In [ ]:
```
import numpy as np
from MCMC_aux import get_model
import Parameters as par
import Models as mod
import Kernels as ker
import GP_Likelihood as gp
from MCMC import run_MCMC as run
import plotting as plot
from saving import save
import auxiliary as aux
```

```
2813.2308006313287 16620.585819951895
```

Generating Fake Data

Much like tutorial 1 and 2, we start by generating some fake cosine data to represent the activity. This time we simulate using multiple data sets possibly from multiple telescopes and therefore potentially with different offsets. We do this by defining 3 different time, rv and rv error arrays that all plot the same cosine curve but each with their own small noise term. Here, time1 and rv1 is acting as our baseline data that we want to calibrate the offsets of the other data to so we don't give the fake data an offset but we give a +50 offset to rv2 and a +20 offset to rv3.

In [ ]:
```
# set up the 3 different time arrays
time1 = np.arange(0., 40., 4.)
time2 = np.arange(0.5, 40.5, 5.)
time3 = np.arange(1.3, 41.3, 8.)
# set up cosine amplitude and period
A = 10.
P = 10.
# set up noise term
err = []
for i in time1:
    err.append(np.random.uniform(-3,3))
# rv1 has no offset
rv1 = A*np.cos(time1*((2*np.pi)/P))+err
rv1_err = np.ones_like(rv1)*4

err = []
for i in time2:
    err.append(np.random.uniform(-3,3))
# rv2 has a +50 offset
rv2 = A*np.cos(time2*((2*np.pi)/P))+err+50
rv2_err = np.ones_like(rv2)*5

err = []
for i in time3:
    err.append(np.random.uniform(-3,3))
# rv3 has a +20 offset
rv3 = A*np.cos(time3*((2*np.pi)/P))+err+20
rv3_err = np.ones_like(rv3)*3
```

The combine_data function turns these 3 data sets into 1 large dataset. It orders the data in order of times and assigns flags to each point which are returned in the flags array. These flags indicate which data set the point was originally from. The order of the data inputted into this function is very important as the first data, in this case time1 and rv1, always relates to the data indended to be used as the baseline. This means when applying offsets further into the code, these offsets will only apply to the data not in the first position, effectively calibrating it to the first data set.

In [ ]:
```
# this can take any number of datasets however the current plotting code can
time, rv, rv_err, flags = mod.combine_data([time1,time2,time3], [rv1,rv2,rv3
```

Whilst it is not neccesary anywhere in the following code, it is possible to obtain the offset subtracted rv data by running the offset_subtract function along with the combined rvs, the flags, and the offsets. The order of this offsets should follow the order of the data inputed into the combine_data function, in this case 50 will be subtracted from rv2 and 20 will be subtracted from rv3.

In [ ]:
```
# obtain offset rvs, this is not needed for the code and should not be input
offset_rv = plot.offset_subtract(rv, flags, [50,20])
```

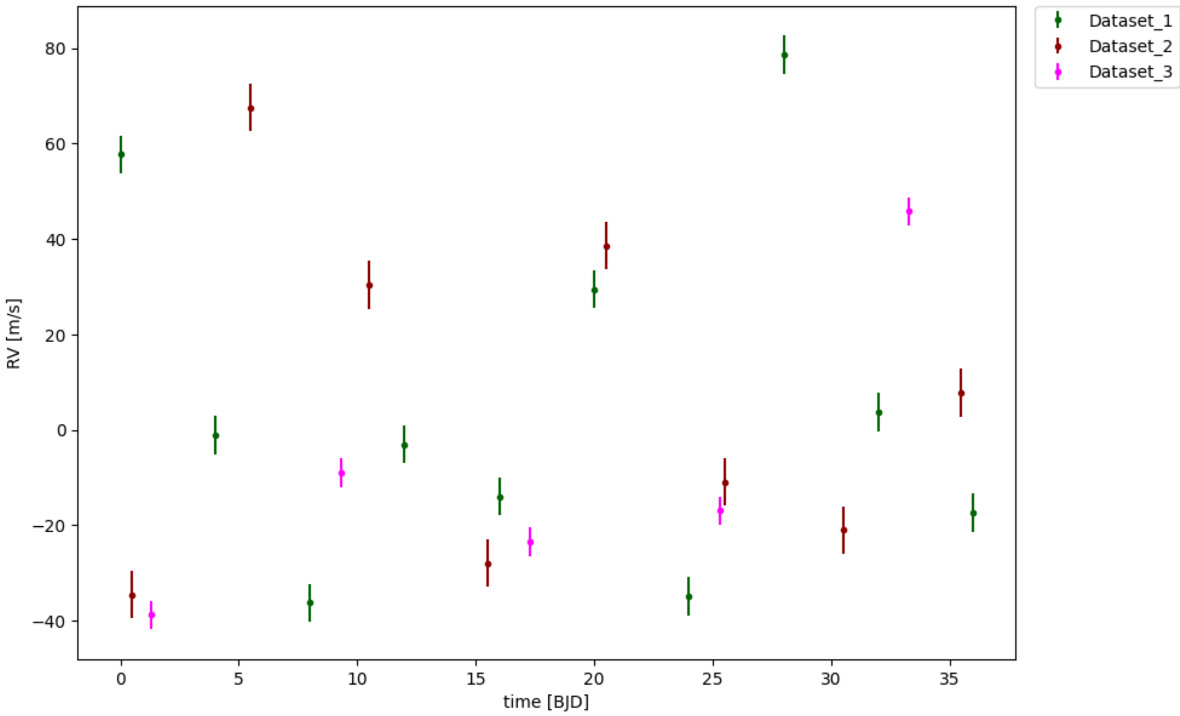
Similarly to creating the polynomial in tutorial 2, we now create 2 Keplerians to inject into the data

In [ ]:
```
# function allows for generations of keplerians
def ecc_anomaly(M, ecc, max_itr=200):
    '''
    ----------
    M : float
        Mean anomaly
    ecc : float
        Eccentricity, number between 0. and 0.99
    max_itr : integer, optional
        Number of maximum iteration in E computation. The default is 200.
    Returns
    -------
    E : float
        Eccentric anomaly
    '''

    E0 = M
    E = M
    #print("E before = ", E)
    for i in range(max_itr):
        f = E0 - ecc*np.sin(E0) - M
        fp = 1. - ecc*np.cos(E0)
        E = E0 - f/fp

        # check for convergence
        if (np.linalg.norm(E - E0, ord=1) <= 1.0e-10):
            return E
            break
        # if not convergence continue
        E0 = E

    # no convergence, return best estimate
    #print('Best estimate E = ',E[0:5])
    return E

# keplerian 1 parameters
K = 30
ecc = 0.5
omega = np.pi/2.
Pl = 7.2
```

```
t0 = time[0]

M = 2*np.pi * (time-t0) / Pl
E = ecc_anomaly(M, ecc)
nu = 2. * np.arctan(np.sqrt((1.+ecc)/(1.-ecc)) * np.tan(E/2.))

# generate keplerian 1 and add it to the data
Kep = K * (np.cos(omega + nu) + ecc*np.cos(omega))
rv = rv + Kep

# keplerian 2 parameters
K = 38
ecc = 0.7
omega = np.pi/4
Pl = 5.6
t0 = time[0]

M = 2*np.pi * (time-t0) / Pl
E = ecc_anomaly(M, ecc)
nu = 2. * np.arctan(np.sqrt((1.+ecc)/(1.-ecc)) * np.tan(E/2.))

# gernerate keplerian 2 and add it to the data
Kep2 = K * (np.cos(omega + nu) + ecc*np.cos(omega))
rv = rv + Kep2
```

The data_plot function works mostly the same as in the past except now it requires a flags input and an offsets input. The offset input works the same as the offset input to offset_subtract where the order should be the same as the order of the data in combine_data. In this case 50 will be subtracted from rv2 and 20 will be subtracted form rv3.

In [ ]:
```
# view the combined data with any offsets subtracted
plot.data_plot(time, rv, y_err = rv_err, flags = flags, offsets = [50,20])
```



Creating the Kernel and the Models

The kernel and the model are created in the exact same way using the same functions as the previous tutorials, the only difference this time being we have multiple models. A Cosine kernel is used along with 2 Keplerian models and 2 Offset models. Importantly, the order of the offset models is the same order as the data was inputted inot combine_data so in this case offset_0 correlates to the offset of rv2 and offset_1 correlates to the offset of rv3.

We will once again skip over the GPLikeihood funcitons as these are not neccessary for the code to run but the steps to obtain initial LogL and GP y values are identical to tutorial 2.

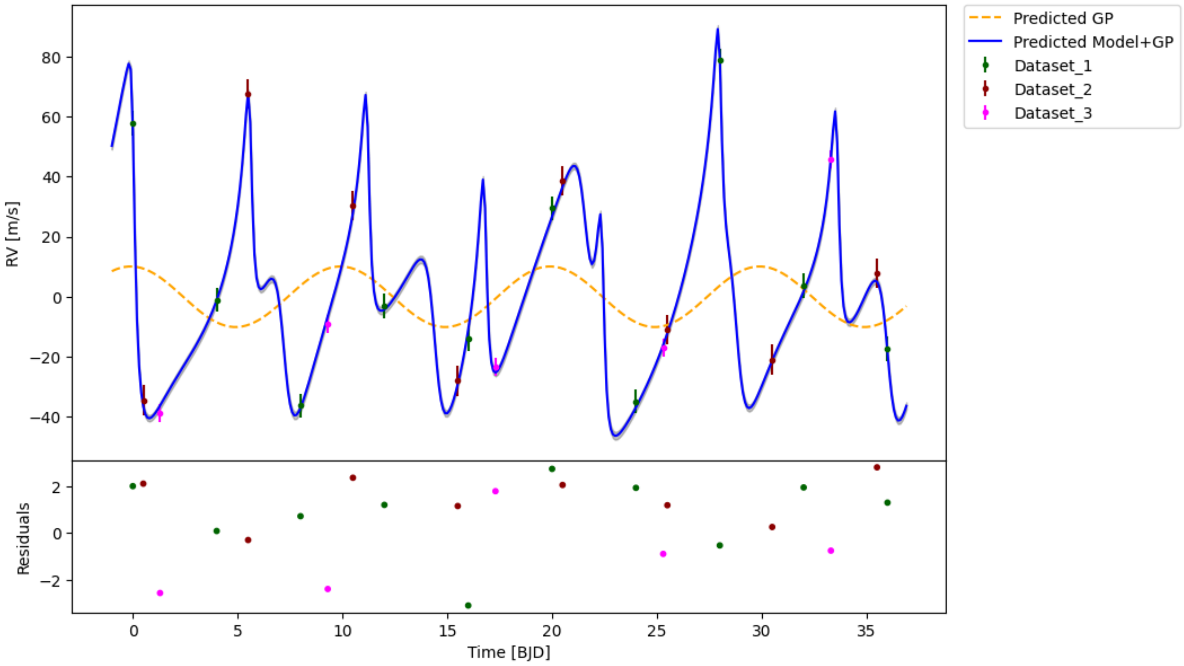
In [ ]:
```
# set up the cosine kernel
hparam = par.par_create("Cosine")
hparam["gp_amp"] = par.parameter(value = 10., error = 0.5, vary = True)
hparam["gp_per"] = par.parameter(value = 10., error = 0.5, vary = True)
#set up the kernel priors
prior_list = []
pri_amp = par.pri_create("gp_amp", "Uniform", [0.,20.])
prior_list.append(pri_amp)
pri_per = par.pri_create("gp_per", "Uniform", [0.,20.])
prior_list.append(pri_per)
# the model list contians 2 keplerians and 2 offsets, the order is unimporta
model_list = ["Keplerian", "Offset", "Offset", "Keplerian"]
model_par = mod.mod_create(model_list)
# set up the model parameters
model_par["P_0"]=par.parameter(value=7.2, error=0.5, vary=True)
model_par["K_0"]=par.parameter(value=30., error=1., vary=True)
model_par["ecc_0"]=par.parameter(value=0.5, error=0.1, vary=True)
model_par["omega_0"]=par.parameter(value=np.pi/2, error=0.05, vary=True)
model_par["t0_0"]=par.parameter(value=0., error=0.1, vary=True)
model_par["P_1"]=par.parameter(value=5.6, error=0.5, vary=True)
model_par["K_1"]=par.parameter(value=38., error=1., vary=True)
model_par["ecc_1"]=par.parameter(value=0.7, error=0.1, vary=True)
model_par["omega_1"]=par.parameter(value=np.pi/4, error=0.05, vary=True)
model_par["t0_1"]=par.parameter(value=0., error=10., vary=True)
# offset_0 refers to the offset of rv2
model_par["offset_0"] = par.parameter(50., 0.5, True)
# offset_1 refers to the offset of rv3
model_par["offset_1"] = par.parameter(20., 0.5, True)
# set up model priors
pri_val = par.pri_create("P_0", "Uniform", [0.,10.])
prior_list.append(pri_val)
pri_val = par.pri_create("K_0", "Uniform", [0.,60.])
prior_list.append(pri_val)
pri_val = par.pri_create("ecc_0", "Uniform", [0.,1.])
prior_list.append(pri_val)
pri_val = par.pri_create("omega_0", "Uniform", [0.,5.])
prior_list.append(pri_val)
pri_val = par.pri_create("t0_0", "Uniform", [0.,5.])
prior_list.append(pri_val)
pri_val = par.pri_create("P_1", "Uniform", [0.,10.])
prior_list.append(pri_val)
pri_val = par.pri_create("K_1", "Uniform", [0.,60.])
prior_list.append(pri_val)
pri_val = par.pri_create("ecc_1", "Uniform", [0.,1.])
prior_list.append(pri_val)
pri_val = par.pri_create("omega_1", "Uniform", [0.,5.])
prior_list.append(pri_val)
pri_val = par.pri_create("t0_1", "Uniform", [0.,5.])
prior_list.append(pri_val)
```

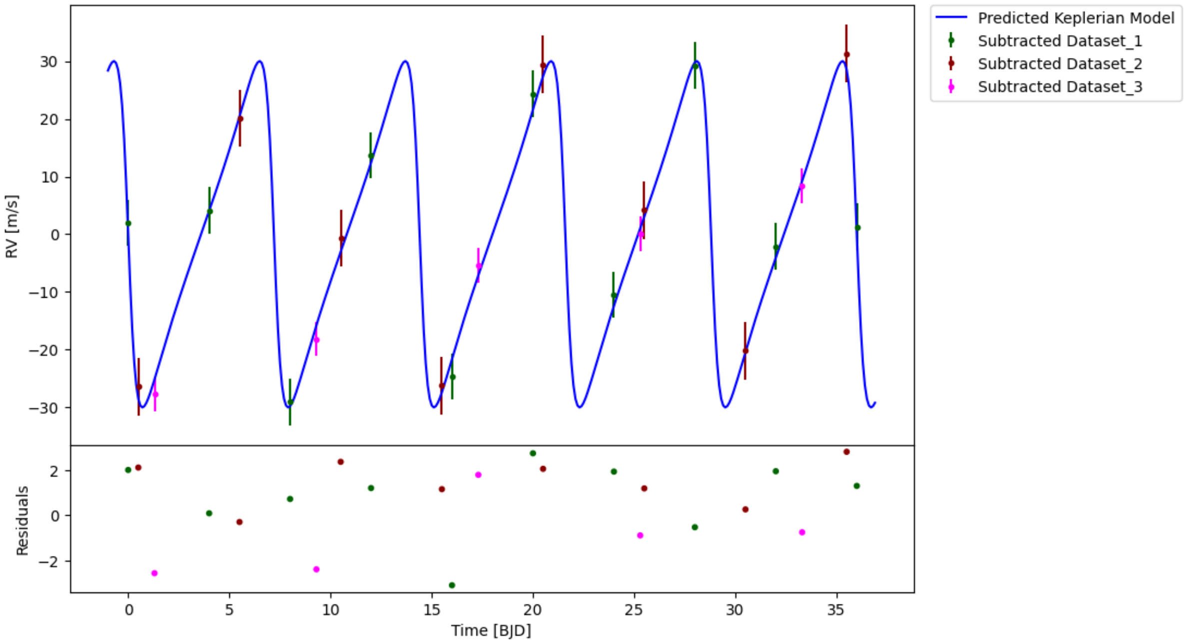
```
pri_val = par.pri_create("offset_0", "Uniform", [40.,60.])
prior_list.append(pri_val)
pri_val = par.pri_create("offset_1", "Uniform", [0.,30.])
prior_list.append(pri_val)
```

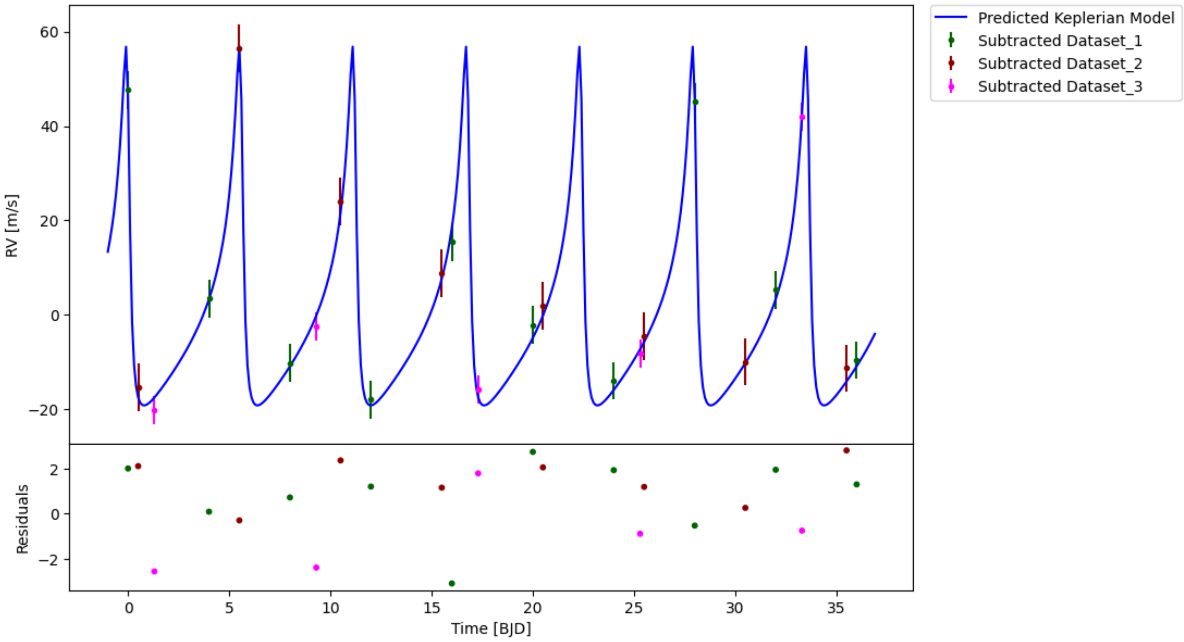
Plotting Initial Conditions

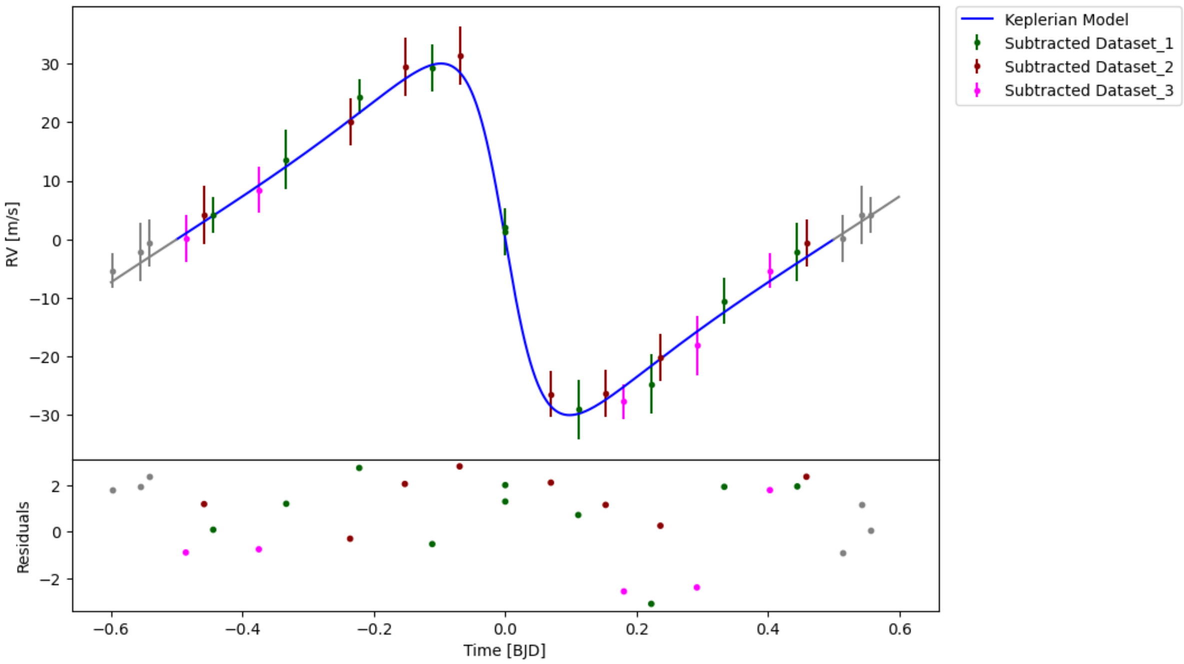
The same plotting functions as tutorial 3 are used to plot the initial parameters, this can also be done at the end similarly to tutorial 3 but in this case the data is fake and we know the exact values so these plots should fit almost perfectly. We are able to choose which kerplerian we plot by changing the keplerian_number in the kerplerian only plots and the phase plots. These functions all now require the extra input of flags to plot the different data and adjust for the offsets properly.

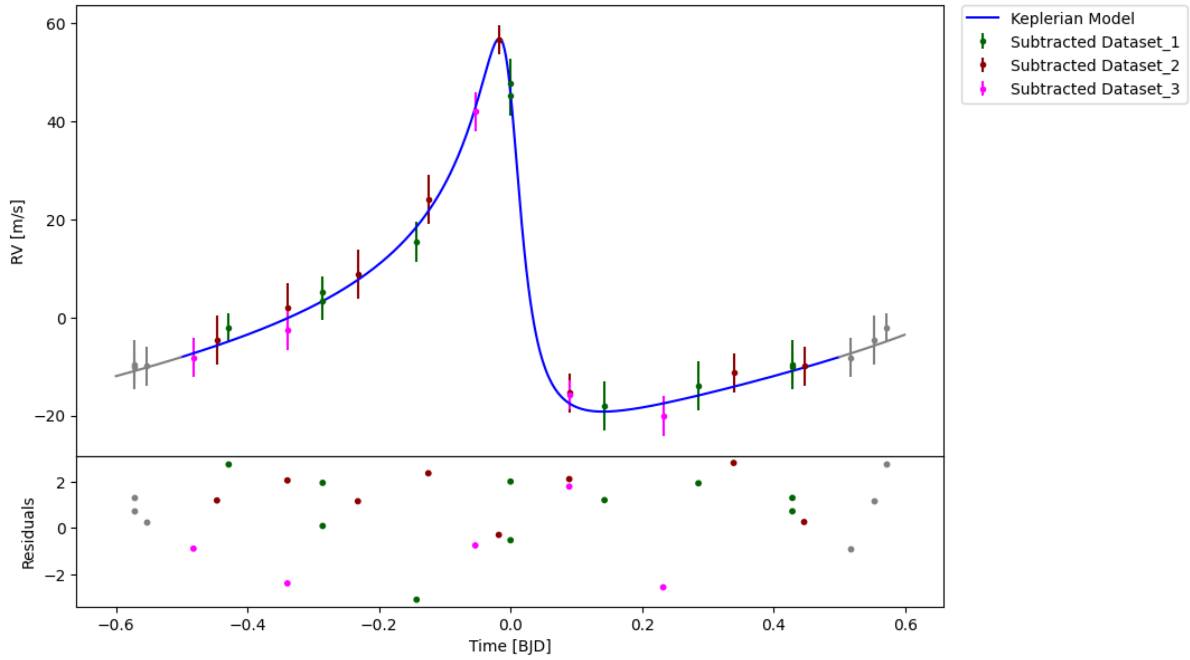
In [ ]:
```
# plot of the offset subtracted data along with the GP in orange and all moc
plot.GP_plot(time, rv, hparam, "Cosine", rv_err = rv_err, model_list = model
# kepplerian only plot relating to the kerplerian with parameters P_0, K_0, e
plot.keplerian_only_plot(time, rv, hparam, "Cosine", model_list, model_par,
# kepplerian only plot relating to the kerplerian with parameters P_1, K_1, e
plot.keplerian_only_plot(time, rv, hparam, "Cosine", model_list, model_par,
# similar to kerplerian only but phase folded instead
plot.phase_plot(time, rv, hparam, "Cosine", model_list, model_par, rv_err, f
plot.phase_plot(time, rv, hparam, "Cosine", model_list, model_par, rv_err, f
```




Predicted Keplerian Model
Subtracted Dataset_1
Subtracted Dataset_2
Subtracted Dataset_3
RV [m/s]
Residuals
Time [BJD]


Predicted Keplerian Model
Subtracted Dataset_1
Subtracted Dataset_2
Subtracted Dataset_3
RV [m/s]
Residuals
Time [BJD]


Keplerian Model
Subtracted Dataset_1
Subtracted Dataset_2
Subtracted Dataset_3
RV [m/s]
Residuals
Time [BJD]



Running the MCMC

The MCMC is run in the same way as the previous tutorials however this time we are calculating the mass of these planets. To run with offsets, the run mcmc function requires flags as an input. To calculate the masses of the keplerians, we must give an extra output, in this case masses, and an extra input of Mstar, the mass of the host star in solar masses. The 5th output, masses, will return a 3d array with nrows = chains, ncolumns = keplerian, ndimensions = iterations. So in this case masses will have 2 columns, the first column being the masses of keplerian 0 and the second being the masses of keplerian 1.

In [ ]:
```
# set up iterations and number of chains
iterations = 100
numb_chains = 100
# to calculate masses we require 5 outputs
logL_chain, fin_hparams, fin_model_param, completed_iterations, masses = run
```

```
Initial hyper-parameter guesses:
[10.0, 10.0]

Initial model parameter guesses (ecc and omega are replaced by Sk and Ck):
[7.2, 30.0, 0.7071067811865476, 4.329780281177467e-17, 0.0, 50.0, 20.0, 5.6,
38.0, 0.5916079783099616, 0.5916079783099617, 0.0]

Initial Log Likelihood:  -60.63486494115643

Number of chains:  100

Start Iterations

Progress:  |██████████████████████████████████████████████████| 100.0% Comp
lete


100 iterations have been completed with 100 contemporaneous chains

Acceptance Rate =  0.14742574257425742
 ---- 0.207955269018809 minutes ----
```

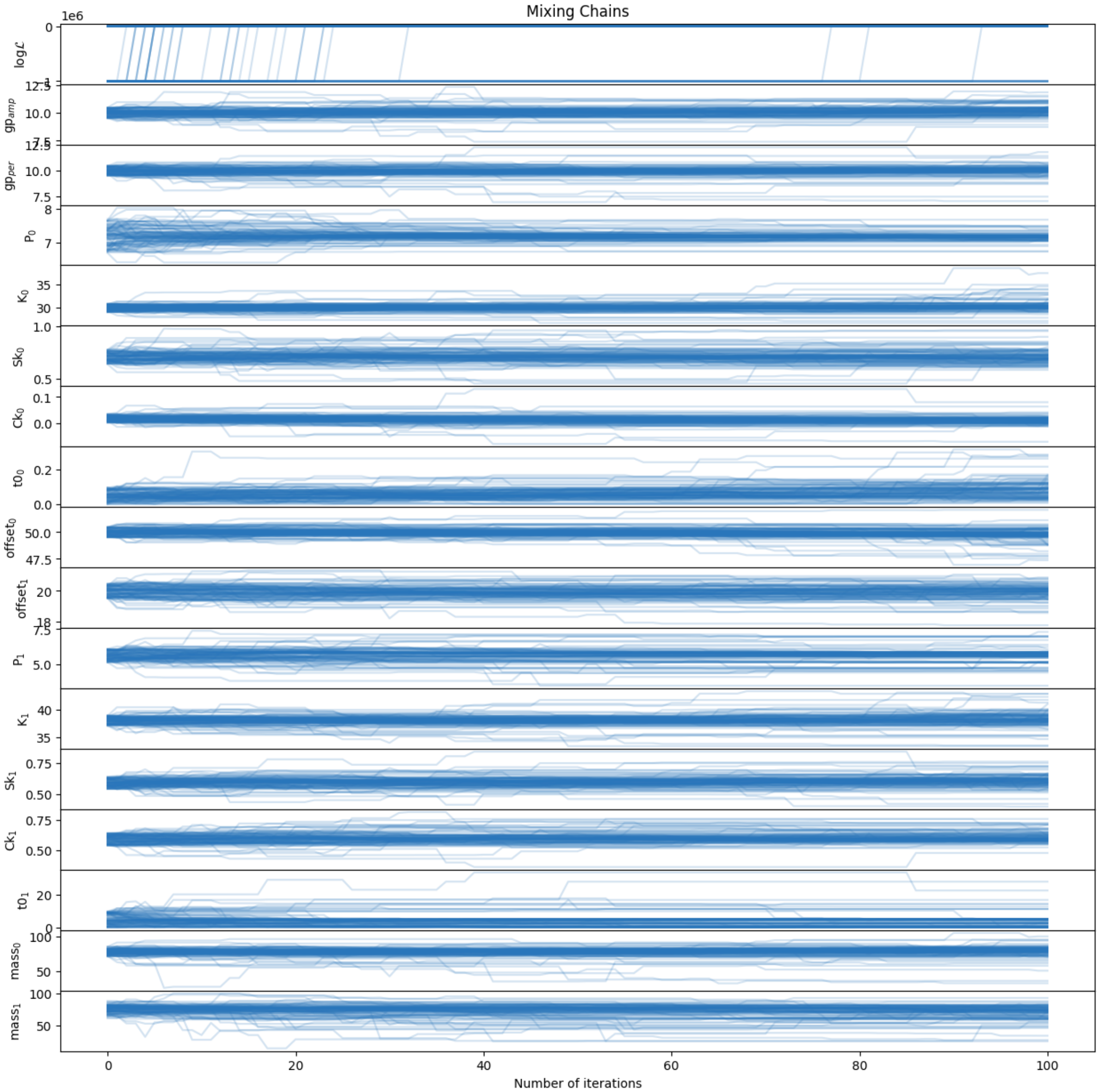
Mixing Plots and Corner Plots

These are also similar to previous tutorials except we must now give the masses array from the mcmc as an extra input to both plotting functions and the saving function. The corner plots will create an additional plot for just the masses and if saving is enabled it will save that additional plot. The mass subplots will also be added on to the end of the plot containing all parameters and hyperparameters.

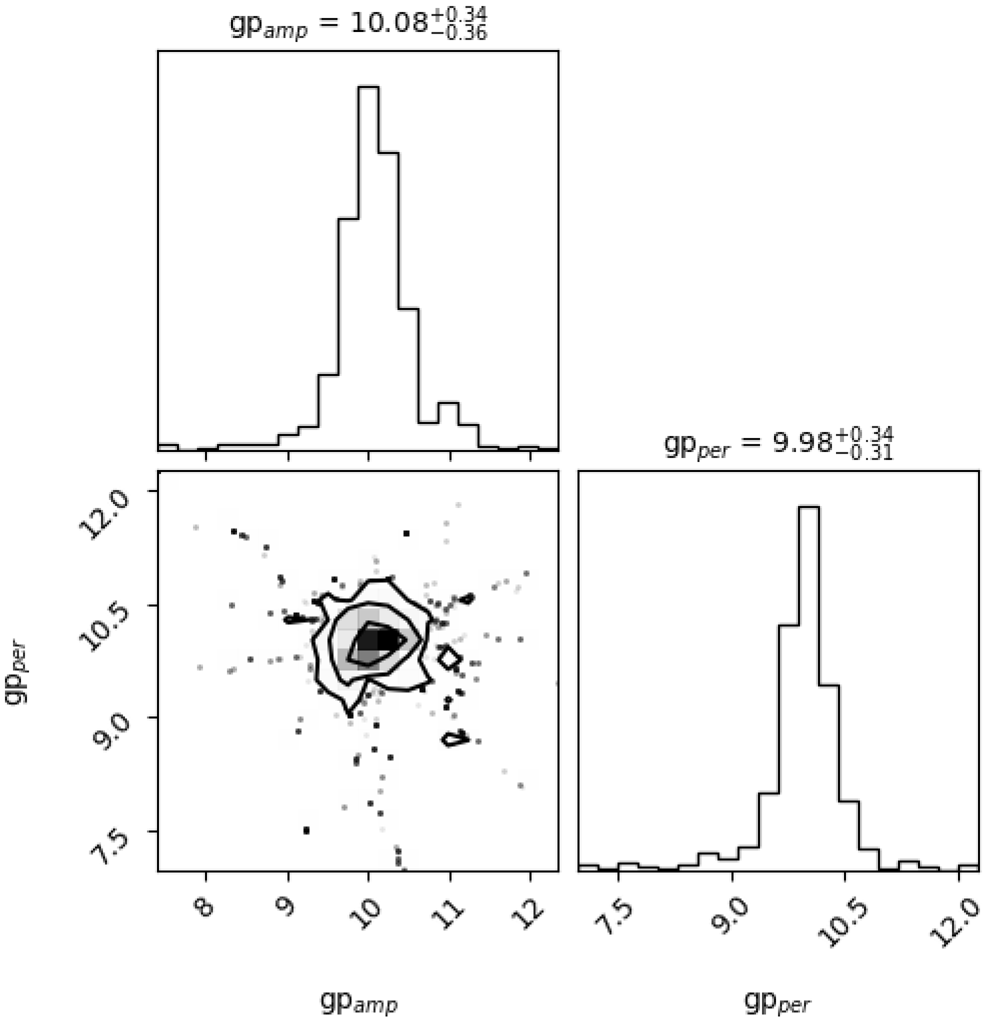
The outputs of the corner plot function will all contain the masses as the last values, in this case we have 2 masses so the last 2 values of these outputs will be the final mass values and the errors respectively.

With masses and Mstar inputted, the saving function will save the mass posteriors individually in their own files and add the final mass values and errors to the final parameter values file.

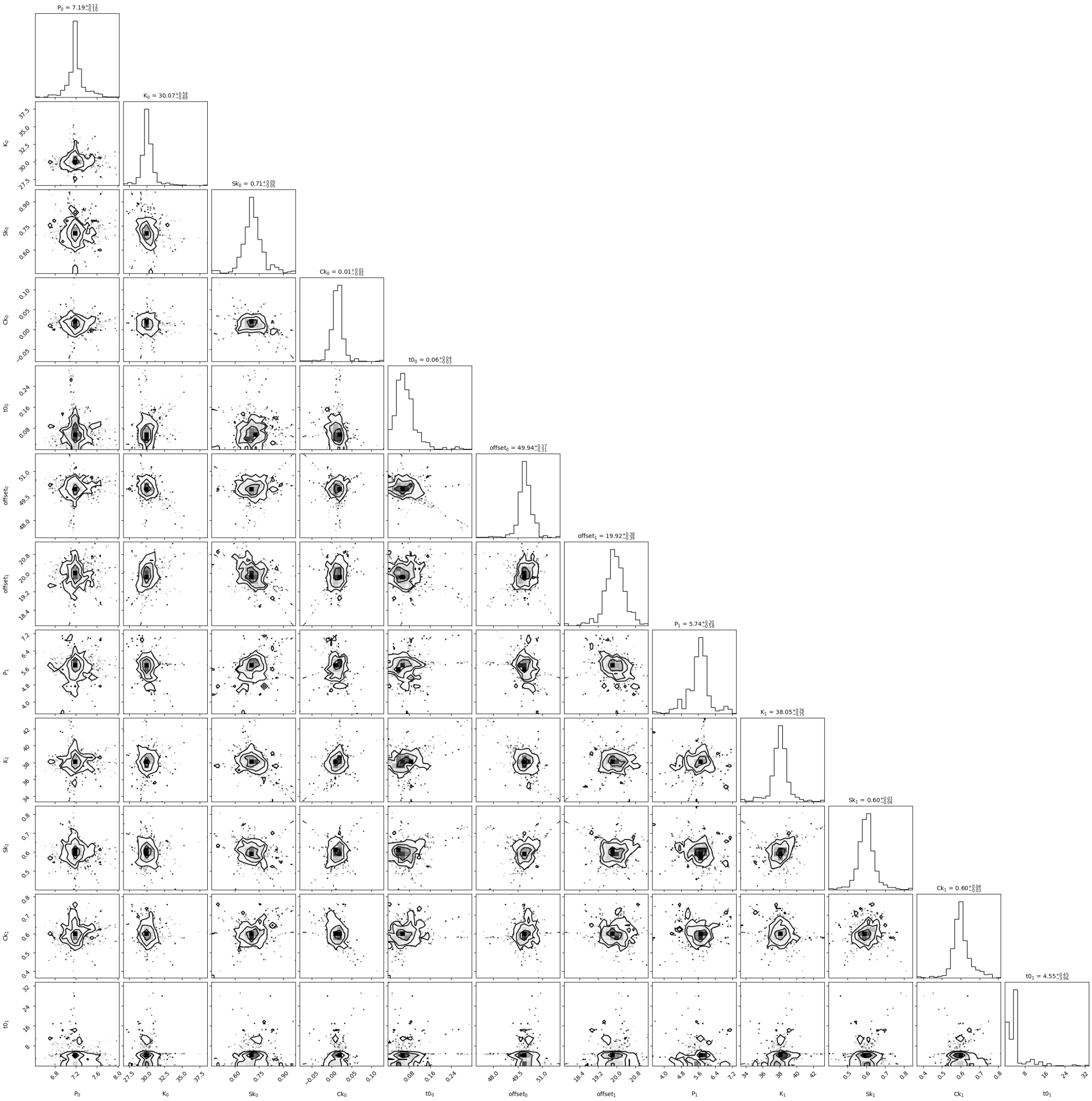
In [ ]:
```
plot.mixing_plot(fin_hparams, "Cosine", fin_model_param, model_list, logL_ch
```

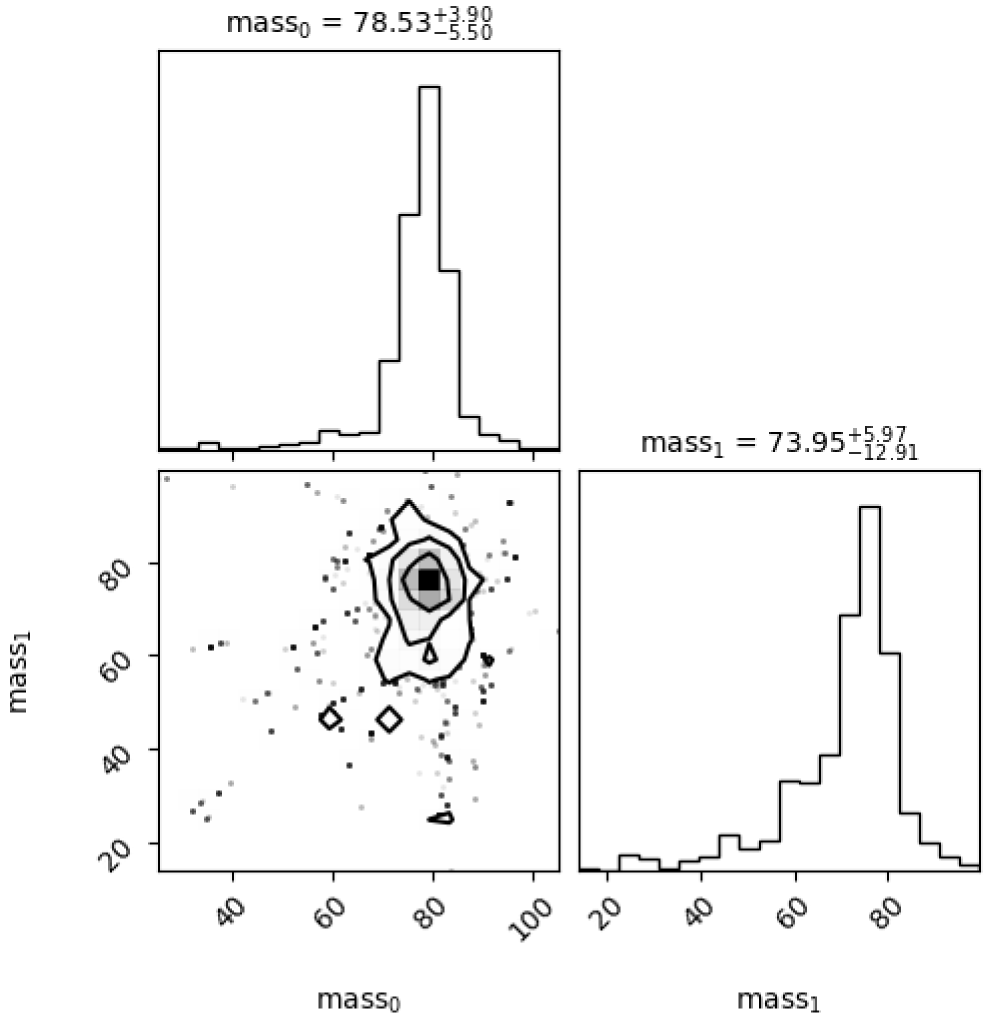


In [ ]:
```
final_param_values, final_param_erru, final_param_errd = plot.corner_plot(fi
```

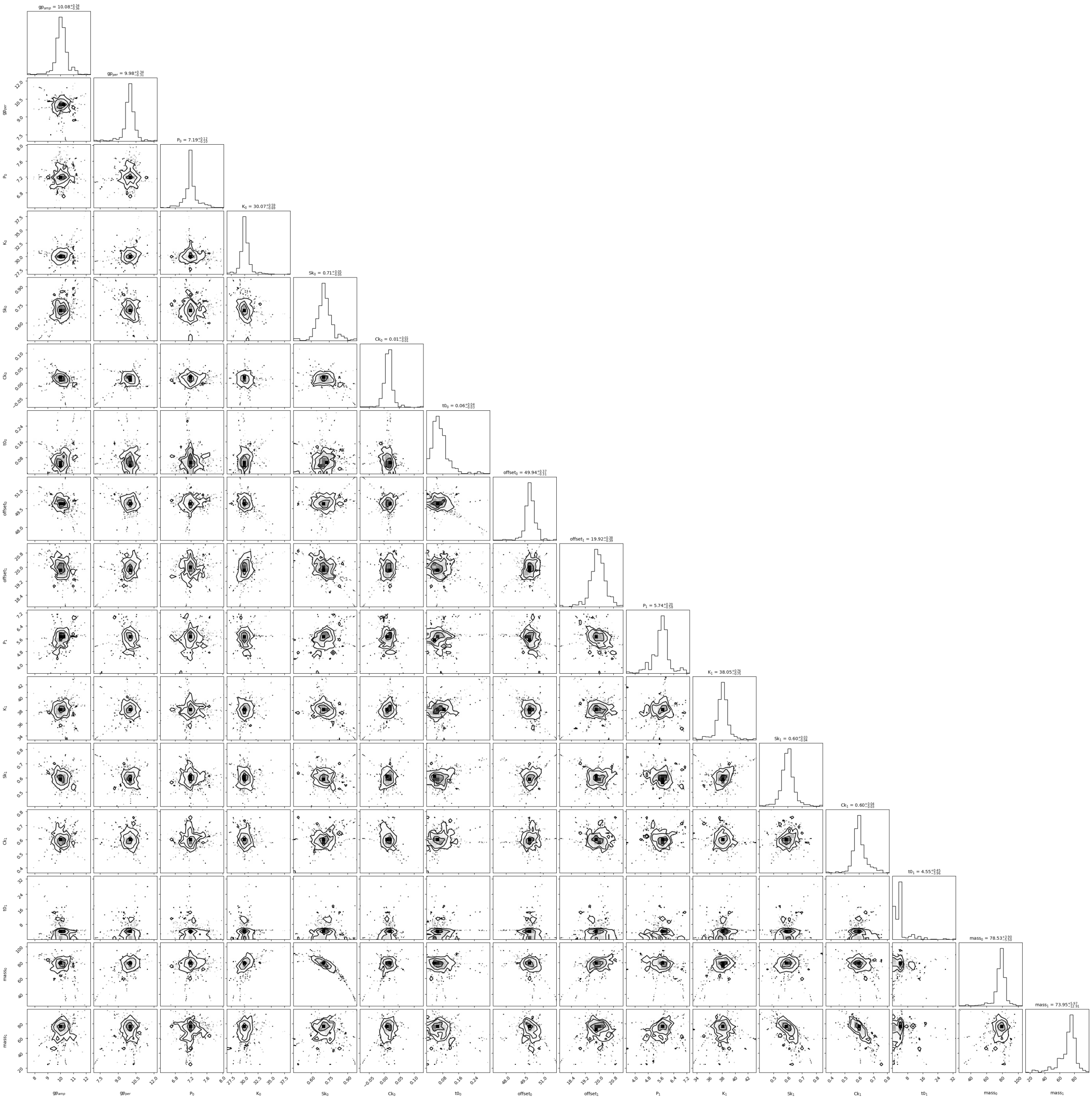


$gp_{amp} = 10.08^{+0.34}_{-0.36}$
$gp_{per} = 9.98^{+0.34}_{-0.31}$
gp_{per}
12.0
10.5
9.0
7.5
8
9
10
11
12
7.5
9.0
10.5
12.0
gp_{amp}
gp_{per}





$mass_0 = 78.53^{+3.90}_{-5.50}$
$mass_1 = 73.95^{+5.97}_{-12.91}$
$mass_1$
80
60
40
20
40
60
80
100
20
40
60
80
$mass_0$
$mass_1$



```
Parameter values after MCMC:  [10.083626953971372, 9.977009577629715, 7.193365329291762, 30.072248030937942, 0.7087233202749448, 0.01449244849715366, 0.060149574941863425, 49.94481605772422, 19.921452337172084, 5.735522045454013, 38.0500990028487, 0.5998675224860768, 0.5960628087481951, 4.545709696881568, 78.5316601935031, 73.94808384244823]
```

In [ ]:
```
save('/file/path/folder-name/', rv, time, rv_err, model_list = model_list, i
```

In [ ]: