THE UNIVERSITY OF SYDNEY

Exercises — 2

Implementing the PM technique

by Simon J. Murphy

1. Introduction

As you saw in the lectures, the phase modulation (PM) technique is one way of determining the orbital parameters of a binary system. In this Exercise, you will learn to extract time delays from orbital motions using a *Kepler* binary system containing a δ Sct pulsator. The PM technique offers handy visualisations of the binary orbit, and the signals from different pulsation frequencies are easily combined.

The PM technique works by looking for shifts in the phases of pulsation frequencies as the light travel time between the star and Earth varies over the binary orbit. As with the FM technique, it is relatively easy to measure an orbital period, but obtaining other orbital parameters is quite difficult. Fortunately, with PM there is some existing software to give you quick approximations for the orbital parameters. Still, obtaining robust orbital solutions with uncertainties requires substantial computational power, so we will skip that in this Exercise. Instead you will learn how to extract time delays yourself in a Jupyter Notebook. It will then be possible to use code that you write and apply it to other systems. Today, you'll have a choice of systems to work with: (i) the high amplitude pulsator KIC 11754974, which has an M-dwarf companion; (ii) KIC 7917485, which has a planetary-mass companion; (iii) KIC 9594022, which was the subject of Exercise 1 on FM; or (iv) KIC 9651065, a more eccentric binary. Since the code you'll write can be readily applied to any system, if time allows you can analyse more than one system.

Warning: here is a repeat of the warnings and advice from Exercise 1: this Exercise is extremely challenging! You will be using `python` to analyse one or more binary systems. If you are not very familiar with `python`, *please* team up with somebody who is. You will struggle otherwise. To make the Exercise easier, some of the harder tasks have been done for you and you will be able to copy-paste the necessary functions from this document. You only have 90 minutes, so if you are struggling, please ask for help. The instructions here are quite explicit to try to guide you through the process.

Why does this Exercise rely so heavily on `python`? `Python` is the most widely-used computer language in astronomy. Much of modern astronomy is conducted in `python`, and an understanding of `python` will open a lot of doors for you. There are excellent libraries to use, as well as a lot of open source software. As we saw in the data retrieval exercises, many official tools for obtaining and analysing lightcurves are written in `python`.

2. Learning Objectives

At the end of this tutorial you should be able to

- Extract frequencies with `Period04` and use a `python` function to fit them.
- Write and use a `python` function to extract time delays.
- Compute weighted-average time delays for a binary system.

3. Plan

You should work in pairs or groups of three for these exercises, even if you each work on a Jupyter Notebook on your own computer. Don't worry if you don't finish the whole Exercise. The instructions are set up so that you can continue at a later time, if you wish.

You should begin by starting a new Jupyter Notebook, giving it a name, then following the instructions below. You'll also need to open `Period04`. Locate the light curve files in the online directory.

4. Extracting the pulsation frequencies

Hopefully you remember how to extract pulsation frequencies in `Period04` from Exercise 1. If you do, go ahead and extract the **strongest two *real*** frequencies for whichever target you decided to work on. **Do not** extract orbital sidelobes. Then, skip ahead to Section 5. If you do not remember, continue reading this section.

Your first task is to import the light curve to `Period04`. You should check the light curve (*Display graph*) to make sure it is okay. Do any outliers need removing? You'll notice this is a full four years of *Kepler* data in long-cadence mode. That's a lot of data!

Calculate a Fourier transform of the light curve. You'll need to think carefully about the frequency range – it will need to be high enough to help you distinguish Nyquist aliases. Use a *low* step rate to make the computation quicker.

Decide which are the strongest real peaks in the Fourier transform. Remember that real peaks will have higher amplitudes than their Nyquist counterparts. You can add them by right-clicking and choosing *Add peak to frequency list*, but be sure you get the peak and not another one nearby – use zoom carefully. You **do not** need to also select the Nyquist aliases of each real peak. If you select the real peak, and *calculate* a fit of this frequency to the light curve, its Nyquist alias will disappear. But beware: if you choose the Nyquist alias, the real peak will disappear! If you select a Nyquist alias now by mistake, then later in the Exercise you'll end up rediscovering *Kepler*'s orbit around the Sun – still a fun thing to do, but not our main goal today. Extract just the two strongest peaks for now.

Using the *Fit* tab, select the peaks you have extracted (using the check-boxes) and click *Calculate*. This determines a linear least-squares fit of the selected frequencies to the light curve. After this, click *Improve all*, which will run a non-linear least-squares fit of these frequencies to improve the frequencies, amplitudes and phases.

Now return to the *Fourier* tab, change the *Calculations based on:* from *Original data* to *Residuals at original*, and recalculate the Fourier transform over the same frequency range. Notice that the real peaks and their Nyquist aliases are now gone.

5. Extracting time delays with `python`

While it is important to learn to use `python` to analyse astronomical data sets, your ability to use `python` should not be a barrier to completing this Exercise. Every task here will have hints and/or solutions available that you can refer to if you are stuck. As always, `Google` is your friend! E.g. if you are not sure what the syntax is for `np.average` (the program for computing an average, optionally with weights, using the `numpy` library), you can quickly find the answer on `Google`. Do try to understand what the code does, even if you couldn't have written it yourself. The solutions are well commented.

5.1 Getting started

Begin by importing `numpy`. For plotting later, you'll also need to `import matplotlib.pyplot as plt`, and you'll need to import `optimize` from the `scipy` package. Each light curve file has only two columns: time (d) and brightness (mag). Read this in so that you have a time array and a flux array [hint-5,1]. You will also need the frequencies f_1, f_2, which you should copy directly from `Period04` into a frequency array, or list. Hereafter, the instructions deviate increasingly from those of Exercise 1. [solution-5,1]

5.2 Preparing to optimize

Soon we are going to optimize a fit of f_1, f_2 to the *Kepler* light curve. Unlike in the FM method, the amplitudes of those frequencies are unimportant here, except for computing a weighted-average time delay later. The uncertainties will also be less important in this exercise – you did enough error propagation in Exercise 1, so we won't do more today.

Remember that our time array has a large, non-zero mean and it is bad practice to optimize on an array of times with such a large mean. We could subtract the mean, like we did in Exercise 1, but instead, in this Exercise we'll just calibrate the zeroth time, `time[0]`, to zero. That is, we will start counting our times at a new $t_0 = 0$. [hint-5,2] [solution-5,2]

5.3 A function that optimizes a multi-sine-wave fit

In Exercise 1, we used an optimizer function to find the best fitting frequencies (and amplitudes, and phases). We will use this one again. Copy the two functions-5,3 into your notebooks and run them. Then, run the following line, like last time, to get the optimized values.

```
a_out, freq_out, phi_out, a_err, freq_err, phi_err = optimizer(time,mag,freqs)
```

where `time` is your array of times, `mag` is the equally-sized array of magnitudes, and `freqs` is your frequency list. There is no need to do anything with these outputs, yet. [solution-5,3]

5.4 A function that calculates phases

The key ingredient for the PM method is the calculation of pulsation phases **at fixed frequency** in different subdivisions of the light curve. Because we must keep frequency fixed, we will not be using our optimizer function any more. We need another function to calculate phases.

It is often forgotten among asteroseismologists that a Fourier transform provides not only frequencies and amplitudes, but also phases. We almost always discard the phase, because most of the time it is inconsequential. Not today. We will use function-5,4 to calculate the phase of our fixed frequencies, whenever we need it. Copy this function into your notebook and run it so it can be called later.

5.5 Write a function to calculate the time delays

This is the core activity for this Exercise. This function will need to:

- slice the light curve into subdivisions;
- store the mid-time of each subdivision;
- calculate the phase of each frequency in each subdivision; and
- convert phase shifts to time delays.

We will write this function in stages. If you think you can write such a function yourself, feel free to attempt to do so without using the steps below, but refer back here if you get lost. If you really cannot work out what to do, and the hints are not helping, feel free to jump to the solutions at each stage, but make sure you understand what is happening.

5.5.1 Setting up our function

This might be the first time you've written a function in `python`. Take a look at the `dft_phase` function to see the main ingredients of a function. It starts with `def`, has a descriptive name (what is the function going to do?), takes some variables in round brackets, and the first line then ends with a colon. Subsequent lines within the function are indented, all the way down to the `return` statement, which specifies what the function will return when called.

We already know we will need the function to receive the array of times, and our list of frequencies. Since we'll be calculating the phases of those frequencies, you can guess we'll need the magnitudes from

the time-series, too. It would be wise to allow the length of our subdivisions to be specified, and set a default value in case the length is not specified.

Eventually the function should return the mid-time of each subdivision and the measured time delay for each frequency, but we haven't got that far yet. Maybe you could return the string `"to do"` until then. Start setting up the function using the specification above. [hint-5,5,1] [solution-5,5,1]

5.5.2 Subdividing the light curve

There are many ways to 'slice' or subdivide the light curve. It makes sense to use a `for` loop, and to iterate over the data points until some critical `subdivision_size` is reached. We will need to store the times and magnitudes of each slice in an array so that we can perform operations on them. Once the critical size is reached, we will need to calculate (and store) the mid-time of the slice. We could return the array of mid-times to make sure our function is working. They'll be needed later anyway. [hint-5,5,2]

Once we have got that far, we can run the function to test it. Note that the returned mid-times might not be spaced *exactly* by `subdivision_size` because of gaps in the data, depending on how you have written your function. [solution-5,5,2]

5.5.3 Calculate phases in each subdivision

Now we will keep modifying our function to increase its utility. The next thing to do is to calculate (and store) the phase in each slice. Lucky we already have a function that returns the phase of each frequency given some time-series. Let's adapt our time-delay function to use the `dft_phase` function, and return both the mid-points and the phases now. [hint-5,5,3] [solution-5,5,3]

5.5.4 Calculate time delays

Time delays, τ, are calculable from phases, ϕ, for given oscillation frequencies, f, as

$$\tau = 86400 \frac{\phi}{2\pi f}, \tag{1}$$

where the coefficient 86400 converts from units of d to units of s (frequencies are measured in d^{-1}).

It's now not so useful to have our lists of phases in pairs, with one pair of measurements per subdivision (n columns for n subdivisions, two rows for two frequencies). It's more useful to have them separated by frequency, as two columns of length n. You know what's coming – we have to transpose the list of phases.

For each frequency and each corresponding array of phases, we now need to calculate time delays using eq. 1. Since we are only interested in the variations in time delay, we should subtract the mean time delay from each frequency. Store these time delays in an array. We should now make our function return the time delays we have calculated. [hint-5,5,4] [solution-5,5,4]

5.6 Plot the time delays

All being well, you should now have a function that returns a list of n mid-times, and a list of time delays of length n for each frequency. Now we will plot them to see how they look.

We haven't yet seen the syntax for pyplot. A simple scatter plot can be made with

```
plt.scatter(t,tau)
```

and the axes can be labelled by sending strings to `plt.xlabel()` and `plt.ylabel()`, e.g.

```
plt.ylabel("Time delay, s").
```

You will want to iterate over your two frequencies to plot each series of time delays. Pay close attention to the shape of the time delay array you receive from your function. Use the hints if you need help. Hopefully, your plot looks something like Fig. 1, though of course the actual system you have analysed

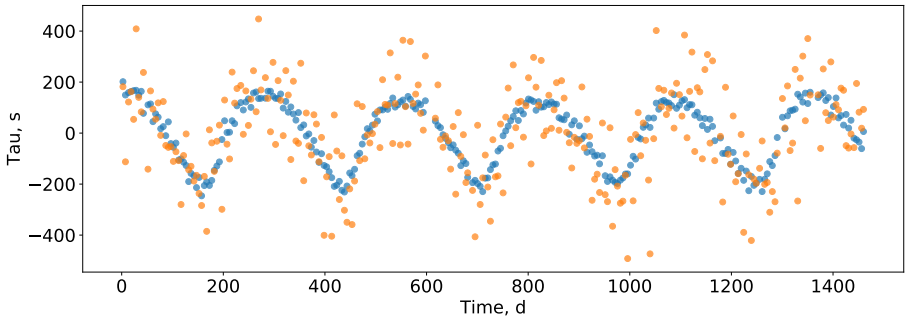

Figure 1: Time delays for the first two frequencies of the eccentric binary, KIC 9651065.

will determine the time delays measured. Consider changing the segment size and see what effect it has on your plot.[hint-5,6] [solution-5,6]

If you have analysed KIC 11754974, notice that the time delays have a linear trend. This is a calculable effect, that stems from having a non-integer number of orbital periods. Basically, the 'mean' pulsation frequencies measured are not the intrinsic pulsation frequencies of the star. Instead, they are the time-average of the (Doppler-shifted) pulsation frequencies during the observations. You can correct for it by also fitting a linear trend, or by encoding eq.15 of Murphy, Shibahashi & Bedding (2016). No need to do that now.

If you have analysed KIC 7917485 (or even generally), you might wonder why the is scatter so large. There are other (lower-amplitude) pulsation frequencies in the data set. They add variance to the data, which limits the precision of the phase determination. Also, consider the amplitude of the scatter. In the case of the planet-hosting δ Sct star, we are observing oscillation periods of 1-2 hr, and looking for changes of ± 10 s. And we're doing this on timescales of 1500 d. It's a lot to ask for!

5.7 Taking the weighted average

In all cases, the scatter can be reduced by plotting the weighted-average time delay. Fortunately, `np.average` can accept weights. We already have some weights we can use: the pulsation amplitudes. It is also fortunate that a lower amplitude mode should have a lower weight, so we don't have to do anything clever like inverting the weights. The weights are just the pulsation amplitudes. Calculate the weighted average now. [hint-5,7] [solution-5,7]

The End

That's the end! As stated in the introduction, this is a challenging exercise. It is okay if you didn't finish all of the steps. You have the data and these instructions and the hints/solutions available, so you can continue in your own time if you wish. If you finished early, or if you wish to pursue the method further, continue reading the next section.

6. Challenges / future work

Extending the analysis to other frequencies or binaries

As with FM, every pulsation frequency of the star should be similarly affected by the binary orbit. You can try to add in additional pulsation frequencies – the code you've written should scale arbitrarily. However, consider that, because of the weighting, frequencies with lower amplitudes will contribute less to the weighted average. You might find other effects due to beating, if you select frequencies with close neighbours.

You can quite quickly analyse another binary. The only things specifically hard-coded to the first binary you analysed should be the list of pulsation frequencies and the light curve file they correspond to. Try analysing a different binary. You have a few to choose from. You could duplicate your notebook so that you also keep the existing work.

Obtaining orbital solutions

In this Exercise we didn't yet extract orbital solutions. The equations for doing this are relatively complicated (see, e.g. Murphy, Shibahashi & Bedding 2016). There is a package, Maelstrom, currently in development by Daniel Hey (USyd) for doing just this. It is covered in the notebook subdividing.ipynb. You might like to take a look through it, even if you don't run everything.

7. Things to note if you use this for research

This tutorial was based on the Maelstrom software, which in turn is based on the PM method papers (all of them). The analysis of the planet host system was written up in a separate ApJ Letter. Please cite them if you use the PM method in your work. It would be wise to read and understand them.

The future development work in the methodology will be via Maelstrom. The first paper on that will be published soon. Watch this space! You can already play with the code though. See the Getting Started page for Maelstrom.