Observational Astronomy: Exoplanet Transit Parameter Retrieval using Markov chain Monte Carlo (MCMC)

Miguel Zammit

November 9, 2022

Introduction

In this project you shall be introduced to several techniques widely used in modern observational astronomy. More specifically, you shall be learning how to apply Markov chain Monte Carlo (MCMC) sampling to model exoplanet transits, so as to then retrieve orbital and physical parameters of the planets in question.

Exoplanets, or extra-solar planets, are planetary bodies beyond the solar system, orbiting stars other than our Sun. As these systems are so far away that we can only see their host stars as point light sources in the night sky, the detection of exoplanets tends to depend on observing variations in the incoming starlight. One such method is known as the *transit method*. When a planet's orbit is aligned in such a way that it crosses between the observer and its host star, such that it transits across the stellar disk, it will periodically block a fraction of the incoming starlight. The depth of this 'dip' in brightness, as well as the duration and shape of the transit, allows for the retrieval of several orbital and physical parameters.

In this project you will be using MCMC to fit a transit model to observational data of a previously

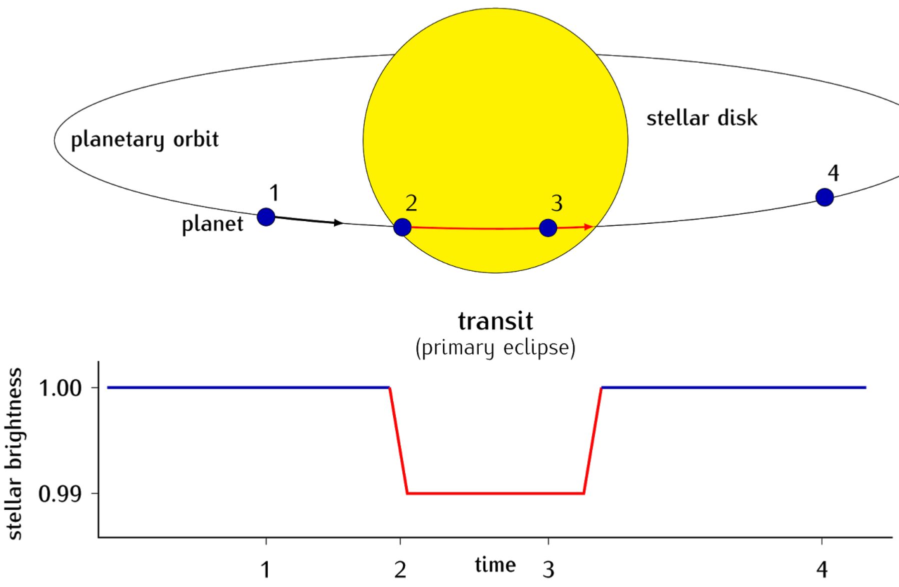

Figure 1: The transit method. Figure from [1].

discovered exoplanet. You will be using two python packages: the `emcee`[1] [2] package for implementing an affine-invariant ensemble sampler for Markov chain Monte Carlo, and the transit model calculation package `batman`[2] [3]. The first part of this project will help you familiarise yourself with each of these packages separately, so that you can then use that familiarity with each method's functionality to combine both to fit the model to the observational data. With the transit and physical parameters from your best fit, you will then be asked to qualitatively interpret the nature of the exoplanet being observed.

It is recommended that prior to starting the practical, you consult the suggested text to get a basic, introductory understanding of the transit method & all the orbital parameters mentioned below, as well as a brief qualitative overview of MCMC.

Task 1A: Calculation of exoplanet transit light curves

For the first task, you will be getting acquainted with the `batman` python package to generate light curves for a number of exoplanet configurations.

1. In the project folder you will find a .csv file named `kepler_90_pl.csv` with the orbital and physical parameters of the Kepler-90 planets. Before you start, make sure you have found a value for the stellar radius of Kepler-90. Open the file as a DataFrame using the planet names as your index.

2. Before we can generate the lightcurves, there's a few things we need to set. First, the values in the DataFrame need to be used to obtain the transit parameters. Particularly, we need to calculate: the ratio of planetary radius to stellar radius $\frac{R_\mathrm{P}}{R_\star}$; and the semimajor axis in units of stellar radii $\frac{a}{R_\star}$. Add these as separate columns to your DataFrame.

3. Now we can start working on our transit models. First, we need to create an object to store the transit parameters for the model as follows.

```
parameters = batman.TransitParams()
```

4. For simplification some of the transit parameters can be assumed equal for all planets. Namely: the time of inferior conjunction t_0 and the argument of periapsis w in degrees.

```
parameters.t0 = 0.          # t_0
parameters.w = 0.           # w (in degrees)
```

5. The next step will involve us defining the mathematical profile of what is known as limb darkening. Essentially, the intensity of the light across the stellar disk is not constant, with the central area appearing brighter than the outer region (limbs) of the disk. Hence, as the planet transits across the disk, the intensity of light blocked will not be constant and will affect the shape of the light curve. The limb darkening intensity profile selected will therefore model the shape of our light curve model accordingly. To select the limb darkening profile to implement in our model, we need to set a string value to

```
parameters.limb_dark = ' '      # limb darkening model
```

In particular we will be using 3 different stellar intensity profiles:

For a normalised radial coordinate $\mu = \sqrt{1 - x^2}$, $0 \leq x \leq 1$ and stellar intensity normalisation constant I_0,

Uniform	$I(\mu) = I_0$
Linear	$I(\mu) = I_0\left(1 - c_1\left[1 - \mu\right]\right)$
Quadratic	$I(\mu) = I_0\left(1 - c_1\left[1 - \mu\right] - c_2\left[1 - \mu\right]^2\right)$

[1] https://emcee.readthedocs.io/en/stable/

[2] https://lweb.cfa.harvard.edu/~lkreidberg/batman/index.html

Once you specify the model to implement, you need to provide the limb darkening coefficients for the model, in the form of a list.

```
parameters.u = []               # Uniform limb darkening model
parameters.u = [c_1]            # Linear limb darkening model
parameters.u = [c_1, c_2]       # Quadratic limb darkening model
```

6. Generate a `numpy` array of 1,000 datapoints for the time from the central transit, centred at time 0. Next, using a uniform limb darkening model, generate & plot light curves for all the Kepler-11 planets. Discuss the shape and depth of the curves, and consult the dataset to explore how they affect the main observables of the transit, namely the transit depth, duration and contact points. To set the parameters for each planet, you need to set the values for each light curve as follows:

```
parameters.rp =                 # R_p/R_stellar
parameters.a =                  # a/R_stellar
parameters.inc =                # orbital inclination
parameters.ecc =                # orbital eccentricity
parameters.per =                # orbital period
```

   Finally, initialise the model and calculate the light curves.

```
m = batman.TransitModel(parameters, # ENTER TIME ARRAY)
rel_flux = m.light_curve(parameters)
```

7. Select a planet and generate light curves for a linear limb darkening profile with c_1 set to several values between 0 and 1. Comment on the changes occurring in the shape of the light curve. Repeat for a quadratic limb darkening profile, with several values of $0 \leq c_1 < 1$ & $0 \leq c_2 < 1$.

Task 1B: Introduction to Markov Chain Monte Carlo (MCMC)

Next, you will use the `emcee` package to fit a curve to a dataset using MCMC. The steps below provide a step-by-step guide to building an MCMC algorithm.

1. Open the .csv file named `mcmc_intro_group_5.csv` as a `pandas` DataFrame. The columns should be the x-axis, y-axis and Δy respectively. As is immediately evident from plotting the data, the true model for the dataset is a sinusoidal curve.

Using MCMC, we will attempt to fit a general form of $y(x) = a + b\sin(cx + d)$ to the data and try to approximate the coefficients of the true model. In this particular case, we have 4 parameters to fit. So let's consider a θ vector which will contain values of those 4 parameters which the walkers in our MCMC sampler will explore within the prior ranges we will set later on. For a certain θ_i,

$$\theta_i = \begin{pmatrix} a_i \\ b_i \\ c_i \\ d_i \end{pmatrix}$$

Each walker will try new values of θ, attempting to find the optimal set that maximises the likelihood and provides the best fit to the dataset.

Before running the algorithm, we need to define 4 functions to be used in the MCMC run: a *model* function, a *log-likelihood* function, a *prior* checking function and a *probability* function.

2. The model function will simply be the general function we are trying to fit the dataset to. Its input parameters should therefore be θ (i.e. a list of all the coefficients in the model) and our x-axis.

```
def model(theta, x=x):
    a, b, c, d = theta

    model = # ENTER MODEL FUNCTION HERE
    return model
```

3. Now that we have generated our model using the current value of θ, the next step is to numerically calculate a value for our likeliness of fit to the data. For this case, we will use a χ^2-type check, adding a negative sign so as to maximise our likelihood as the optimiser minimises this value:

$$\ln \mathscr{L}(\theta | x_1, x_2, \ldots, x_n) = -\frac{1}{2} \sum_i^n \left(\frac{y_{\mathrm{obs}_i} - y_{\mathrm{model}_i}}{y_{\mathrm{err}_{\mathrm{obs},i}}} \right)^2$$

   Translating this to code,

```
def ln_like (theta, x, y, y_err):
    ln_like = # ENTER FUNCTION HERE
    return ln_like
```

   If we consider a single walker, calculating this likelihood will allow it to start moving across the parameter space to regions where it thinks the likeness of fit is highest, such that eventually the walkers will converge onto a θ that approximates the true model parameters and achieves a global maximum.

4. The next function to define is the prior checking function. Essentially, we set our priors here, then call this function at every step to check if all arguments are within their respective priors. If they are satisfied, 0.0 is returned by the function. If not, $-\infty$ is returned.

```
def ln_prior(theta):
    a, b, c, d = theta

    if # ENTER PRIOR RANGES FOR EACH COMPONENT OF THETA:
        return 0.
    else:
        return -np.inf
```

   The range we set for our priors at this stage will drastically effect whether or not the MCMC will converge. Certain parameter values can be immediately guessed from the data plot, while others can tend to be more challenging to constrain. Use your plot to estimate priors for a and b, and use the following for c & d:

$$20 < c < 25$$
$$6 < d < 11$$

5. The fourth function to define is the probability function. This will call the prior function and check if it returns $-\infty$, to which it then itself returns $-\infty$, or 0., in which case the function returns the likelihood function for that θ. By convention, this latter step is written as the output of the prior function + the likelihood function's output, which in this case would simply be 0. + `ln_like`. So our probability function will be:

```
def ln_prob(theta, x, y, y_err):
    lp = ln_prior(theta)
    if not np.isfinite(lp):
        return -np.inf
    return lp + ln_like(theta, x, y, y_err)
```

With the 4 functions defined, there are a few final lines of code to add to initialise the MCMC run.

6. Include the following code to your script. We first define a tuple with our data. Set the number of walkers to 500 and the number of iterations to 50,000. Next, we need to set an array of initial set of values for the parameters. Enter initial guesses based on the priors we chose earlier. Finally, we have a function to calculate p_0, the step methodology across the parameter space from one place of the grid to the next.

```
# data input has to be set as a tuple as follows
# (x_data, y_data, y_data_err)
data = (x, y, y_err)

n_walkers = #number of walkers for the MCMC
n_iter = #number of iterations for the MCMC

initial = np.array([#INITIAL a, #INITIAL b,
                    #INITIAL c, #INITIAL d])
n_dim = len(initial)

def _p0(initial, n_dim, n_walkers):
    return [np.array(initial) + [1e-4, 75e-3, 1e-2, 1e-2]
      * np.random.randn(n_dim) for i in range(n_walkers)]

p0 = _p0(initial, n_dim, n_walkers)
```

7. Finally, we can define our MCMC function as follows:

```
def mcmc(p0, n_walkers, n_iter, n_dim, ln_prob, data):
    sampler = emcee.EnsembleSampler(n_walkers, n_dim,
                                    ln_prob, args=data)

    print("Running burn-in...")
    p0, _, _ = sampler.run_mcmc(p0, 2000, progress=True)
    sampler.reset()

    print("Running production...")
    posteriors, prob, state = sampler.run_mcmc(p0, n_iter,
                                               progress=True)

    return sampler, posteriors, prob, state
```

8. Run the mcmc. This might take some time.

```
sampler, posteriors, prob, state = mcmc(p0, n_walkers,
                                        n_iter, n_dim,
                                        ln_prob, data)
```

9. The best fitting parameter values will be informed by the posterior distributions. Flatchain flattens our samples such that we can then plot a corner plot with all the fit parameters.

```
samples = sampler.flatchain
```

Generate corner plots using the following, and set values for the quantiles to get some elementary uncertainties:

```
import corner

labels = ['a', 'b', 'c', 'd']
fig = corner.corner(samples,show_titles=True,
                    labels=labels, plot_datapoints=True,
                    quantiles=[# LOWER LIMIT, 0.5, # UPPER LIMIT])
```

Task 2: Fitting a transit light curve to observational data

The final task is to essentially combine the procedure you have learned in the previous sections to fit a transit curve to an observational dataset of Kepler-1b.

Parameter	Value
argument of periapsis w	0.
eccentricity e	0.
limb darkening model	quadratic
limb darkening coeff	[0.5,0.01]
walkers	200
iterations	100,000
p_0 randomisation	[.1, 0.2, 0.01, 0.3, 5.]

1. Kepler-1b is a hot Jupiter reported to transit approximately every 2.5 days, as it orbits around its stellar host Kepler-1 ($M_\star = 0.9800 \pm 0.0620 M_\odot$, $R_\star = 1.003 \pm 0.027 R_\odot$) [4]. The csv file titled `kepler_lc_group_4.csv` contains the photometric data from an observation by [4]. Your job now is to fit the `batman` model to the data and retrieve 5 parameter estimates from the best fit: t_0, the orbital period, planetary radius, semi-major axis and orbital inclination. Below you will find the values to assume for the rest of the `batman` parameters and MCMC run. The MCMC metrics are not set in stone and can be viewed as more of a suggested starting point.

2. From the light curve data and your fit, what initial expectations can you have regarding specific observables? What priors can you set so far?

3. For the priors of $\frac{a}{R_\star}$, look into Kepler's Laws on how to find an initial rough estimate from the initial guess of the orbital period. The range for the priors will have great impact on whether or not the MCMC run succesfully finds the optimal θ. For i use a range from $60° - 90°$, with an initial guess of $90°$.

4. Run the MCMC sampler and obtain values for the 5 parameters from your best fit. It should be expected that the model might not immediately fit the data well. Keep a personal log of all amendments you attempted on the priors and initial estimates, so as to note how certain changes affect convergence.

5. Once you manage to obtain an accurate fit, consult the results from [4] (can also be found on the NASA Exoplanet Archive) and comment on any discrepancies in your values.

6. Discuss the nature of the planet based on your results and some light research into hot Jupiters and Kepler-1b in particular. What makes hot Jupiters particularly suited to the transit method and transit spectroscopy?

7. Look into ways on how you can sample walkers from your MCMC run to plot a 1σ posterior spread with the highest likelihood model.

Further Reading

The Transit Method:
https://www.paulanthonywilson.com/exoplanets/exoplanet-detection-techniques/the-exoplanet-transit-method/
MCMC :
https://machinelearningmastery.com/markov-chain-monte-carlo-for-probability/
MCMC (in-depth):
https://events.mpifr-bonn.mpg.de/indico/event/30/material/slides/12.pdf

References

[1] Exoplanet Diagrams: The Transit Method; 2015. Available from: http://exoplanet-diagrams.blogspot.com/2015/07/the-transit-method.html.

[2] Foreman-Mackey D, Hogg DW, Lang D, Goodman J. emcee: the MCMC hammer. Publications of the Astronomical Society of the Pacific. 2013;125(925):306.

[3] Kreidberg L. Batman: basic transit model calculation in Python. Publications of the Astronomical Society of the Pacific. 2015;127(957):1161.

[4] Holman MJ, Winn JN, Latham DW, O'donovan FT, Charbonneau D, Torres G, et al. The transit light curve (TLC) project. VI. Three transits of the exoplanet TrES-2. The Astrophysical Journal. 2007;664(2):1185.