Numerical Methods with Python

1 Introduction

You will be given light curve data for several RR Lyrae variables. This data will be processed to find the periods and flux averaged magnitudes of the stars.

2 Objectives

1. Plot the raw light curves.
2. Find the periods in the light curves.
3. Phase the light curves.
4. Fit a Fourier series to the light curves.
5. Fit the light curves to a template.
6. Integrate the light curves to find the flux averaged magnitude.

3 Plotting the Data

The data will be provided in a text file with each line containing (in this order) the time (in mean Julian date), brightness (apparent magnitude), and error in brightness of the star. Plot the brightness as a function of time. By looking at this plot, what can you tell about the behavior of the brightness as a function of time, if you can tell anything at all?

4 Finding the Periods in the Data

Fourier analysis (described in chapter 5 of the appendix) can be used to extract the period(s) from a set of irregularly and/or sparsely sampled periodic data. Instead of writing the code in python to do this yourself, you can use a mainstream astronomy tool, Period04.

Download the free tool Period04:

`http://www.univie.ac.at/tops/Period04/`

Be sure to also download the Period04 manual and the tutorial data files. Go through both tutorials in the manual.

After you have gone through these tutorials, you should be able to extract the dominant periods from the data. Record and report these periods for each data set; you will need them later. Also show and describe a periodogram.

For reference, typical RR Lyrae periods range from a few hours to 1.5 days.

In the event Period04 fails to work properly, then try Supersmoother. The instructor should provide you this code.

5 Phasing the Data

Phasing the data is also referred to as period folding.

Data cannot be taken continuously (data can only be taken at night, other people have to use the telescope, etc.) nor can it necessarily be taken at the same rate (some nights may have more data than others). The period of the observed star may be less than the period of data taking. Therefore, each night of data taking may sample a different portion of the star's cycle. The objective of period folding is to take observations spanning many cycles and condense them to one cycle.

Since the periods are known, the data can be folded to any period P. Suppose the first datum occurs at time t_0. Then, to phase the data at time $t \geq t_0$, the time difference is divided by the period: $(t - t_0)/P$. However, since we want $0 \leq P \leq 1$ and the data is periodic, the integral portion of the ratio is subtracted to find where the datum at time t lies (given by the phase Φ) within one period:

$$\Phi = \frac{t - t_0}{P} - \text{int}\left(\frac{t - t_0}{P}\right).$$

A video explanation of period folding can be found at the University of North Carolina's website:

`http://skynet.unc.edu/ASTR101L/videos/folding/`

Fold the light curves into their primary period(s) and plot the phased data.

6 Fitting a Fourier Series to the Data

We want to fit the light curves to a Fourier series such that we have a continuous expression for the light curve. Fit the data to a nine-term Fourier series using a χ^2 minimization process. This is rigorously explained in chapter 3, particularly section 3.6, of the appendix. The form of the Fourier series follows:

$$m(t) = \langle m \rangle + \sum_{k=1}^{9} A_k \sin\left(2\pi k f t + \phi_k\right)$$

where $m(t)$ is the observed brightness, $\langle m \rangle$ is the mean brightness, A_k is the amplitude of the k^{th} harmonic of the Fourier series, $f = 1/P$ is the frequency where P is the fitted period of the magnitude variation, and ϕ_k is the phase of the k^{th} harmonic at $t = 0$. The parameters to be fit here are A_k and ϕ_k.

Fit the light curves to Fourier series and report the results.

Hint: If the data and its error are given by v_i and σ_i, respectively:

$$\chi^2 = \sum_i \left(\frac{m - v_i}{\sigma_i}\right)^2$$

7 Fitting a Template to the Data

Templates are sets of data points corresponding to the expected shape of a light curve. Often, stars can be categorized by a finite and small number of templates, and the best fit template of a light curve can reveal the type of star for which the light curve is taken.

You will be given a number of templates. Fit these templates to the period-folded data using a χ^2 minimization procedure and see which template fits best. Interpolation (which can be performed by python) will be required, as the templates are not guaranteed (and, in general, do not) have points at the same times as you have data. Furthermore, the amplitudes and means of the templates will have to be adjusted in order to fit the data.

8 Integrating the Light Curve

In astronomy, the brightness of a star is generally given by apparent magnitude m, and luminosity is generally given by absolute magnitude M such

that for a star a distance d away, the flux F is given by

$$F = \frac{M}{4\pi d^2}.$$

Most RR Lyrae stars have $M = 0.6$. Knowing this, the distance d can be found in parsecs:

$$m - M = -5 + 5 \log_{10} d.$$

The flux averaged magnitude FAM is given by the integral of the flux in phase space:

$$\text{FAM} = -2.5 \int_0^1 F(\Phi) d\Phi$$

The flux averaged magnitude is used in various physical analyses of stellar processes.

Integrate the phased light curves and report the values.

9 Appendix

Table 3.1: Observations of the periods, P (days), the absolute magnitude M, and the colour $B - V$, of Cepheids in the LMC.

$\log P$	M	B-V	$\log P$	M	B-V	$\log P$	M	B-V
0.408	-2.39	0.58	0.978	-4.81	0.84	1.367	-5.18	0.90
0.429	-2.53	0.51	0.993	-4.10	0.69	1.388	-5.50	0.71
0.492	-3.38	0.53	1.038	-4.28	0.75	1.425	-5.01	1.04
0.529	-2.72	0.60	1.051	-4.05	0.76	1.453	-5.15	0.90
0.569	-2.95	0.58	1.119	-4.23	0.85	1.484	-5.80	0.77
0.677	-3.32	0.64	1.134	-4.58	0.75	1.503	-5.33	0.93
0.703	-3.45	0.61	1.160	-4.40	0.85	1.536	-5.23	0.81
0.841	-3.81	0.66	1.209	-5.05	0.72	1.563	-6.11	0.83
0.911	-3.90	0.73	1.272	-4.79	0.82	1.655	-5.34	1.20
0.936	-4.11	0.72	1.303	-4.76	0.86	1.684	-6.13	0.96
0.960	-4.24	0.57	1.353	-4.65	0.98	1.897	-6.45	1.24

$$\begin{aligned} \epsilon_i &= a_0 + a_1 x_i - y_i \\ \epsilon_i^2 &= (a_0 + a_1 x_i - y_i)^2 \\ S = \sum_{i=1}^{n} \epsilon_i^2 &= \sum_{i=1}^{n} (a_0 + a_1 x_i - y_i)^2 \end{aligned}$$

where n is the number of observations. Our problem is to minimize S with respect to a_0 and a_1. To minimize with respect to a_0 we find $\partial S/\partial a_0$ and set it to zero, and we do the same for a_1:

$$\begin{aligned} \frac{\partial S}{\partial a_0} &= 2 \sum_{i=1}^{n} (a_0 + a_1 x_i - y_i) = 0 \\ \frac{\partial S}{\partial a_1} &= 2 \sum_{i=1}^{n} (a_0 + a_1 x_i - y_i) x_i = 0 \end{aligned}$$

Therefore

$$\begin{aligned} \sum_{i=1}^{n} (a_0 + a_1 x_i - y_i) &= 0 \\ \sum_{i=1}^{n} (a_0 x_i + a_1 x_i^2 - x_i y_i) &= 0 \end{aligned}$$

We can re-write these two equations as follows

$$\begin{aligned} n a_0 + a_1 \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i &= 0 \\ a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i y_i &= 0 \end{aligned}$$

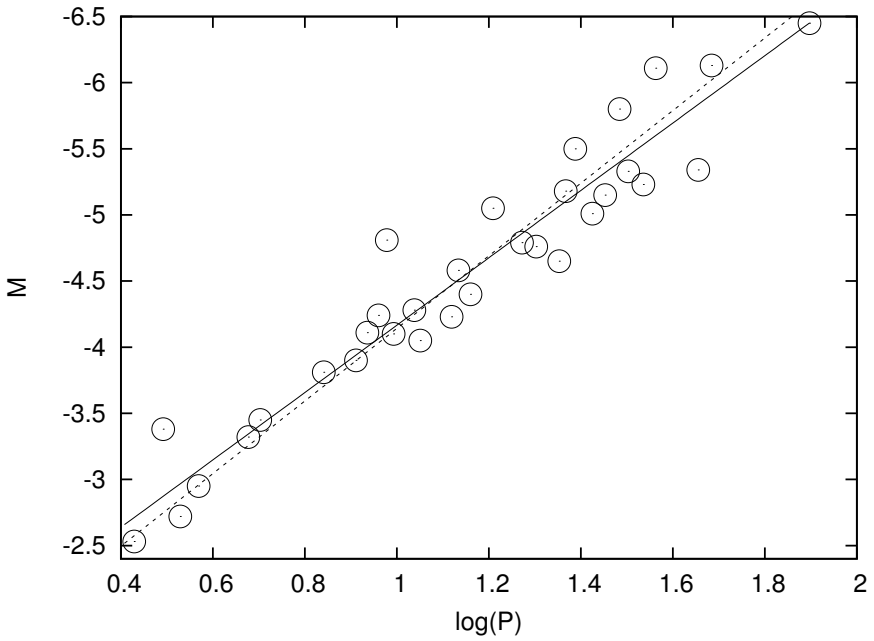

Figure 3.1: The relationship between the absolute magnitude, M, and the logarithm of the period (days), $\log P$, for Cepheids in the LMC (from Table 3.1). The solid line is the least-squares fit when $\log P$ is error free; the dashed line is the least-squares fit when M is error free.

From the data in Table 3.1 we can clearly determine the sums $\sum x_i$, $\sum y_i$, $\sum x_i y_i$ and $\sum x_i^2$. Therefore the values of a_0 and a_1 can be found because we have two equations in two unknowns.

The above notation is clumsy, so let us represent the average of any quantity as follows:

$$< x >= \frac{1}{n}\sum_{i=1}^{n} x_i$$

which allows us to write the above equations as

$$\begin{aligned} a_0 + a_1 < x > &= < y > \\ a_0 < x > + a_1 < x^2 > &= < xy > \end{aligned}$$

In matrix notation we can write these two simultaneous equations as

$$\begin{pmatrix} 1 & < x > \\ < x > & < x^2 > \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} < y > \\ < xy > \end{pmatrix}.$$

or more simply as

$$\boldsymbol{XA = Y}$$

from which the unknown coefficient matrix, $\mathbf{A}$, can be obtained:

$$\boldsymbol{A = (X^{-1})Y.}$$

where $\boldsymbol{X^{-1}}$ is the *inverse of the matrix* $\boldsymbol{X}$. In this simple case we do not need direct matrix inversion and algebraic manipulation easily leads to the solution

$$\begin{aligned} a_1 &= \frac{< xy > - < x >< y >}{< x^2 > - < x >^2} \\ a_0 &= < y > - a_1 < x > \end{aligned}$$

Applying these equations to our problem of Cepheid variables, we find $M = -2.547 \log P - 1.619$ (solid line in Fig. 3.1) . Given the observed values of $\log P_i$ and M_i in Table 3.1, we can calculate

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} \epsilon_i^2 = \frac{1}{n}\sum_{i=1}^{n}(a_0 + a_1 \log P_i - M_i)^2$$

using $a_0 = -1.619, a_1 = -2.547$, from which we can determine the standard deviation, σ, of the fit to the straight line. This turns out to be $\sigma = 0.28$ mag. Therefore we can state that the equation $M = -2.547 \log P - 1.619$ allows the absolute magnitude, M, of any Cepheid with known period to be determined with a standard deviation of 0.28 mag.

Now, we chose to assume on reasonable grounds that all the error resides in the absolute magnitude, M (the y value) and that $\log P$ (the x value) is free of error. If we had, instead, assumed that all the error can be attributed to $\log P$, we will obtain different results. Re-writing the equation as $\log P = \frac{1}{a_1}M - \frac{a_0}{a_1}$ and repeating the calculations gives $a_1 = -2.745$, $a_0 = -1.397$. This is quite different (dashed line in Fig. 3.1), emphasizing that we need to think carefully in such matters.

In the first case we minimized the errors in y, assuming that the errors in x were negligible compared to those in y. In the second case we minimized the errors in x, assuming that the errors in y were negligible compared to those in x. What do we do if the errors in x are comparable to those in y? This problem is beyond the scope of this course, but it will clearly lead to values of a_0 and a_1 which lie somewhere between the two cases we have considered.

Exercises

3.1.1 Write a program which reads the data of Table 3.1 from a file and calculates the linear least-squares coefficients a_0 and a_1 in $M = a_0 + a_1 \log P$. Plot the data points and the straight line thus calculated. [Answer: $a_0 = -1.619033, \quad a_1 = -2.547323$.]

3.2 χ^2 minimization

In the above discussion we assumed that the errors in the straight line fit belong to a single normal distribution with standard deviation σ. What happens if the errors belong to different normal distributions with different values of σ? This may happen if, for example, if the data were obtained with instruments of different precisions, then some points deserve more weight than others. In cases like these, we minimize χ^2 where

$$\chi^2 = \sum_{i=1}^{n}\left(\frac{y - y_i}{\sigma_i}\right)^2 .$$

Here y_i is the variable subject to errors, $y = a_0 + a_1 x_i$ is the function whose coefficients are required (x_i assumed to be free of error) and σ_i is the standard deviation of data point i. Least squares fitting is just a special case of χ^2 minimization with $\sigma_i = \sigma =$ a constant.

Our problem is to minimize χ^2 with respect to a_0 and a_1 for

$$\chi^2 = \sum_{i=1}^{n}\left(\frac{a_0 + a_1 x_i - y_i}{\sigma_i}\right)^2 .$$

Equating the partial differentials with respect to a_0 and a_1 to zero, we obtain the result

$$\begin{aligned}
\Delta &= \sum 1/\sigma^2 \sum x^2/\sigma^2 - \left(\sum x/\sigma^2\right)^2 \\
a_0 &= \left(\sum y/\sigma^2 \sum x^2/\sigma^2 - \sum x/\sigma^2 \sum xy/\sigma^2\right)/\Delta \\
a_1 &= \left(\sum 1/\sigma^2 \sum xy/\sigma^2 - \sum x/\sigma^2 \sum y/\sigma^2\right)/\Delta
\end{aligned}$$

When $\sigma_i^2 = \sigma^2 =$ constant, we recover the least squares result.

Exercises

3.2.1 Derive the equations for a_0 and a_1 using χ^2 minimization of $y_i + \epsilon_i = a_0 + a_1 x_i$ given the standard deviations σ_i for each of the n data points.

3.2.2 Write a program to calculate the χ^2 estimates of a_0 and a_1. Test your program by reading the data of Table 3.1 and setting $\sigma_i = \log P_i$. [Answer $a_0 = -1.511525$ and $a_1 = -2.651885$].

3.3 Uncertainties in the coefficients

The uncertainites in the coefficients a_0 and a_1 are quite easy to do derive, but rather tedious. First, we note that the errors in the coefficients depend only on the errors in y (if the y_i were free of error there would be no errors in the coefficients). This is a problem in propagation of errors - we want to know how the standard deviation of a_0 and a_1 depend on y_i. Therefore

$$\sigma_{a0}^2 = \left(\frac{\partial a_0}{\partial y_1}\right)^2 \sigma_1^2 + \left(\frac{\partial a_0}{\partial y_2}\right)^2 \sigma_2^2 + \ldots + \left(\frac{\partial a_0}{\partial y_n}\right)^2 \sigma_n^2 = \sum_{i=1}^{n} \left(\frac{\partial a_0}{\partial y_i}\right)^2 \sigma_i^2$$

and in the same way

$$\sigma_{a1}^2 = \sum_{i=1}^{n} \left(\frac{\partial a_1}{\partial y_i}\right)^2 \sigma_i^2$$

so we need partial derivatives with respect to each individual point, y_i. These are

$$\begin{aligned}
\frac{\partial a_0}{\partial y_i} &= \left(1/\sigma_i^2 \sum x^2/\sigma^2 - x_i/\sigma_i^2 \sum x/\sigma^2\right)/\Delta \\
\left(\frac{\partial a_0}{\partial y_i}\right)^2 \sigma_i^2 &= \frac{1/\sigma_i^2 \left(\sum x^2/\sigma^2\right)^2 - 2x_i/\sigma_i^2 \sum x^2/\sigma^2 \sum x/\sigma^2 + x_i^2/\sigma_i^2 \left(\sum x/\sigma^2\right)^2}{\Delta^2} \\
\sum_{i=0}^{n} \left(\frac{\partial a_0}{\partial y_i}\right)^2 \sigma_i^2 &= \frac{\sum 1/\sigma^2 \left(\sum x^2/\sigma^2\right)^2 - 2\sum x/\sigma^2 \sum x^2/\sigma^2 \sum x/\sigma^2 + \sum x^2/\sigma^2 \left(\sum x/\sigma^2\right)^2}{\Delta^2} \\
&= \frac{\sum 1/\sigma^2 \left(\sum x^2/\sigma^2\right)^2 - \sum x^2/\sigma^2 \left(\sum x/\sigma^2\right)^2}{\Delta^2} = \frac{\sum x^2/\sigma^2}{\Delta} \\
\frac{\partial a_1}{\partial y_i} &= \left(x_i/\sigma_i^2 \sum 1/\sigma^2 - 1/\sigma^2 \sum x/\sigma^2\right)/\Delta \\
\left(\frac{\partial a_1}{\partial y_i}\right)^2 \sigma_i^2 &= \frac{x_i^2/\sigma_i^2 \left(\sum 1/\sigma^2\right)^2 - 2x_i/\sigma_i^2 \left(\sum 1/\sigma^2\right)\left(\sum x/\sigma^2\right) + 1/\sigma^2 \left(\sum x/\sigma^2\right)^2}{\Delta^2} \\
\sum_{i=1}^{n} \left(\frac{\partial a_1}{\partial y_i}\right)^2 \sigma_i^2 &= \frac{\sum x^2/\sigma^2 \left(\sum 1/\sigma^2\right)^2 - 2\sum x/\sigma^2 \left(\sum 1/\sigma^2\right)\left(\sum x/\sigma^2\right) + \sum 1/\sigma^2 \left(\sum x/\sigma^2\right)^2}{\Delta^2} \\
&= \frac{\sum x^2/\sigma^2 \left(\sum 1/\sigma^2\right)^2 - \left(\sum 1/\sigma^2\right)\left(\sum x/\sigma^2\right)^2}{\Delta^2} = \frac{\sum 1/\sigma^2}{\Delta}
\end{aligned}$$

So we finally obtain $\sigma_{a0}^2 = \sum x^2/\sigma^2/\Delta$ and $\sigma_{a1}^2 = \sum 1/\sigma^2/\Delta$ with $\Delta = \sum 1/\sigma^2 \sum x^2/\sigma^2 - \left(\sum x/\sigma^2\right)^2$.

Exercises

3.3.1 Modify the program in Ex. 3.2.2 to calculate and print the standard deviations of a_0 and a_1. Test your program by reading the data of Table 3.1 and setting $\sigma_i = 0.1 \times \log P_i$. [Answer $a_0 = -1.511525$, $\sigma_{a0} = 0.034749$, $a_1 = -2.651885$, $\sigma_{a1} = 0.040433$.]

3.4 Multivariate least squares

In the last section we saw how we could obtain best estimates of the coefficients involving only two variables, x and y. It is sometimes the case that we have functional relationships involving more than two variables. Expanding on the example of the pulsating Cepheid stars, it can be shown that the period - luminosity law ought to depend somewhat on the temperature of the star. In other words, one should be able to obtain a more accurate absolute magnitude, M, (and thus a more accurate distance) if the temperature of each Cepheid is included in the relationship. The temperatures of stars are measured by the ratio of the brightness through blue and red filters. This ratio expressed in magnitudes is called $B - V$. Hence we may expect a relationship of the form

$$M = a_0 + a_1 \log P + a_2(B - V)$$

where, as before, the coefficients a_0, a_1, a_2 can be determined using the method of least squares. Let us see if we can apply the same technique to the general relationship $y_i + \epsilon_i = a_0 + a_1 x_{1i} + a_2 x_{2i}$, where we have written $M_i = y_i$, $\log P_i = x_{1i}$ and $(B - V)_i = x_{2i}$. Note that we assume $(B - V)$ to be free of error (which is not strictly true). We have

$$\begin{aligned} \epsilon_i &= a_0 + a_1 x_{1i} + a_2 x_{2i} - y_i \\ \epsilon_i^2 &= (a_0 + a_1 x_1 i + a_2 x_{2i} - y_i)^2 \\ S = \sum_{i=1}^{n} \epsilon_i^2 &= \sum (a_0 + a_1 x_{1i} + a_2 x_{2i} - y_i)^2 \end{aligned}$$

Minimizing S with respect to a_0, a_1, a_2

$$\begin{aligned} \frac{\partial S}{\partial a_0} &= 2 \sum (a_0 + a_1 x_{1i} + a_2 x_{2i} - y_i) \\ \frac{\partial S}{\partial a_1} &= 2 \sum (a_0 + a_1 x_{1i} + a_2 x_{2i} - y_i) x_{1i} \\ \frac{\partial S}{\partial a_2} &= 2 \sum (a_0 + a_1 x_{1i} + a_2 x_{2i} - y_i) x_{2i} \end{aligned}$$

Equating these to zero, you can see that we obtain three equations in the three unknowns, a_0, a_1, a_2:

$$\begin{aligned} a_0 + a_1 < x_1 > + a_2 < x_2 > &= < y > \\ a_0 < x_1 > + a_1 < x_1^2 > + a_2 < x_1 x_2 > &= < x_1 y > \\ a_0 < x_2 > + a_1 < x_1 x_2 > + a_2 < x_2^2 > &= < x_2 y > \end{aligned}$$

This time the simple algebraic solution is more laborious and it is best to use matrix methods. The above system of equations is written as

$$\begin{pmatrix} 1 & < x_1 > & < x_2 > \\ < x_1 > & < x_1^2 > & < x_1 x_2 > \\ < x_2 > & < x_1 x_2 > & < x_2^2 > \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} < y > \\ < x_1 y > \\ < x_2 y > \end{pmatrix},$$

or $\mathbf{XA} = \mathbf{Y}$. The matrix of unknown coefficients, $\mathbf{A}$, is obtained by finding the inverse of $\mathbf{X}$, so that $\mathbf{A} = (\mathbf{X^{-1}})\mathbf{Y}$. Matrix inversion is one of the most important numerical techniques. In python, matrix arithmetic, including matrix inversion, is available by importing the `numpy` module as explained in Chapter 1.

How would we program this in python? Actually, we may as well expand further and generalize the problem to the model

$$y_i + \epsilon_i = a_0 + a_1 x_{1i} + a_2 x_{2_i} + a_3 x_{3i} + ... + a_m x_{mi}$$

which means that the linear equation of the previous section is a special case with $m = 1$, while the case described above has $m = 2$. m is therefore the number of independent variables. Our program needs to construct two matrices, $\mathbf{X}$ and $\mathbf{Y}$:

$$\mathbf{X} = \begin{pmatrix} 1 & < x_1 > & < x_2 > & ... & < x_m > \\ < x_1 > & < x_1^2 > & < x_1 x_2 > & ... & < x_1 x_m > \\ < x_2 > & < x_1 x_2 > & < x_2^2 > & ... & < x_2 x_m > \\ ... & ... & ... & ... & ... \\ < x_m > & < x_1 x_m > & < x_2 x^m > & ... & < x_m x_m > \end{pmatrix}, \quad \mathbf{Y} = \begin{pmatrix} < y > \\ < x_1 y > \\ < x_2 y > \\ ... \\ < x_m y > \end{pmatrix}.$$

Since we will be dealing with arrays and matrices, we must import the `numpy` module. Here is the full least squares program.

```
 1  from numpy import *
 2
 3   # Multivariate least squares fit y = a0 + a1*x1 + a2*x2 + ... + am*xm
 4
 5   m = int(raw_input("Number of independent variables? "))
 6   m1 = m + 1
 7   fp = open("mlstsq.dat","r")
 8   x = zeros((m1,m1))
 9   y = zeros((m1,1))
10   xi = [0.0]*m1
11   n = 0
12   for line in fp:
13     xi[0] = 1.0
14     for j in range(1,m1):
15       xi[j] = float(line.split()[j-1])
16     yi = float(line.split()[m])
17     for j in range(m1):
18       for k in range(m1):
19         x[j,k] += xi[j]*xi[k]
20       y[j,0] += yi*xi[j]
21     n += 1
22   fp.close()
23   X = mat( x.copy() )
24   Y = mat( y.copy() )
25   A = X.I*Y
26   for j in range(m1):
27     print "A[%d] = %f" % (j, A[j])
```

Let's examine this program. In line 1 we import the `numpy` module for the reason mentioned above. In line 5 we ask the user to type in the number of independent variables. This is 1 for $y = a_0 + a_1 x_1$, 2 for $y = a_0 + a_1 x_1 + a_2 x_2$, etc. Note that the matrix `X` has dimension `m1` × `m1` where `m1` is one more than the number of independent variables, `m`. In line 7 we open the input file for reading and in line 22 we close it after we have read all the data. In line 8 we define `x` to be an `m1` × `m1` array and we set all its elements to zero using the `zeros` statement. Remember that the `Y` matrix is a *row* matrix with `m1` rows and one column. In line 9 we define the `y` array and set the array elements to zero. We will be using the `x` and `y` arrays to hold the running total as we loop through the data. In line 10 we define `xi` to be a list of zeros of size `m1`. The `xi` list will be used for reading the numbers, $x_{i1}, x_{i2}, ..., x_{im}$ from each line of the file. The variable `yi` will hold y_i which is the last number on each line.

From line 12 to line 21 we read and operate on each line of the file as it is read, using `n` to count the number of lines (i.e. the number of data points). We set `xi[0] = 1` because we need not only products such as $< x_1 x_2 >$, but also $< x_1 >=< 1 \times x_1 >=< x_0 x_1 >$. In line 15 we store each number in the line to the corresponding value of `xi`. In line 16 we store the last number in the line (the y_i value) to `yi`. In lines 18 and 19 we form the products $x_{ij} x_{ik}$ and add the result to the running total in array `x[j,k]`. In line 20 we add the product $y_i x_{ij}$ to the running total `y[j,0]`.

By the time we reach line 22 we have read all the lines in the file and we have these sums in arrays `x` and `y` as follows:

$$\texttt{x[j,k]} = \sum_{i=1}^{n} x_{ij} x_{ik} \quad \text{j,k = 0,1,2, ..., m}$$

$$\texttt{y[j,0]} = \sum_{i=1}^{n} x_{ij} y_i \quad \text{j = 0,1,2, ..., m}$$

These are just n times the required averages, $< x_j x_k >, < x_j y >$, but since we will be dividing by n after matrix inversion, there is really no need to divide these arrays by n. In lines 23 and 24 we convert the arrays to the corresponding matrices which then allows us to form the inverse of matrix $\mathbf{X}$ and multiply it by matrix $\mathbf{Y}$ to get the matrix of coefficients, $\mathbf{A}$. The coefficients are our final answers which we print out in a loop (lines 26 and 27).

Applying this method to our Cepheid problem, we obtain the following coefficients:

$$M = -2.145 - 3.117 \log P + 1.486(B - V)$$

with a residual standard deviation of $\sigma = 0.25$ mag in M. This is certainly a useful improvement over the simple linear relationship ($\sigma = 0.28$ mag) discussed in the previous section.

Exercises

3.4.1 Use the above program to calculate the linear least-squares coefficients in $M = a_0 + a_1 \log P + a_2(B-V)$ using the data of Table 3.1. [Answer: $a_0 = -2.1452$, $a_1 = -3.1173$, $a_2 = 1.4857$.]

3.5 Fitting a polynomial

It often happens that simple linear relationship of the form $y_i = a_0 + a_1 x_i$ is not a good representation of the data and that it is best represented by a polynomial of degree m, i.e.

$$y_i + \epsilon_i = a_0 + a_1 x_i + a_2 x_i^2 + a_3 x_i^3 + ... + a_m x_i^m.$$

This is just another example of the multivariate case described above. In the case of a quadratic, $y = a_0 + a_1 x + a_2 x^2$, we can simply put $x_1 = x, x_2 = x^2$ and use the above program to obtain the coefficients. For the general polynomial we just put $x_i = x^i$.

Exercises

3.5.1 The following program generates a cubic polynomial with coefficients in list a and with simulated observational errors:

```
import random
fp = open("polynomial.dat","w")
a = [1.0, -2.0, 3.0, -4.0] # Change the coefficients as desired.
n = 100                    # Number of data points.
for i in range(n):
  x = 0.01*i               # Value of x.
  y = a[0]
  for j in range(1,4):
    y += a[j]*x**j         # Value of y.
# Mess up y by adding some noise with standard deviation 0.1.
  y += random.gauss(0.0,0.1)
  fp.write("%8.3f %10.3f\n" % (x,y))
fp.close()
```

Use this program to generate (x, y) data in file `polynomial.dat`. Modify the general least-squares program discussed in the previous section to read this file and to fit a polynomial of any given degree. Test your program by comparing the resulting coefficients with those used to generate the file. The values will not be identical because of the added scatter.

3.6 Fourier fitting

There are many other uses for least squares. One of the most interesting is for *Fourier fitting*. Any periodic function can be described by a sum of Fourier components of the form

$$Y = a_0 + A_1 \sin(\omega t + \phi_1) + A_2 \sin(2\omega t + \phi_2) + A_3 \sin(3\omega t + \phi_3) + ... + A_m \sin(m\omega t + \phi_m)$$

where $\omega = 2\pi/P$, P is the period, ϕ is the phase in radians and $a_0, A_1, ..., A_m, \phi_1, ..., \phi_m$ are unknown coefficients. We can write this more compactly in the form

$$\begin{aligned} Y &= a_0 + \sum_{k=1}^{m} A_k \sin(k\omega t + \phi_k) \\ &= a_0 + \sum_{k=1}^{m} (A_k \cos\phi_k \sin(k\omega t) + A_k \sin\phi_k \cos(k\omega t)) \\ &= a_0 + \sum_{k=1}^{m} (a_{2k-1} \sin(k\omega t) + a_{2k} \cos(k\omega t)) \end{aligned}$$

So finally we have

$$Y = a_0 + a_1 \sin(\omega t) + a_2 \cos(\omega t) + a_3 \sin(2\omega t) + a_4 \cos(2\omega t) + ... + a_{2m-1} \sin(m\omega t) + a_{2m} \cos(m\omega t).$$

Suppose we have some periodic data, it could be for example the light curve of a Cepheid. We are given the time, t_i, and magnitude, y_i, of an observation and we have n such observations. We know the period, P, and therefore $\omega = 2\pi/P$. For any given observation we can put $x_{1i} = \sin(\omega t_i)$, $x_{2i} = \cos(\omega t_i)$, $x_{3i} = \sin(2\omega t_i)$, $x_{4i} = \cos(2\omega t_i)$, ..., etc. As you can see, this is the same kind of problem as before,

$$Y_i = y_i + \epsilon_i = a_0 + a_1 x_{1i} + a_2 x_{2i} + a_3 x_{3i} + ... + a_{2m-1} x_{(2m-1)i} + a_{2m} x_{2mi},$$

and we use least squares to determine the unknown coefficients, a_k. The python program is shown below.

```
#
# Program to fit a time series with Fourier components.
#
from numpy import *
import math

per = float(raw_input("Period? "))
w = 2.0*math.pi/per
nf = int(raw_input("Number of Fourier components? "))
fp = open("fourier.dat","r")
# m1 is the number of unknown coefficients.
m1 = 2*nf + 1
# Create empty matrices.
x = zeros((m1,m1))
y = zeros((m1,1))
xi = [0.0]*m1
n = 0
#
# Read (time, value) from each line of the file.
for line in fp:
  t = float(line.split()[0])
  yi = float(line.split()[1])
  xi[0] = 1.0
  for k in range(1,nf+1):
    xi[2*k-1] = math.sin(k*w*t)
    xi[2*k] = math.cos(k*w*t)
  for j in range(m1):
    for k in range(m1):
      x[j,k] += xi[j]*xi[k]
    y[j,0] += yi*xi[j]
  n += 1
fp.close()
# Copy to big matrices.
X = mat( x.copy() )
Y = mat( y.copy() )
# Invert X and multiply by Y to get coefficients.
A = X.I*Y
# Solution  is A0 + Sum[ Amp*sin(k*wt + phi) ]
print "a[0] = %f" % A[0]
for k in range(1,nf+1):
  amp = math.sqrt(A[2*k-1]**2 + A[2*k]**2)
  phs = math.atan2(A[2*k],A[2*k-1])
  print "amp[%d] = %f phi = %f" % (k, amp, phs)
```

Exercises

3.6.1 Write a program which asks for the period, the number of Fourier components, m, the constant, c_0 and the amplitude and phase of each Fourier component. Choose a suitable time interval between each data point so as to cover one cycle with about 100 points and write the time, t, and value, Y, to a file. Then use the above program to recover c_0, the amplitudes and phases.

3.7 Non-linear least squares

Suppose you were asked to find the least-squares solution to the constants, a_0, a_1, a_2 in the following model

$$y_i + \epsilon_i = a_0 + a_1 \exp(-(x - a_2)^2/a_3).$$

The problem here is that this functional relationship is *non-linear* in the unknown coefficients. Application of the usual technique leads to systems of equations which cannot be solved analytically.

The technique for solving such a problem is to start with approximate values of the unknown constants and to apply an algorithm which will iteratively improve the solution. A popular method is the *Levenberg-Marquardt algorithm*. This is an iterative technique that locates the minimum of a multivariate function that is expressed as the sum of squares of non-linear real-valued functions. It approximates better and better solutions by a combination of steepest descent and the Gauss-Newton method. When the current solution is far from the correct one, the algorithm behaves like a steepest descent method: slow, but guaranteed to converge. When the current solution is close to the correct solution, it becomes a Gauss-Newton method. A detailed analysis of the LM algorithm is beyond the scope of this course.

Chapter 4

Numerical integration and solutions of differential equations

4.1 Differential equations

A differential equation is an equation involving a function and its derivatives. For example, the equation

$$\frac{dy}{dx} = f(x, y)$$

has a solution given by the fundamental theorem of calculus,

$$y(x) = \int_a^x f(x, y)dx + C,$$

where C is a constant. Since C is undetermined, the differential equation alone is not enough to specify a particular solution. One way to select a particular solution is to give its value at some point, $y = y_0$ at $x = x_0$, which then allows C to be determined. We call this type of problem an *initial-value problem*. In real life problems we encounter many differential equations that either cannot be solved by elementary classical methods or for which the evaluation of the analytic solution is quite difficult. In such instances we resort to numerical methods of solution.

4.2 The Euler method

A first-order differential equation contains only the first derivative, i.e. $dy/dx = y' = f(x, y)$. If we are given a value $y = y_n$ at $x = x_n$, we can find the value, y_{n+1} at some neighbouring point, $x_{n+1} = x_n + h$ using Taylor's expansion

$$y_{n+1} = y_n + y'_n h + y''_n \frac{h^2}{2!} + y'''_n \frac{h^3}{3!} \ldots.$$

If we choose h sufficiently small, we can neglect terms higher than the first so that

$$y_{n+1} \approx y_n + y'_n h.$$

Given an initial value $y = y_0$ at $x = x_0$ and choosing step size h, we can calculate further values as follows:

$$\begin{aligned} y_1 &= y_0 + f(x_0, y_0)h \\ y_2 &= y_1 + f(x_1, y_1)h \end{aligned}$$

$$\begin{aligned} y_3 &= y_2 + f(x_2, y_2)h \\ &\dots \\ y_{n+1} &= y_n + f(x_n, y_n)h \end{aligned}$$

This is called *Euler's method* for numerical solution of a first order differential equation. Fig. 4.1 shows a geometrical interpretation of the method. Given an initial point (x_n, y_n) on the curve $y(x)$, the value of y_{n+1} is approximated by the straight line which passes through (x_n, y_n) and having slope $y'_n = f(x_n, y_n)$.

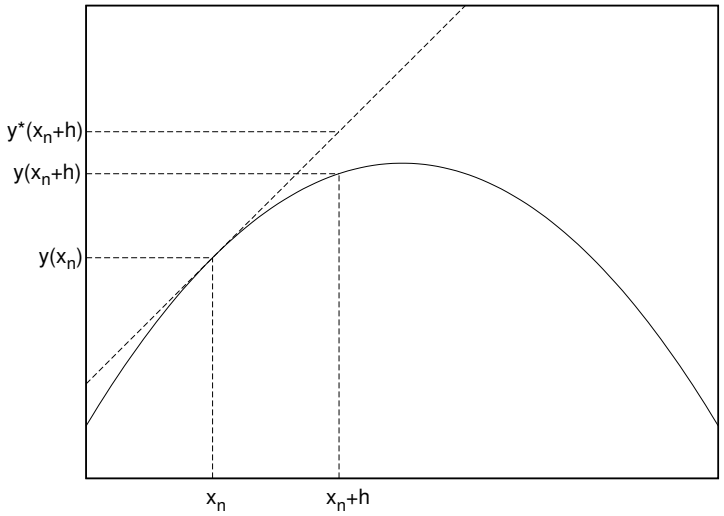

Figure 4.1: In the Euler method, the value of $y_{n+1} = y(x_n + h)$ is found by approximating the curve by a straight line passing through (x_n, y_n) and having a slope of $y'(x_n)$.

As an example, suppose we wish to find the value of y at $x = 10$ for

$$\frac{dy}{dx} = y - xy^2$$

given that $y = 0.1$ when $x = 0$. If we choose $h = 0.05$, we have the following solutions at $x_0 = 0$, $x_1 = 0.05$, $x_2 = 0.10$, $x_3 = 0.15$ ...:

$$\begin{aligned} y_0 &= 0.1 \\ y_1 &= y_0 + (y_0 - x_0 y_0^2)h = 0.10500 \\ y_2 &= y_1 + (y_1 - x_1 y_1^2)h = 0.11022 \\ y_3 &= y_2 + (y_2 - x_2 y_2^2)h = 0.11567 \\ &\dots \end{aligned}$$

We continue in this way until we arrive at the solution for $x_n = 10.0$, which is $y_n = 0.11103$.

It can be shown that for small h, the dominant error per step is proportional to h^2. To solve the problem over a given range of x, the number of steps needed is proportional to $\frac{1}{h}$ so it is to be expected that the total error at the end of the fixed step will be proportional to h (error per step times number of steps). For this reason, the Euler method is said to be first order. This makes the Euler method less accurate (for the same h) than other higher-order techniques. The Euler method can also be numerically unstable. This limitation, along with its slow convergence of error with h, means that the Euler method is not often used.

Exercises

4.2.1 Write a program which uses Euler's method to solve the differential equation

$$\frac{dy}{dx} = \frac{1}{1+x^2}$$

where $y = 1$ at $x = 0$. Start with a step size $h = 0.1$ and calculate values of (x, y) for each step until $x = 1.0$. This equation has the analytical solution $y = 1 + \arctan(x)$; plot a graph showing how the difference between the numerical and analytical solution varies as a function of x. What is the error at $x = 1$ when (a) $h = 0.1$ [answer 0.025], (b) $h = 0.01$ [answer 0.0025], (c) $h = 0.001$ [answer 0.00025].

4.2.2 Use Euler's method to solve the

$$\frac{dy}{dx} = y - xy^2$$

where $y = 0.1$ at $x = 0$ using a step size $h = 0.05$ and find the value of y when $x = 10$ [answer $y = 0.111$].

4.3 Differential equations of second and higher order

So far we have considered a single equation containing just one derivative of the first order. Usually we are interested in solving the initial-value problem with equations containing second or higher order derivatives. These are handled by transforming the original equation to several first-order differential equations. For example, equation

$$\frac{d^2y}{dx^2} = f(x, y)$$

can be transformed into two first-order equations by introducing a new unknown $v = dy/dx$. So therefore

$$\begin{aligned} \frac{dy}{dx} &= v \\ \frac{dv}{dx} &= f(x, y). \end{aligned}$$

In order to solve a first order equation we require one initial condition. To solve this second-order equation we require two initial conditions. In this example, we need to specify the value of y and $v = dy/dx$ at some value of x.

Any differential equation can be transformed into a series of first-order differential equations. The solution by Euler's method is just an extension of that for a single equation. Given initial values $y = y_0$ and $v = v_0$ at $x = x_0$ and choosing step size, h, we can calculate further values as follows:

$$\begin{aligned} y_1 &= y_0 + v_0 h \\ v_1 &= v_0 + f(x_0, y_0)h \\ y_2 &= y_1 + v_1 h \\ v_2 &= v_1 + f(x_1, y_1)h \\ &\dots \\ y_{n+1} &= y_n + v_n h \\ v_{n+1} &= v_n + f(x_n, y_n)h \end{aligned}$$

Consider, for example, the differential equation

$$\frac{d^2y}{dt^2} + 10\frac{dy}{dt} + 100y = 100|\sin(t)|$$

with initial conditions $y = 0.1, \quad dy/dt = -0.5$ at $t = 0$. We let $y_0 = y$ and $y_1 = dy/dt$ so that

$$\frac{dy_1}{dt} + 10y_1 + 100y_0 = 100|\sin(t)|.$$

So the original second order differential equation becomes

$$\begin{aligned} \frac{dy_0}{dt} &= y_1 \\ \frac{dy_1}{dt} &= 100|\sin(t)| - 10y_1 - 100y_0 \end{aligned}$$

with initial conditions $y_0 = 0.1, y_1 = -0.5$ at $t = 0$. A simple python code to solve this problem is the following:

```
import math
h = 0.01
t = 0.0
y0 = 0.1
y1 = -0.5
while t < 10.0:
  y0 = y0 + y1*h
  y1 = y1 + (100.0*abs(math.sin(t)) - 10.0*y1 - 100.0*y0)*h
  t = t + h
  print t,y0,y1
```

In order to make the code quite general so that a differential equation of any order can be solved, it is convenient to use a python list to store values of the variables $y_0, y_1, y_2, ...$ as shown in the revised code below. In this code all we need to do to implement any arbitrary system of first-order differential equations is to define the right hand side of each equation in function `fun(n,x,y)`. In calling this function, `n` is the equation number to be evaluated. In this example line 8 returns `y[1]` because the first equation is $dy_0/dt = y_1$ and because `y[1]` stores y_1. Line 12 evaluates and returns the RHS of the second differential equation, $100|\sin(t)| - 10y_1 - 100y_0$. We could add on more equations as required. The function `euler` implements the Euler algorithm for each equation. Whenever we call `fun`, we use the original values and store the updated values in a temporary list, `yt`, in line 19. When we have finished evaluating all the differential equation, we copy the updated values in the temporary list, `yt`, to the original list, `y`, in line 34.

The system of equations are started with the initial values of t and y in lines 24 and 25. The program also needs to know the step size (line 27). In line 31 we begin the integration loop. Each time the loop executes we integrate through one step by calling `euler` (line 33) and increasing the value of t by the step size (line 35). We write out the values of $t, \quad y_0$ and y_1 in the loop. Results are plotted in Fig. 4.2.

```
 1 import math
 2
 3 # Functions to integrate.
 4 # Place the RHS of 1st-order diff equations here.
 5 def fun(n,x,y):
 6 # The 1 st diff eqn:
 7   if n == 0:
 8     return y[1]
 9
10 # The 2nd diff eqn:
11   if n == 1:
12     return 100.0*abs(math.sin(x)) - 10.0*y[1] - 100.0*y[0]
13 # etc....
14
15 # Integrates system of diff equations using Euler's method.
16 def euler(order, h, x, y):
17 # Put updated values in temporary list
18   for i in range(order):
19     yt[i] = y[i] + h*fun(i,x,y)
20   return
21
22 fp = open("euler.dat","w")
23 # Initial values
24 t = 0.0
25 y = [0.1, -0.5]
26 # Step size
27 h = 0.01
28 # Order is number of 1-st order diff equations.
29 order = 2
30 yt = [0.0]*order # Temporary list
31 while t < 10.0:
32 # Integrate by one step.
33   euler(order, h, t, y)
34   y = yt  # Update y from temporary list
35   t += h
36   fp.write("%10.5f %10.5f %10.5f\n" % (t, y[0], y[1]))
37 fp.close()
```

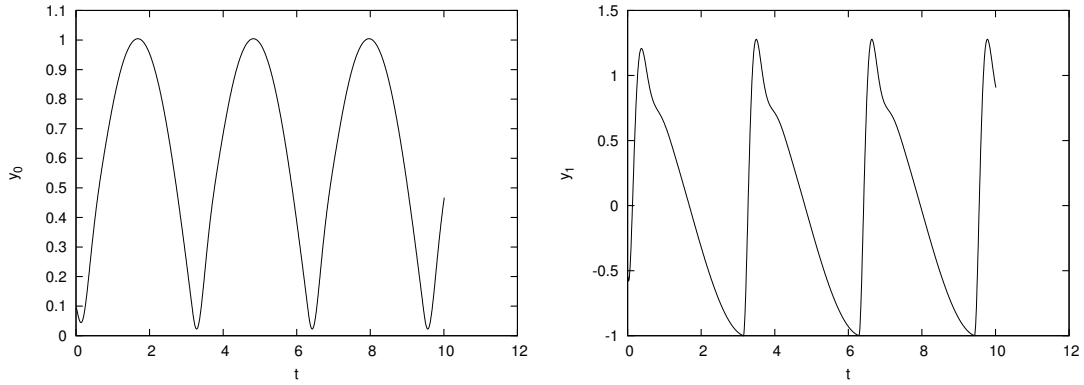

Figure 4.2: Solution to the second-order differential equation discussed in Section 3.

Exercises

4.3.1 Consider a planet orbiting the Sun. By Newton's law of gravity, the components of acceleration in the x- and y-directions are

$$\frac{d^2x}{dt^2} = -\frac{GMx}{r^3}$$
$$\frac{d^2y}{dt^2} = -\frac{GMy}{r^3}$$

where $r^2 = x^2 + y^2$, G is the gravitational constant and M is the mass of the Sun. Re-write these equations as four first-order differential equations. (a) These four differential equations require four initial conditions. Choosing units such that $GM = 1$, the initial conditions at $t = 0$ are $x = 1.0$, $y = 0.0$, $dx/dt = 0.0$, $dy/dt = 1.0$. Solve the differential equations using Euler's method and write values of t, x, y for $0 < t < 7$ using a step size of 0.001 in t. Plot the the orbit (y as a function of x). [Answer: a circle of unit radius centered at the origin].
(b) Repeat above solution using initial conditions $x = 1.0$, $y = 0.0$, $dx/dt = 0.0$, $dy/dt = 1.1$ at $t = 0$ for $0 < t < 10$ and the same step size. Plot the orbit. [Answer: an ellipse].

4.4 The Runge-Kutta method

In the Euler method we used only the first two terms in the Taylor expansion

$$y_{n+1} = y_n + y'_n h + y''_n \frac{h^2}{2!} + y'''_n \frac{h^3}{3!} \ldots .$$

The Runge-Kutta method assumes

$$y_{n+1} = y_n + h(w_1 k_1 + w_2 k_2 + w_3 k_3 + \ldots)$$

where w_i is a weight and k_i is a function of y and previous values of k. The idea behind Runge-Kutta methods is to select w_i and k_i so that

$$h(w_1 k_1 + w_2 k_2 + w_3 k_3 + \ldots) \approx y'_n h + y''_n \frac{h^2}{2!} + y'''_n \frac{h^3}{3!} \ldots .$$

The fourth-order Runge-Kutta method (RK4) is the most widely used algorithm for solving an ordinary first-order differential equation. It is given by the following equation:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f(x_n + h, y_n + hk_3) \end{aligned}$$

Thus, the next value y_{n+1} is determined by the present value, y_n, plus the product of the step size, h, and an estimated slope. The slope is a weighted average of slopes:

- k_1 is the slope at the beginning of the interval;
- k_2 is the slope at the midpoint of the interval, using slope k_1 to determine the value of y at the point $x_n + h/2$ using Euler's method;
- k_3 is again the slope at the midpoint, but now using the slope k_2 to determine the y-value;
- k_4 is the slope at the end of the interval, with its y-value determined using k_3.

In averaging the four slopes, greater weight is given to the slopes at the midpoint:

$$\text{slope} = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}.$$

The RK4 method is a fourth-order method, meaning that the error per step is on the order of h^5, while the total accumulated error has order h^4.

Below is an implementation of the RK4 algorithm for the problem of Ex. 2.1. We use a list, `y`, because we want to extend the code to differential equations of higher order. As in the Euler method, we use a function `fun` which evaluates and returns the RHS of each 1st order differential equation. In the `rungeKutta` function we start by saving the original value of `y[n]` to a temporary variable, `yn`, because we do not want to use the updated value of `y[n]` when evaluating each equation. We return with the update value of y[n]. In the calling program, note that we use a temporary list, `yt`, to store the updated values of `y` (line 41), and that we call `rungeKutta` with the original values of `y`. When all the equations have been evaluated, we update `y` (line 43).

The Runge-Kutta method is more accurate than the Euler method. For Ex. 2.1 and $h = 0.1$, the maximum error is 0.01; for $h = 0.01$ it is 0.001; for $h = 0.001$ it is 0.0001.

```
 1 import math
 2
 3 # Functions to integrate.
 4 # Place the RHS of 1st-order diff equations here.
 5 def fun(n,x,y):
 6 # The 1 st diff eqn:
 7   if n == 0:
 8     return y[1]
 9
10 # The 2nd diff eqn:
11   if n == 1:
12     return 100.0*abs(math.sin(x)) - 10.0*y[1] - 100.0*y[0]
13
14 # etc....
15
16 # 4-th order Runge-Kutta Method.  Do not change anything.
17 def rungeKutta(n,x,y,f,h):
18      yn = y[n]               # Original value of y[n].
19      k1 = f(n,x,y)
20      y[n] = yn + 0.5*h*k1    # Function expects a list.
21      k2 = f(n, x+0.5*h, y)
22      y[n] = yn + 0.5*h*k2
23      k3 = f(n, x+0.5*h, y)
24      y[n] = yn + h*k3
25      k4 = f(n,x + h, y)
26 #    Must use original value here.
27      return yn + h*(k1 + 2.0*k2 + 2.0*k3 + k4)/6.0
28
29 fp = open("rk4.dat","w")
30 # Order is the number of 1st-order diff equations.
31 order = 2
32 # Initial conditions:
33 t = 0.0
34 y = [0.1, -0.5]
35 yt = [0.0]*order  # Temporary list
36 # Step size
37 h = 0.01
38 while t < 10.0:
39   for i in range(order):
40 #   We use original y in function, not yt.
41     yt[i] = rungeKutta(i,t,y,fun,h)
42   t += h
43   y = yt
44   fp.write("%8.3f %16.7e %16.7e\n" % (t,y[0],y[1]))
45 fp.close()
```

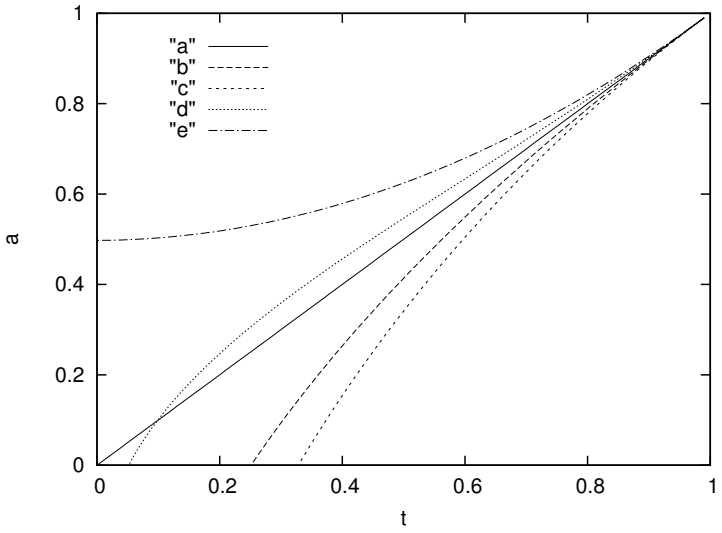

Figure 4.3: Backwards evolution of the cosmic scale factor.

Exercises

4.4.1 Although the actual size of our Universe is "unmeasurable", one usually describes distances at different times (which will change due to expansion or contraction) in terms of a *cosmic scale*, $R(t)$, with its present value denoted by $R_0 = R(t_0)$. Additionally, one can normalize $R(t)$ by its present value and define a *cosmic scale factor*, $a(t) = R(t)/R_0$. The rate of change of $a(t)$ with time, $\dot{a}(t) = H_0$, is the present value of the Hubble parameter $H_0 = 73$ km s^{-1} per megaparsec. The evolution of the scale factor with time is governed by the equation,

$$\frac{da}{dt} = \sqrt{1 - \Omega_0 + \frac{\Omega_M}{a} + \frac{\Omega_R}{a^2} + a^2\Omega_\lambda},$$

where $\Omega_0 = \Omega_M + \Omega_R + \Omega_\lambda$, is the *total density parameter*, Ω_M is the relative density of matter (including dark matter), Ω_R is the relative radiation energy density and Ω_λ is the relative dark energy density. In this equation the unit of time has been chosen so that the present age of the universe, $t_0 = 1$. It is thought that currently the values are $\Omega_M \approx 0.3$, $\Omega_R \approx 0$, $\Omega_\lambda \approx 0.7$.

Use RK4 to determine the *backwards* evolution of the scale factor from its present value to the start of the Big Bang at $t = 0$. Use a step size $h = -0.01$ to work backwards from the initial condition $t = 1$, $a = 1$ and stop when t reaches zero. Plot a as a function of t for the following choices:

(a) An empty Universe ($\Omega_M = \Omega_R = \Omega_\lambda = \Omega_0 = 0$).

(b) The critical density case $\Omega_0 = 1$ with $\Omega_M = 1, \Omega_R = \Omega_\lambda = 0$.

(c) $\Omega_0 = 2$ with $\Omega_M = 2.0, \Omega_R = \Omega_\lambda = 0$.

(d) $\Omega_0 = 1$ with $\Omega_M = 0.3$, $\Omega_R = 0$, $\Omega_\lambda = 0.7$.

(e) The steady state model with $\Omega_0 = 1$, $\Omega_\lambda = 1$, $\Omega_M = \Omega_R = 0$.

Plot all on one graph; compare your results with Fig. 4.3.

4.4.2 The gravitational potential, ϕ, is given by Poisson's equation $\nabla^2\phi = 4\pi G\rho$ where G is the gravitational constant and $\rho(r)$ is the mass density. For the spherically-symmetric case, we have

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\phi}{dr}\right) = 4\pi G\rho$$

For stellar interiors the relationship $\rho = \lambda\phi^n$, where λ and n are constants, is a reasonable approximation. Putting $r = ax$ where

$$a^2 = \frac{(n+1)K\lambda^{(1-n)/n}}{4\pi G}$$

results in

$$\frac{1}{x^2}\frac{d}{dx}\left(x^2\frac{d\phi}{dx}\right) = -\phi^n.$$

At the centre of the star $x = 0$, $\phi = 1$ and $d\phi/dx = 0$.

(a) Show that when $x \approx 0$, $\phi = 1 - \frac{1}{6}x^2$ and $d\phi/dx \approx -\frac{1}{3}x$.

(b) Plot ϕ as a function of x for $n = 3$ and determine the value of x at the surface of the star when $\phi = 0$. [Answer $x = 3.142$]

4.5 Boundary value problems

So far we have only considered solutions for differential equations where the boundary conditions are specified at the same point. This is called an *initial value problem* (IVP). We may encounter a differential equation where the boundary values are specified at different points. This is called a *boundary value problem* (BVP). As an example, suppose we want to know the deflection, y, of a beam supported at two ends, $x = 0$ and $x = L$. The differential equation is

$$\frac{d^2y}{dx^2} = ax(x - L)$$

where a is a constant. We know that the deflection will be zero at the two ends, so the boundary conditions are $y(0) = 0$ and $y(L) = 0$. In this problem we are given two boundary conditions, as required to solve a second order differential equation, but they are specified at two different points. In order to solve this problem as an IVP, we need to know $y'(0)$ or $y'(L)$.

One way to solve a BVP like this is to guess the missing boundary condition, thereby turning it into an IVP. In the example, we guess the value of $y'(0)$, solve this IVP and obtain $y(L)$. In general, $y(L)$ will not have the required value. Using trial and error or some scientific approach, one makes another guess at $y'(0)$ and tries to get as close to the required boundary value as possible. This is called the *shooting method* because it is like aiming for a distant target with a gun.

We could try plotting the value of $y(L)$ for different initial guesses of $y'(0)$ and read off from the graph what value of $y'(0)$ is required to give the correct value of $y(L)$. If the graph turns out to be a straight line, we only need two guesses. Suppose that the straight line can be written as $y'(0) = a_0 + a_1 y(L)$. For our first guess $y'_1(0) = a_0 + a_1 y_1(L)$ and for the second guess $y'_2(0) = a_0 + a_1 y_2(L)$. Solving for the coefficients we obtain

$$y'(0) = y'_1(0) + \left(\frac{y'_2(0) - y'_1(0)}{y_2(L) - y_1(L)}\right)(y(L) - y_1(L)).$$

So putting $y(L) = 0$ in this equation we can determine the required value of $y'(0)$.

Once we have obtained a better guess by using the value given by the straight line, we can use this guess to see how close it takes us to the required value of $y(L)$. In general, we will find that the

answer does take us closer to the required value, but not close enough for our purposes. In that case we define a value, ϵ, which defines how close we want to get to the required value. If the value we obtain after using the straight line is $y_n(L)$, then we want to continue until $|y_n(L) - y(L)| < \epsilon$. If $y_n(L)$ does not satisfy this condition, we repeat the procedure until it is eventually satisfied.

Let us see how we can program this BVP using the 4-th order Runge-Kutta code we have developed. If we let $y_0 = y$ and $y_1 = dy/dx$, we have

$$\frac{dy_0}{dx} = y_1, \qquad \frac{dy_1}{dx} = ax(x - L).$$

In the example we will use $a = 5 \times 10^{-4}$ and $L = 10$. The code is shown below.

```
 1 import math
 2
 3 # Functions to integrate.
 4 # Place the RHS of 1st-order diff equations here.
 5 def fun(n,x,y):
 6 # The 1 st diff eqn:
 7   if n == 0:
 8     return y[1]
 9 # The 2nd diff eqn:
10   if n == 1:
11     return 0.0005*x*(x - 10.0)
12
13 # 4-th order Runge-Kutta Method.  Do not change anything.
14 def rungeKutta(n,x,y,f,h):
15      yn = y[n]               # Original value of y[n].
16      k1 = f(n,x,y)
17      y[n] = yn + 0.5*h*k1    # Function expects a list.
18      k2 = f(n, x+0.5*h, y)
19      y[n] = yn + 0.5*h*k2
20      k3 = f(n, x+0.5*h, y)
21      y[n] = yn + h*k3
22      k4 = f(n,x + h, y)
23 #    Must use original value here.
24      return yn + h*(k1 + 2.0*k2 + 2.0*k3 + k4)/6.0
25
26 # Integrates diff equation from x to x1
27 def ode(x,x1,h,y,fun,order):
28   while x < x1:
29     for i in range(order):
30       yt[i] = rungeKutta(i,x,y,fun,h)
31     x += h
32   return
33
34 # Calculates straight line coefs and returns value at y01
35 def fitline(y01,g1,g2,f1,f2):
36   a1 = (g2[1] - g1[1])/(f2[0] - f1[0])
37   a0 = g1[1] - a1*f1[0]
38   return a0 + a1*y01
```

```
39
40 order = 2
41 # Starting and ending value of x
42 x0 = 0.0
43 x1 = 10.0
44 #
45 # Initial value with guess for y1(x0)
46 y0_0 = 0.0
47 y1_0 = 0.1  # A guess
48 #
49 # Required value for y0(x1)
50 y0_1 = 0.0
51 #
52 # Tolerance, step size
53 eps = 1.0e-4
54 h = 0.01
55 yt = [0.0]*order  # Temporary list
56 #
57 # Initial guess
58 g1 = [y0_0, y1_0]
59 while 1:
60 # Integrate with first guess
61   y = [g1[0],g1[1]]
62   ode(x0,x1,h,y,fun,order)
63 # Test if required precision has been attained.
64   if abs(y0_1-yt[0]) < eps:
65     break
66   f1 = [yt[0], yt[1]]
67 # Take a second guess
68   g2 = [g1[0], g1[1]+0.1]
69   y = [g2[0],g2[1]]
70 # Integrate with second guess
71   ode(x0,x1,h,y,fun,order)
72   f2 = [yt[0],yt[1]]
73 # Fit a straight line and return new guess.
74   g1[1] = fitline(y0_1,g1,g2,f1,f2)
75 print "Initial conditions:   ", g1
76 print "Boundary conditions: ", yt
```

We start at 40 by defining the order of the system (there are two 1st order equations, so `order = 2`). The starting and ending values of x are 0 and 10 respectively. The initial conditions are $y_0(0) = 0$, but $y_1(0)$ is unknown and we have guessed $y_1(0) = 0.1$. Any other guess will be equally valid. The required value of $y_0(L) = 0$ (line 50). We define a tolerance of 1.0×10^{-4}, meaning that the program will run until the initial guess leads to a value of $y_0(L)$ which differs from the required value by less than this amount. In line 54 we define the step size and create a temporary list, `yt`. We keep the initial conditions for the first guess in the list `g1`.

In line 59 we begin the loop which will continue until the tolerance level has been met. We start with the first guess, `g1`, and integrate the differential equations all the way to $x = L$ by calling the function `ode`. After the call, `yt` contains the boundary conditions at $x = L$. We store this in list `f1`

(Line 66). Then we take a second guess, `g2`, in which we simply increment $y_1(0)$ by 0.1 (line 68). We could have changed $y_1(0)$ by some other value, this does not matter because all we want to do is to take any two points so that we can fit a straight line. We then call `ode` again, this time using the second guess as initial values and store the boundary conditions in `f2` (line 72).

So now we have two guesses, `g1` and `g2`, at the initial conditions which result in the boundary conditions `f1` and `f2` respectively. With this information we can fit a straight line to `g` as a function of `f` by calling `fitline`. `fitline` calculates the straight line coefficients, `a0` and `a1` in $y'(0) = a_0 + a_1 y(L)$ or `g[1] = a_0 + a_1f[0]` and returns a better guess for $y'(0)$, which is used as the next initial guess.

What happens if the dependence of $y'(0)$ on $y(L)$ is not a straight line? In this case we will obtain a value for $y'(0)$ which is closer to the true value than our initial guess. On the next iteration we obtain an even better solution. The procedure is repeated until the specified tolerance is met.

The shooting method is conceptually easy to visualize, but sometimes it is not the best method of solving a boundary value problem because the boundary conditions at the end point might be very sensitive to the initial boundary conditions. This can sometimes be alleviated by choosing a suitable value of $x = x_f$ between the initial value, $x = x_0$ and the final value $x = x_1$ (x_f is called a *fitting point*). The shooting method is used to integrate from x_0 to x_f and also to integrate between x_1 and x_f. The idea is to choose initial values at x_0 and x_1 so that the values all match at the fitting point.

A very powerful method begins by transforming the system of differential equations to a system of *finite difference* equations. For example, we can replace the differential equation

$$\frac{dy}{dx} = f(x, y)$$

with an algebraic equation relating function values at two points k and $k - 1$:

$$y_k - y_{k-1} = (x_k - x_{k-1})\, f\left(\frac{1}{2}(x_k + x_{k-1}), \frac{1}{2}(y_k + y_{k-1})\right).$$

We are given values at some starting point $k = k_0$ and at some end point, $k = k_1$. We then have a system of coupled linear equations which can be solved at all values of k. This is called the *relaxation method*.

Exercises

4.5.1 Find a solution for the boundary value problem

$$\frac{d^2y}{dx^2} = 0.0005x(x - 10)$$

with $y(0) = 0$ and $y(10) = 0$ and plot the beam displacement as a function of x. [Answer: $y'(0) = 0.04166$ at $x = 0$.]

4.5.2 Solve the boundary value problem

$$\frac{d^2y}{dx^2} = -\frac{1}{(1 + y)^2}$$

with $y(0) = y(1) = 0$. [Answer: $y'(0) = 0.43215$.]

Chapter 5

Searching for hidden periods in data

5.1 Introduction

Many stars are *variable* in light, i.e. the brightness of the star changes with time. By plotting the brightness as a function of time, it is sometimes possible to determine the period of variation just by visual inspection, as in the example below. To find the period of variation, one determines the time

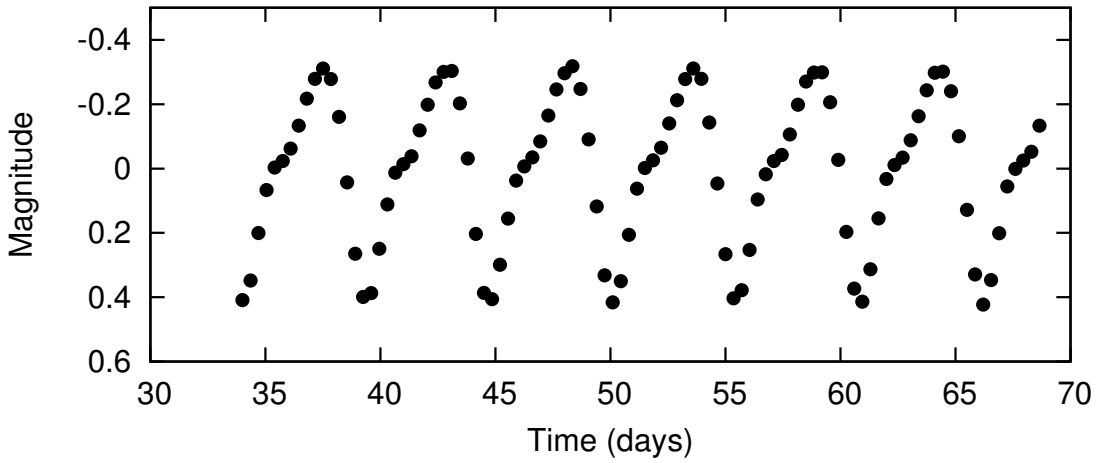

Figure 5.1: Brightness variation of the star δ Cephei.

of the first maximum $T_1 = 37.50$, and the time of the last maximum $T_2 = 64.45$, and the number of cycles between the two, $n = 5$. The period, P, is:

$$P = \frac{64.45 - 37.50}{5} = 5.39 \quad \text{days.}$$

Clearly, the larger the number of cycles, the more accurate the resulting period.

To show that the period is correct, one can fold the variation into cycles. The result is a *phase* diagram, as shown in Fig. 5.2.

But what if our observations were not so well sampled? After all, stars can only be observed at night, so there will be gaps in the data due to the daylight hours. There will also be gaps due to cloudy weather. Real observations are more likely to resemble Fig. 5.3 instead of Fig. 5.1. Now it is much more difficult to identify the maxima. Some other way must be found to determine the period. One could guess the period, plot a phase diagram such as that of Fig. 5.2 and choose the period which gives the best looking phase diagram. Proceeding in this way, and knowing that the

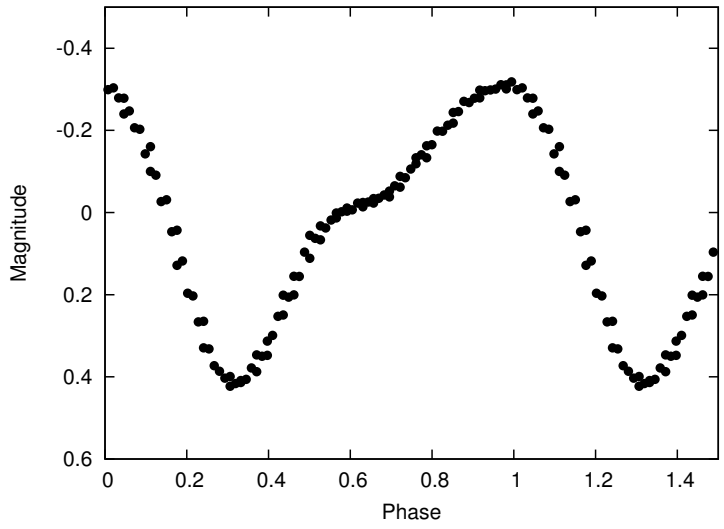

Figure 5.2: Brightness variation folded with period $P = 5.39$ days.

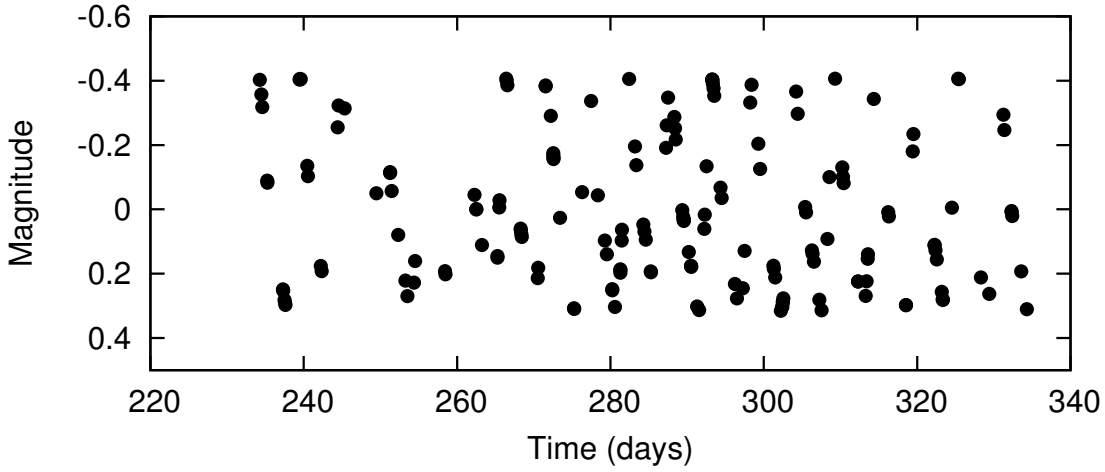

Figure 5.3: Actual observations of the star δ Cephei.

period is somewhere around 5.4 days, we can construct a large number of phase diagrams close to this period (Fig. 5.4) and choose the one with smallest scatter.

Constructing phase diagrams in this way is a rather laborious enterprise. It would be better if we could just calculate the scatter about the best-fitting curve without necessarily plotting the phase diagram. To do this we need to know the functional relationship between the brightness, V, and the time, t. We see that the curve is roughly sinusoidal (though there is definitely a fair degree of asymmetry), so we may assume that the true brightness of the i-th observation is given by $V_{T,i} = a_0 + A\sin(\omega t_i + \phi)$. The *angular frequency* is $\omega = \frac{2\pi}{P}$. We cannot measure $V_{T,i}$, instead we measure V_i, where $V_{T,i} = V_i + \epsilon_i$, and ϵ_i is the error in the observation. Thus we write

$$\begin{aligned} V_i + \epsilon_i &= a_0 + A\sin(\omega t_i + \phi) \\ &= a_0 + A\cos\phi\sin\omega t_i + A\sin\phi\cos\omega t_i \\ &= a_0 + a_1\sin\omega t_i + a_2\cos\omega t_i \end{aligned}$$

where the amplitude $A = \sqrt{a_1^2 + a_2^2}$. This kind of equation should now be familiar. Putting $x_i = \sin\omega t_i$, and $y_i = \cos\omega t_i$, we see that this is just another example of multivariate least squares.

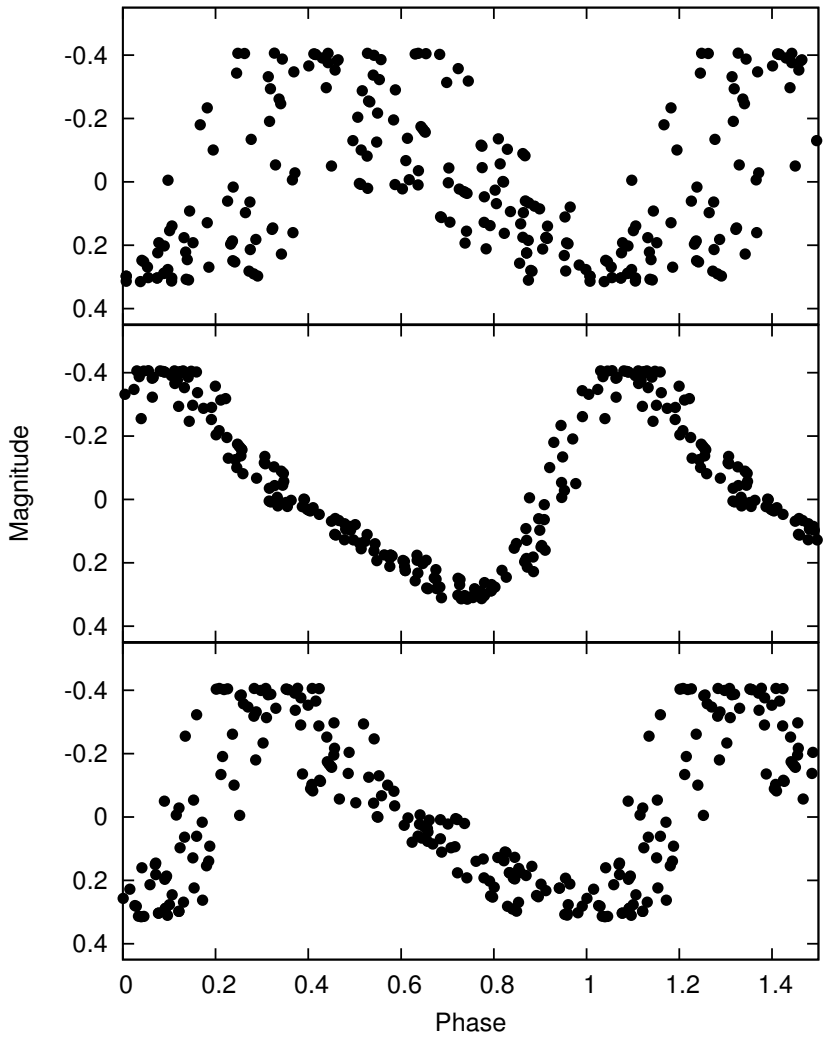

Figure 5.4: Observations of Fig. 3 phased with $P = 5.3$ (top), 5.4 (middle) and 5.5 (bottom) days.

To find the solution which minimizes the sum of the squares of the errors, $\sum \epsilon_i^2$, we equate the partial derivatives with respect to a_0, a_1 and a_2 to zero and we end up with the following simultaneous equations:

$$\begin{aligned} < V > &= a_0 + a_1 < \sin\omega t > + a_2 < \cos\omega t > \\ < V \sin\omega t > &= a_0 < \sin\omega t > + a_1 < \sin^2\omega t > + a_2 < \sin\omega t \cos\omega t > \\ < V \cos\omega t > &= a_0 < \cos\omega t > + a_1 < \sin\omega t \cos\omega t > + a_2 < \cos^2\omega t > \end{aligned}$$

We could proceed in the usual way and solve these equations by matrix inversion. Since we will be trying a large number of periods, and since we need a complete solution for each trial period, this will require a great deal of computation. Fortunately, some good approximations can be made which will allow an approximate solution without matrix inversion.

First, we assume that we have a large number of data points, in which case the mean of $\sin\omega t$ and $\cos\omega t$ will be very close to zero. This is simply due to the fact that the sine and cosine curves spend an equal time being positive as being negative. Making this approximation we have from the

first equation $a_0 \approx < V >$. In other words, the constant term is just the average value, $< V >$. It is convenient to remove this constant term from the data, $v_i = V_i - < V >$, so the remaining two equations become

$$\begin{aligned} < v \sin \omega t > &= a_1 < \sin^2 \omega t > + a_2 < \sin \omega t \cos \omega t > \\ < v \cos \omega t > &= a_1 < \sin \omega t \cos \omega t > + a_2 < \cos^2 \omega t > \end{aligned}$$

Note that $< \sin \omega t \cos \omega t > = \frac{1}{2} < \sin 2\omega t >$ should also be very close to zero. What is the average of $\sin^2 \theta$ over a cycle? This is how we work it out:

$$< \sin^2 \theta > = \frac{\int_0^{2\pi} \sin^2 \theta d\theta}{\int_0^{2\pi} d\theta} = \frac{\frac{1}{2}\int_0^{2\pi}(1 - \cos 2\theta) d\theta}{2\pi} = \frac{1}{2}\frac{2\pi}{2\pi} = \frac{1}{2}$$

We can see that $< \cos^2 \theta > = \frac{1}{2}$ as well. Making these approximations, we have

$$a_1 \approx 2 < v \sin \omega t >, \qquad a_2 \approx 2 < v \cos \omega t > .$$

The squared amplitude is

$$A^2 = a_1^2 + a_2^2 = 4\left(< v \sin \omega t >^2 + < v \cos \omega t >^2 \right).$$

To determine the standard deviation, we note that

$$\begin{aligned} \epsilon_i &= a_0 + a_1 \sin \omega t_i + a_2 \cos \omega t_i - V_i \\ &= a_1 \sin \omega t_i + a_2 \cos \omega t_i - v_i \end{aligned}$$

Multiplying both sides by ϵ_i and forming averages we find

$$< \epsilon^2 > = a_1 < \epsilon \sin \omega t > + a_2 < \epsilon \cos \omega t > - < \epsilon v >$$

The terms $< \epsilon \sin \omega t >, < \epsilon \cos \omega t >$ are both approximately zero because the errors are random and $\sin \omega t$ and $\cos \omega t$ average out to zero. Therefore

$$< \epsilon^2 > \approx - < \epsilon v >$$

In terms of $v_i = V_i - < V >$, our original model is

$$v_i + \epsilon_i = a_1 \sin \omega t_i + a_2 \cos \omega t_i$$

Multiplying this equation by v_i, we have

$$< v^2 > + < \epsilon v > = a_1 < v \sin \omega t > + a_2 < v \cos \omega t > .$$

Using $< \epsilon v > \approx - < \epsilon^2 >$ and substituting for a_1, a_2, we finally have

$$< \epsilon^2 > \approx < v^2 > -2 < v \sin \omega t >^2 -2 < v \cos \omega t >^2 .$$

Given a particular value of $\omega = 2\pi/P$, we can evaluate the quantities on the right-hand side of this equation to obtain the standard deviation about the best-fitting sinusoid, $\sigma = \sqrt{< \epsilon^2 >}$. By calculating σ for a large number of trial periods and by plotting σ as a function of P, the period for which σ is a minimum can be found. In Fig. 5.5 we show such a plot for the example discussed above. The period which has minimum σ is found to be $P = 5.376$ days.

In practise, astronomers prefer not to use σ in this plot. We note that

$$< \epsilon^2 > = \sigma^2 = < v^2 > - \frac{1}{2} A^2$$

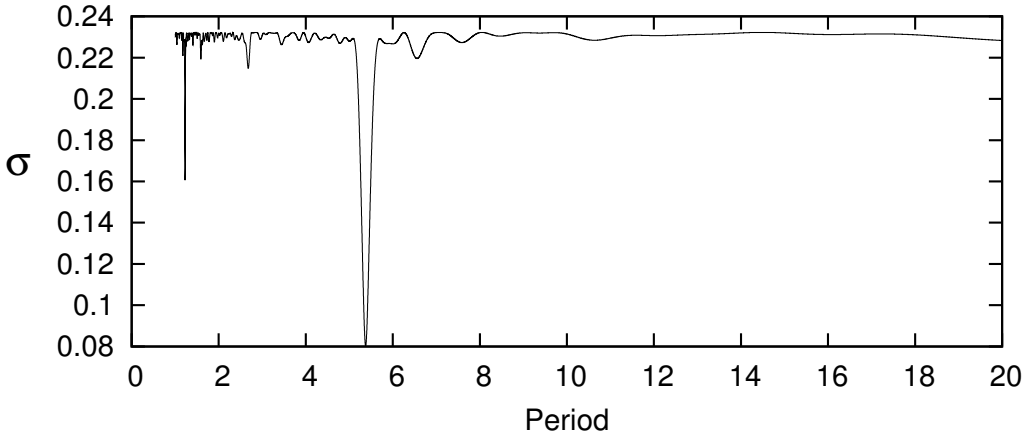

Figure 5.5: The standard deviation about the best-fitting sinusoid, σ, as a function of the period in days.

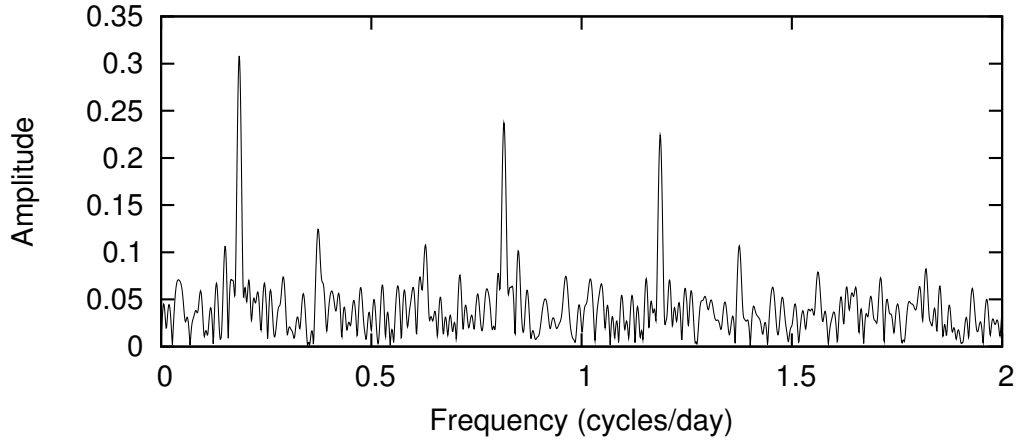

Figure 5.6: The periodogram of the data shown in Fig. 5.3. The strongest peak occurs at $f_0 \approx 0.18$, but notice the peaks at $f \approx 0.82$ and and $f \approx 1.18$. These secondary peaks at $1 - f_0$ and $1 + f_0$ are called *aliases*.

and since $< v^2 >$ is a constant, the smaller the value of σ the larger the amplitude, A. Instead of plotting σ as a function of P, it is convenient to plot A as a function of the frequency $f = 1/P$. This is called the *periodogram*. The periodogram for our example is shown in Fig. 5.6. The highest amplitude occurs at $f = 0.186$ cycles/day which is the same as $P = 5.376$ days.

The frequencies selected for the periodogram should be sufficiently closely spaced so as to resolve the peaks. If the step size is too large, a peak might be missed, but if it is too small the program might take too long to execute. It can be shown that the width of a peak is roughly $1/\Delta t$ where Δt is the time span covered by the data (latest time - earliest time). To ensure that a peak is defined by at least four points, the frequency step size should be about $\frac{1}{4}\Delta t$ or smaller.

If the data were sampled at equal time intervals, δt, the highest frequency that can be extracted is one-half the sampling frequency or $1/2\delta t$. This is called the *Nyquist frequency*. For unequally sampled data, the Nyquist frequency is undefined. The mean sampling frequency is $n/\Delta t$, where n is the number of data points. One could define the mean Nyquist frequency as $n/2\Delta t$, but in most cases this too low. A reasonable value is to search all frequencies up to a maximum of $n/\Delta t$.

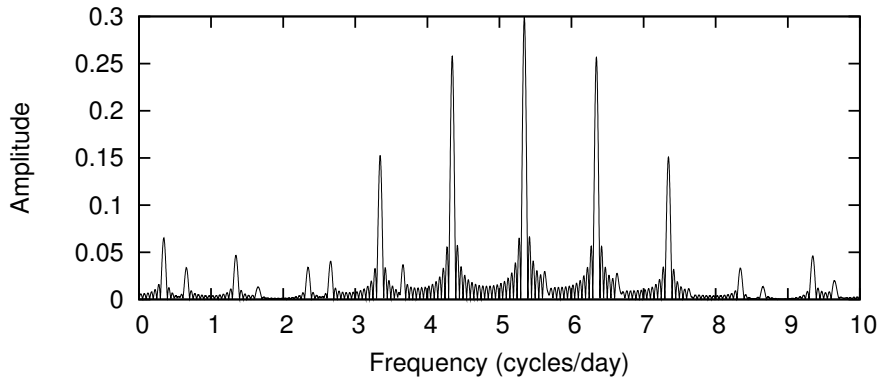

Figure 5.7: Periodogram of a star with $P = 0.187$ days showing the aliasing caused by daytime gaps in the data.

Exercises

5.1.1 Write a program that reads the the time (in days) and the magnitude from file `cepheid.dat`. The program should ask for the period and then write the phase and magnitude to another file. Ensure the phase, ϕ, is in the range $< \phi < 1.5$ and plot the phase diagram. The true period is between $5.2 < P < 5.6$ days. Determine the period by visual inspection of the phase diagram.

5.1.2 Write a program which asks for the starting frequency, stopping frequency and frequency step size. The program then reads a file containing the time, t, and magnitude, V, for a variable star. Store these values in separate python lists. For each frequency, f, the program calculates the amplitude $A = 2\sqrt{< v \sin \omega t >^2 + < v \cos \omega t >^2}$, and writes f, A to an output file. Apply the program to the data in file `cepheid.dat` and

(a) plot the periodogram and visually locate the frequency of maximum amplitude;

(b) use the polynomial-fitting program to fit a quadratic to A as a function of f. Find the maximum of the quadratic and hence the best value of f.

5.2 Aliases

In Fig. 5.6, which is the periodogram of the data in Fig. 5.3, we see that in addition to the large peak at $f_1 = 0.186$ cycles/day, there are smaller peaks at $f_2 = 0.814$ and $f_3 = 1.186$ cycles/day. Note that $f_2 = 1 - f_1$ and $f_3 = 1 + f_1$. These two peaks arise because there are *gaps* in the data set. Because the data in Fig. 5.3 could only be obtained at night, there is a gap of about one day between each measurement (sometimes more than one day if the night was cloudy). Because of these gaps, one could squeeze in an extra cycle or leave out a cycle and still get a reasonable fit to the sinusoid. This effect is responsible for the two weaker peaks which are called *aliases*.

Aliases are separated from the true peak by the frequency at which the data were sampled. In astronomical observations done from a single site, the sampling frequency is 1 cycle/day so that one always obtains aliases separated by 1 cycle/day from the true frequency. Aliasing occurs in any data set which is not completely sampled. The more frequent the gaps, the larger the amplitude of the alias peaks. In some cases it is not possible to determine a unique frequency because the aliases are so strong.

Fig. 5.7 shows a periodogram which illustrates the concept of aliases rather clearly. This periodogram is of a star with a period $P = 0.187$ days or $f = 5.345$ cycles/day. Notice that the highest

peak does, indeed have a frequency $f = 5.345$, but notice the 1 cycle/day aliases at 4.345 and 6.345 cycles/day. In this data set, there are frequent gaps lasting for 2 days so that the 2 cycle/day aliases at 3.345 and 7.345 cycles/day are also quite strong. The other, smaller peaks are also due to aliasing.

5.3 Harmonics

In the model we have been using, we assume that the data is well represented by a pure sinusoid

$$V_i + \epsilon_i = a_0 + A \sin(\omega t_i + \phi).$$

This may be a good approximation in many cases, but not in others as shown in Fig. 5.8. This is the light curve of the RR Lyrae star SS Leo which shows a very steep rise in brightness followed by a gradual decline. The actual data from which the phase diagram of Fig. 5.8 was constructed is shown in Fig. 5.9. Notice the very uneven data sampling and the large gaps.

The periodogram of SS Leo is shown in Fig. 5.10. First of all, notice that there are several peaks of almost equal height at $f = 0.60$, 1.60, 2.20, 2.60, 3.20 and 4.20 cycles/day. These correspond to periods of 1.67, 0.62, 0.45, 0.38, 0.31, and 0.24 days. It is clear, however, that at least some of the frequencies 0.60, 1.60 and 2.60 cycles/day are aliases owing to the daily gaps in the data. The true period is 0.6263441 days which is $f = 1.60$ cycles/day. Therefore the peaks at 0.60 and 2.60 are aliases. In this case it is almost impossible to select the correct period owing to severe aliasing.

The other sequence of peaks at 2.23, 3.23 and 4.23 cycles/day also seem to be mostly aliases, but this time it is due to the highly non-sinusoidal nature of the variations. The light curve is poorly represented by a sine wave, but it is better described by a sine wave with frequency $f = 0.61$ cycles/day together with the *first harmonic* at twice the frequency, $f = 1.20$ cycles/day. Therefore 3.20 and 4.20 are the aliases of the first harmonic. Thus the periodogram is well described by just two frequencies, 0.60 and 1.20 cycles/day, and their 1 cycle/day aliases. The second harmonic at $f = 1.80$ cycles/day is also present, but is much weaker.

This example, although extreme, illustrates the limitations of the method. To describe the light curve of SS Leo accurately requires several Fourier components and not just the fundamental frequency. As a result the power that would otherwise be concentrated in the fundamental frequency is spread over several components, lowering the amplitudes. In this example we have, in addition, observations which are sparse and include large gaps. This further complicates the periodogram which now includes not only the two Fourier components of highest amplitude (the fundamental and first harmonic), but also their 1 cycle/day and 2 cycle/day aliases. Under these circumstances, it is best to use a different period finding technique.

Exercises

5.3.1 Calculate and plot the periodogram of the data in file `dsct.dat` between 0 and 10 cycles/day.

(a) Find the most likely frequency [Answer: 4.789 cycles/day].

(b) Use the Fourier analysis program in Chapter 3, Section 4 to compute the Fourier coefficients which best fit these data at this frequency. [Answer: $a_0 = 8.456144$, $A_1 = 0.099895$, $\phi = 2.343396$ radians]

(c) Use the program in Ex. 5.1.1 to construct a phase diagram and plot the phase diagram with the fitted Fourier curve.

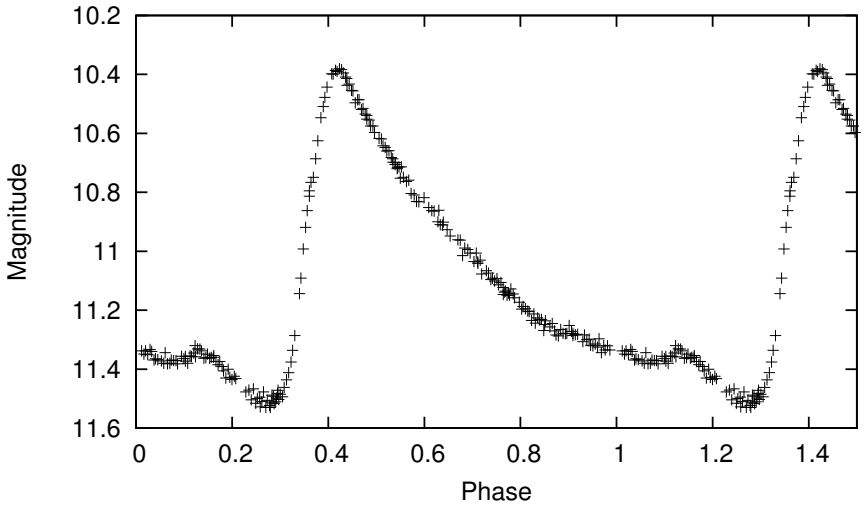

Figure 5.8: Light curve of SS Leo (period 0.6263441 days) showing large departure from a pure sinusoid.

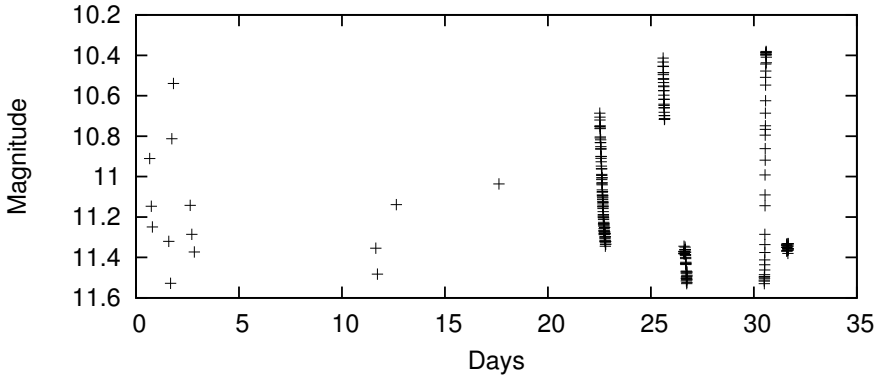

Figure 5.9: Data of Fig. 5.8 plotted versus time in days.

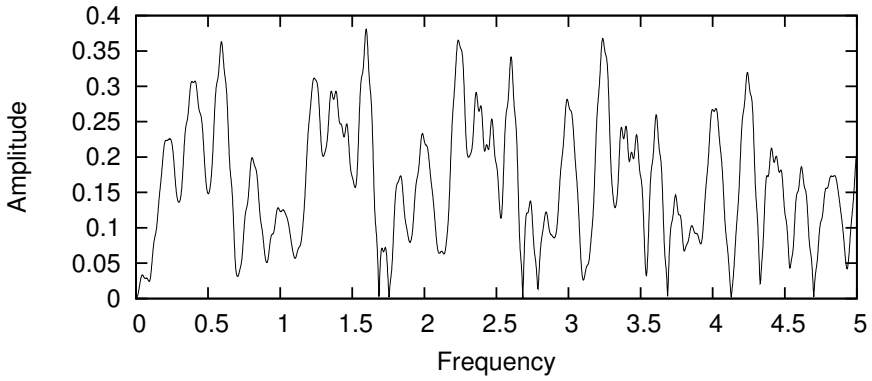

Figure 5.10: Periodogram of data in Fig. 5.9.

5.4 Multiperiodic variations

Harmonics (multiples of the fundamental frequency) as discussed above are not really independent periods since they are just a consequence of departure from a pure sinusoid. Many stars pulsate in more than one period. For example some δ Scuti stars have 3, 4, 5 or even more periods. We extract the frequencies in such data in the same way as we do for the case of just one period - by using the periodogram.

Fig. 5.11 shows the periodogram of a δ Scuti star with three periods. Note that one can immediately pick out the frequency with the highest amplitude at $f_1 = 7.156$ cycles/day and another one at a frequency close to 5 cycles/day. The periodogram is very clean and we note that the 1 cycle/day aliases are very small. This is due to the fact that the star was observed from three continents round the clock - from Australia, South Africa and Chile.

In order to be able to detect further frequencies, we need to remove the frequency of highest amplitude from the original data. This process is called *prewhitening*. We fit a Fourier curve to the data using the known frequencies and subtract the fitted Fourier curve from the data. In other words, we find the least-squares solution to

$$\begin{aligned} V_i + \epsilon_i &= a_0 + a_1 \sin 2\pi f_1 t_i + a_2 \cos 2\pi f_1 t_i \\ &+ a_3 \sin 2\pi f_2 t_i + a_4 \cos 2\pi f_2 t_i \\ &+ a_5 \sin 2\pi f_3 t_i + a_6 \cos 2\pi f_3 t_i + \ldots \end{aligned}$$

where $f_1, f_2, f_3, \ldots$ are the frequencies so far extracted. Using the solution, we calculate the value of V at time, t, subtract the calculated value, V_{ci}, from the observed value, V_i and write t_i and $V_i - V_{ci}$ to a new file. A new periodogram is calculated and the next frequency is determined until the peak is not high enough to distinguish it from noise. The program listed below implements this procedure.

Fig. 5-11 to 5-14 is an example of this procedure. The periodogram of the original data in Fig. 5.11 gives $f_1 = 7.156$ cycles/day. Fitting and removing the sinusoid with this frequency gives the periodogram in Fig. 5-12 from which we find the next frequency, $f_2 = 4.921$ cycles/day. Fitting the original data by f_1 and f_2 gives Fig. 5-13 from which we obtain $f_3 = 5.345$ cycles/day. Removing f_1, f_2, f_3 from the original data gives Fig. 5-14. The highest peak is only about twice the height of the noise and we cannot therefore be certain that it is real. There is no definite criterion to judge whether a peak is real or not, but a general rule of thumb is not to accept peaks unless they are at least 4 times higher than the background noise. We conclude that there are three certain frequencies in this star: $f_1 = 7.156$, $f_2 = 4.921$ and $f_3 = 5.345$ cycles/day.

We chose a data set where the aliases were of very low amplitude. More often than not, the aliases are at least half the height of the main peak and it may become impossible to decide on whether a frequency is real or an aliases, especially if it is close to an aliases of another frequency.

Exercises

5.4.1 Using the program of Ex. 1.2 (calculating the periodogram) and the program listed on page 10 (prewhitening with given frequencies), determine the frequencies present in file `multiperiod.dat`. These are the data used to generate Fig. 5.11 to 5.14 and you should obtain the same frequencies.

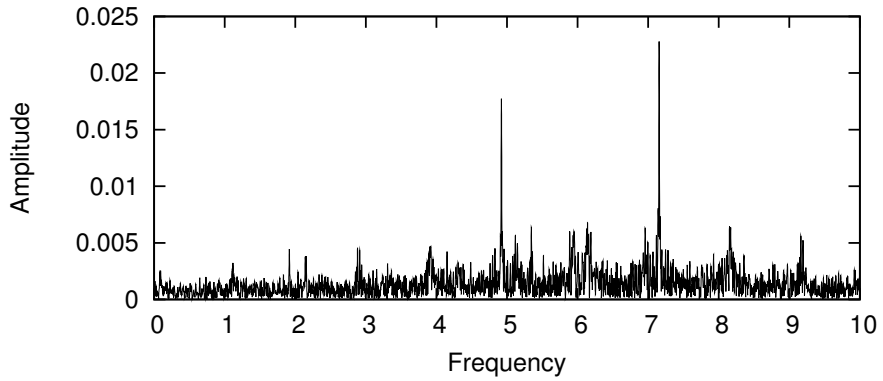

Figure 5.11: Periodogram of a star with three frequencies.

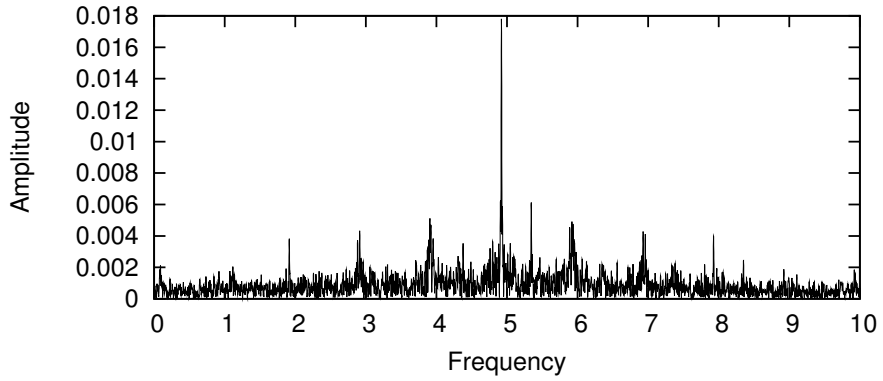

Figure 5.12: The periodogram of data prewhitened by $f_1 = 7.156$ cycles/day.

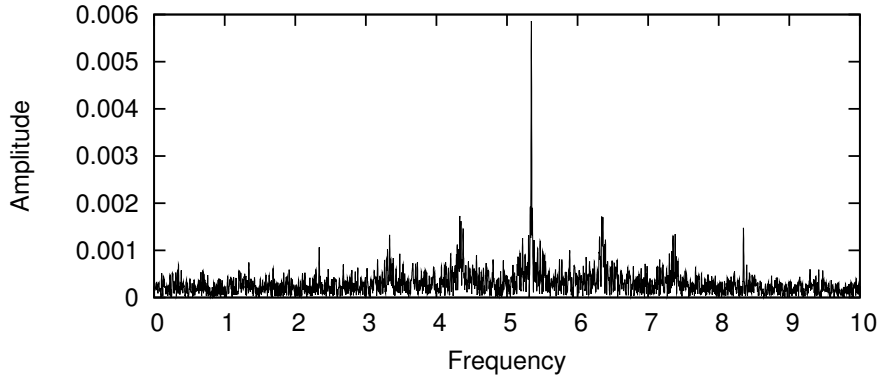

Figure 5.13: The periodogram of data prewhitened by f_1 and $f_2 = 4.921$ cycles/day.

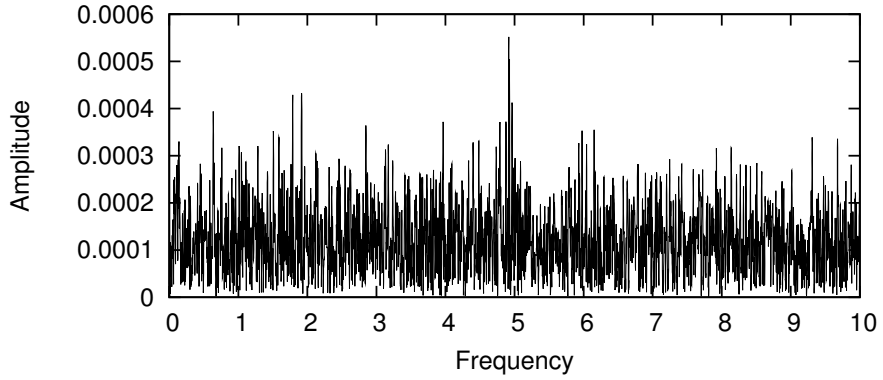

Figure 5.14: The periodogram of data prewhitened by f_1, f_2 and $f_3 = 5.345$ cycles/day.

```
from numpy import *
import math
twopi = 2.0*math.pi

# Function to calculate the magnitude from the Fourier components.
def fourier(nf,f,A,t):
  yc = 0.0
  for k in range(1,nf+1):
    wt = twopi*f[k-1]*t
    yc += A[2*k-1]*math.sin(wt)+ A[2*k]*math.cos(wt)
  return yc + A[0]

f = []
nf = int(raw_input("Number of frequencies? "))
for i in range(nf):
  str = raw_input("Frequency %d? " % (i+1))
  f.append(float(str))
fp = open("data.in","r")
m1 = 2*nf + 1
x = zeros((m1,m1))
y = zeros((m1,1))
xi = [0.0]*m1
n = 0
day = []
mag = []
for line in fp:
  t = float(line.split()[0])
  yi = float(line.split()[1])
  day.append(t)
  mag.append(yi)
  xi[0] = 1.0
  for k in range(1,nf+1):
    wt = twopi*f[k-1]*t
    xi[2*k-1] = math.sin(wt)
    xi[2*k] = math.cos(wt)
  for j in range(m1):
    for k in range(m1):
      x[j,k] += xi[j]*xi[k]
    y[j,0] += yi*xi[j]
  n += 1
fp.close()
X = mat( x.copy() )
Y = mat( y.copy() )
A = X.I*Y
# Remove Fourier fit from data and write prewhitened data.
fp = open("data.pre","w")
for i in range(n):
  dmag = mag[i]- fourier(nf,f,A,day[i])
  fp.write("%10.4f %10.4f\n" % (day[i],dmag))
fp.close()
```