega = 2\pi f$) is used.

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 sueful for determining shifts between two functions (e.g., spatial shifts of images, or spectral shifts from velocity).

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 of a model 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:

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

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

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

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.

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 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*. All points have homeoschedastic (equal) uncertainties, σ, that are distributed according to a Gaussian. We want to know what the probability is of observing a given data value from a known Gaussian distribution, and the probability of observing a series of two independently drawn data values.

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

$$P(D|M) \propto \prod_{i=0}^{N-1} \exp\left[\frac{-0.5(y_i - y(x_i|a_j))^2}{\sigma^2}\right]\Delta y$$

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 mazimizing the logarithm of the probability, which is the same thing as maximizing the negative of the logarithm, i.e. minimize:

$$-\log P(D|M) = \sum_{i=0}^{N-1} \left[\frac{0.5(y_i - y(x_i))^2}{\sigma^2} \right] - N\log(\Delta y) + const$$

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

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

i.e., least squares.

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 $x = f(u, v, \ldots)$,

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

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

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.

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

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

which is expected to have a value near unity. The *probability* of χ^2 should be calculated since the spread in χ^2_ν 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: (insert equations) i.e., a weighted mean. Again, you can use error propagation to get (work not shown): (more equaitons).

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:

$$\chi^2 = \dots$$

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.

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

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:

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

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

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

Define:

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). Simply 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,

8.4 Nonlinear fits

8.4.1 A nonlinear fitter without derivatives

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.

8.6 Bayesian analysis

8.6.1 The prior

8.6.2 Marginalization

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

where

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.