The following people have participated in creating these solutions: Nicolaas E. Groeneboom, Magnus Pedersen Lohne, Karl R. Leikanger
NOTE: There might be errors in the solution. If you find something which doens't look right, please let me know

Partial solutions to problems: Part 1C

Problem 1C.1

1. $v_r \approx 12$ m/s, yes!
2. $v_r \approx 0.1$ m/s, no!
3. Using formulae in the text, you should find that $v_r \propto 1/\sqrt{a}$ such that the radial velocity decreases for larger distances a.
4. 1.2×10^6 km.

Problem 1C.2

1. We assume that the orbits of all planets are circular. In the previous exercise, we found the speed of the sun's wobbling due to Jupiter was given as $v_* = 12.20$m/s. The radius a_* the sun wobbles is given from

$$v_* = \frac{2\pi a_*}{P}$$

such that

$$a_* = \frac{v_* P}{2\pi}$$

What we are interested in is the maximum distance from us to the star, and this is given by $\theta = (a_*/d)$. **Note:** When using this approximation ($\tan\theta \sim \theta$ for small values of θ), it is important to perform the actual computations using *radians* and not *degrees*. Now,

$$d = 2 * \frac{a_*}{\theta} = \frac{v_* P}{\pi\theta}$$

where $\theta = 0.1'' = \frac{1}{60}\frac{1}{60}\frac{2\pi}{360} = 4.848 \cdot 10^{-7}$ radians. The extra factor 2 is added, as the star wobbles with a magnitude a_* in *two* directions, doubling the effect.

When inserting the values for Jupiter, one obtains

$$d \approx 3.04 \cdot 10^{15} m = 0.32\text{ly}$$

which is the *distance from us* the star needs to be in order to detect any wobbling. Note that the distance to Sun's closest neighbour Proxima Centauri is 4.22 ly.

2. $d \approx 0.0002$ly.

3. First of all, wikipedia states that *Proxima Centauri* contains 0.123 solar masses. We use a formula presented in chapter 2, giving a relation between different orbit sizes. We have:

$$a_1 = \frac{\mu a}{m_1}$$

where $a_1 = a_*$ is the radius of the orbit of the wobbling star about the center of mass, $m_1 = m_*$, $a \equiv a_d$ is the distance between the star and the planet while the reduced mass $\hat{\mu} \sim m_p$ can be approximated to be the mass of the orbiting planet. Thus,

$$m_p = \frac{m_1 a_1}{a} = \frac{m_* a_*}{a_d}$$

We need to decide on a_*. Now, in order to detect wobbling, $\theta = 0.1''$, and we know $d = 4.22ly$ such that a_* must be

$$2a_* = d\theta$$

inserting this into 3, we find

$$m_p = \frac{m_* d\theta}{2a_d} = \frac{0.123 \cdot \text{Msun} \cdot 4.22\text{ly} \cdot \frac{0.1 \cdot 2\pi}{60 \cdot 60 \cdot 360}}{1\text{AU} \cdot 2} \approx 8\,\text{Jupiter masses}$$

4. $1.6 m_{\text{jupiter}}$

Problem 1C.3

1. Answer found in the text.
2. 2300 m/s.
3. $v_{*r} \approx 170$ m/s, $P \approx 285$ hours.
4. $m_p \approx 1.7 m_{\text{jupiter}}$.
5. Answer found in the text./
6. $\Delta t \approx (10.5 - 10.1) \times 3600 \approx 1440$ seconds, $v_p \approx 86$ km/s, $r_p \approx 62 \times 10^3$ km, $\rho \approx 3100 kg/m^3$
7. This density is just between the density of a gas planet and a rocky planet. Note that the uncertainties in our estimate of the radius is big, with the noise it is hard to judge when the eclipse starts and when it becomes full. The density depends on this radius to the 3rd power which makes it very sensitive to small errors in the radius. Depending on whether we have under- or overestimated the radius, this could be a gas planet or a rocky planet. (**Note:** you might have read off slightly different numbers from the figure and thereby obtained a slightly different radius which might give you a more conclusive density. As long as the radius is not *very* different from the one we obtained here your answer is probably also correct within the uncertainties.

Problem 1C.4

Here is an outline of a Python code which can be used:

```
from scitools.all import *


#STAR CLASS
class star:

   #Constructor
   def __init__(self, filename):
      #Declare self-variables/lists
      self.filename = filename
      self.time = []
      self.flux = []
      self.lambda_obs = []
      ...
      ...

      #Calculations/call functions
      self.read_file()
      self.rad_velocity()
      self.pec_velocity()
      self.light_curve()
      self.vel_curve()
      self.model()


   #Function that reads data from the file self.filename
   def read_file(self):
      ...
      self.time.append(...)         #Store data in self-lists
      self.flux.append(...)
      self.lambda1.append(...)
      ...
      #Make arrays
      ...


   #Function that calculates the radial velocity for each observed wavelength
   def rad_velocity(self):
      self.rad_vel = ...      #Use Doppler's formula


   #Function that calculates the peculiar velocity
   def pec_velocity(self):
      self.pec_vel = sum(..)/len(..)  #sum(array) - sum all elements in an array
                                      #len(array) - length of array
      #Write to file
```

```
   ...


#Function that plots the light-curve
def light_curve(self):
   #Plot flux vs time
   ...


#Function that plots the velocity-curve
def vel_curve(self):
   #Relative velocity - Deviation of the radial velocity from the peculiar
   #(average) velocity
   self.rel_vel = self.rad_vel - self.pec_vel

   #Plot relative velocity vs time
   ...


#Function that evaluates the best values of the constants v_r, P and t_0
#in the cosine-model, based on the least-square method
def model(self):
   #Ask in screen
   planet = float(raw_input('Does %s have a planet eclipsing? (yes/no)' % (self.name)))

   if planet == 'yes':
      #Read in min and max possible value of t0, vr and P from screen
      t0_min = float(raw_input(' ... '))
      t0_max = ...
      vr_min = ...
      ...

      #Declare t0, vr and P - arrays
      t0 = linspace(t0_min, t0_max, 'number of evaluation-points per constant')
      vr = ...
      P = ...

      #Declare best-variables (set equal the first value in their arrays)
      best_t0 = ...
      best_vr = ...
      best_P = ...

      #Go through every combinations of the constants t0, vr and P, one
      #for-loop for each constant-array
      for i in range(len(t0)):
         for j in (...):
            for k in (...):
               #Calculate the array of model-velocity, use t0[i], vr[j] and P[k]
               rel_vel_model = ...
```

```
                #Calculate the sum of the difference between all the elements
                #in the self.rel_vel-array and the rel_vel_model-array squared.
                delta = sum((self.rel_vel - rel_vel_model)**2)

                #Check if you have a better constant-set (t0[i], vr[j] and P[k])
                #than the previously stored best values
                if delta < best_delta:
                ...
                ...


        #Plot the result, use hold('on') to plot two graphs in the same
        #figure, remember to use hold('off') afterwords
        ...

        #Write the best constant-values to file - write all star-values in 1 file
        if self.name = 'star0':
           file = open('filename', 'w')
        else:
           file = open('filename', 'a')
        ...                              #Write to file
        file.close()


#____
#MAIN

#Make star-objects
star0 = star('star0.txt')
star1 = star('star1.txt')
star2 = star('star2.txt')
star3 = star('star3.txt')
star4 = star('star4.txt')
```

The best program approach is to make a star-class. In this way we get a compact and well-presented program where we don't have too many variables to think of. For each star object that we make (in the beginning of main), with argument 'filename', the constructor function automatically runs. Since all the functions are called from the constructor, all the calculations are carried out by just making the object. If we were to do more calculations in main, and especially if we needed variables inside the star objects, the gain in the class approach would be even clearer. But even in this problem it should be fairly obvious what we gain by the class approach.