Binary Orbit Fitting with Gaia DR3 and MCMC

Written by Logan A Pearce, 2020
https://github.com/logan-pearce (https://github.com/logan-pearce)
http://www.loganpearcescience.com (http://www.loganpearcescience.com)

Science case

Pearce et al. 2020 (accepted to ApJ, arxiv (https://arxiv.org/abs/2003.11106)) explored the applicability of using the *Gaia* satellite astrometry to constrain orbital solutions for wide stellar binaries for which both objects are well resolved by *Gaia*. Stars in wide binaries can have orbital peroids on the order of thousands of years or more, making observing orbital motion using time-series astrometric measurements costly in terms of observing resources and time. Often, even after observing a particular system for ~100 years, the curve of the orbit still is not visible due the exceedingly long orbital period. Nevertheless, understanding something of the orbit, even if it is only a lose constraint, can provide meaningful insight to the dynamics and formation history of a system. For example, recently Bryan et al. (2020, https://arxiv.org/pdf/2002.11131.pdf (https://arxiv.org/pdf/2002.11131.pdf)) measured the angular momentum vectors of the stellar spin, wide substellar companion spin, and the companion's orbit to provide the first ever measurement of all three for a substellar companion system, which allowed them conclude the disk graviational instability was the most likely formation scenario. This type of work, comparing wide companion orbital angular momentum vectors to other vectors such as circumstellar planet orbit, prototplanetary disks, or stellar spin axis, is a key piece of the puzzle for complex star and planet formation and evolution.

Traditional orbit fitting relies on astrometric measurements spanning some amount of time to observe orbital motion. Determining orbits by observing motion of graviationally bound objects go back as far as Galileo and Jupiter's moons. But wide stellar companions present a problem if thier periods are too long to observe orbital motion on a reasonable timescale. For example, DS Tuc AB is a wide stellar binary ($\rho \approx 240$ AU) that has been monitored in the Washington Double Star catalog (WDS) for ~100 years. Linear motion can be observed in the WDS astrometry, but orbital curvature still is not quite observed yet, making it difficult to place even loose constraints on the orbit of DS Tuc B relative to A. DS Tuc A hosts a transiting exoplanet with a well-constrained orbital inclination, making it good target for examining angular momentum vector alignment.

The *Gaia* satellite measures astrometry in the same way, by observing the motion of objects in time-series photometry, yet the Gaia archive reports a single time point measurement of position and velocity in the plane of the sky (and radial velocity for a small number of objects). So even though determined using time series, we have readily available and easily accessible velocities (and in the future accelerations) of gravitationally bound objects which we can exploit to place limits on their orbits.

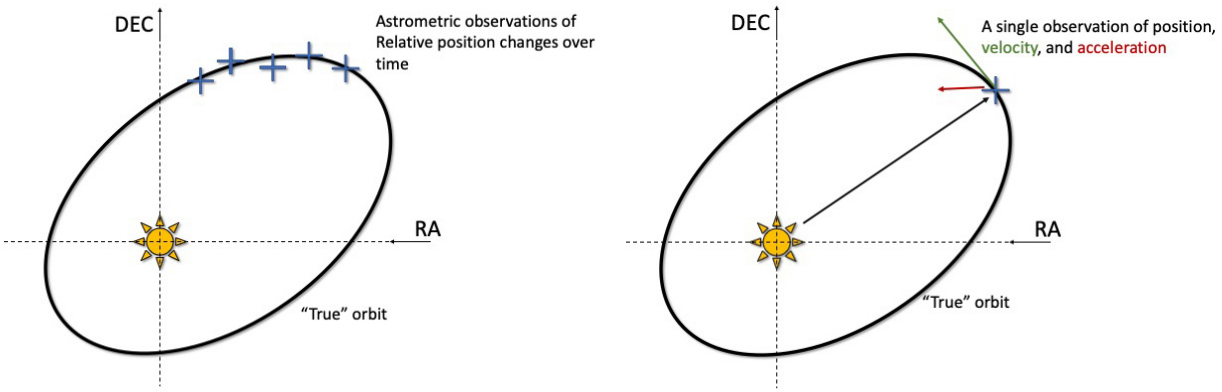


Figure 1: comparison of measurement constraints that can be used to find the "true" Keplerian orbit. Left: traditional orbit fitting of several realtive position measurements spaced in time. Right: *Gaia* -like measurements reported by the archive of a single time point of relative position, velocity, and acceleration.

DS Tuc A and B are both well resolved in *Gaia* DR2, with precise measurements for position and velocity in the plane of the sky, and both even have precise radial velocities (rare in DR2). In Newton et al. 2019 (https://arxiv.org/pdf/1906.10703.pdf (https://arxiv.org/pdf/1906.10703.pdf)), we used the *Gaia* DR2 measurements to provide the observational constraints rather than WDS time-series astrometry for fitting orbital parameters for the stellar binary. We found the Gaia observations were precise enough to provide meaningful information, and concluded the stellar binary is ~ aligned with the planet's orbital inclination, suggesting the system formed without much dynamical interactions. This led us to explore how useful *Gaia* measurements could be for a wide selection of wide binary systems.

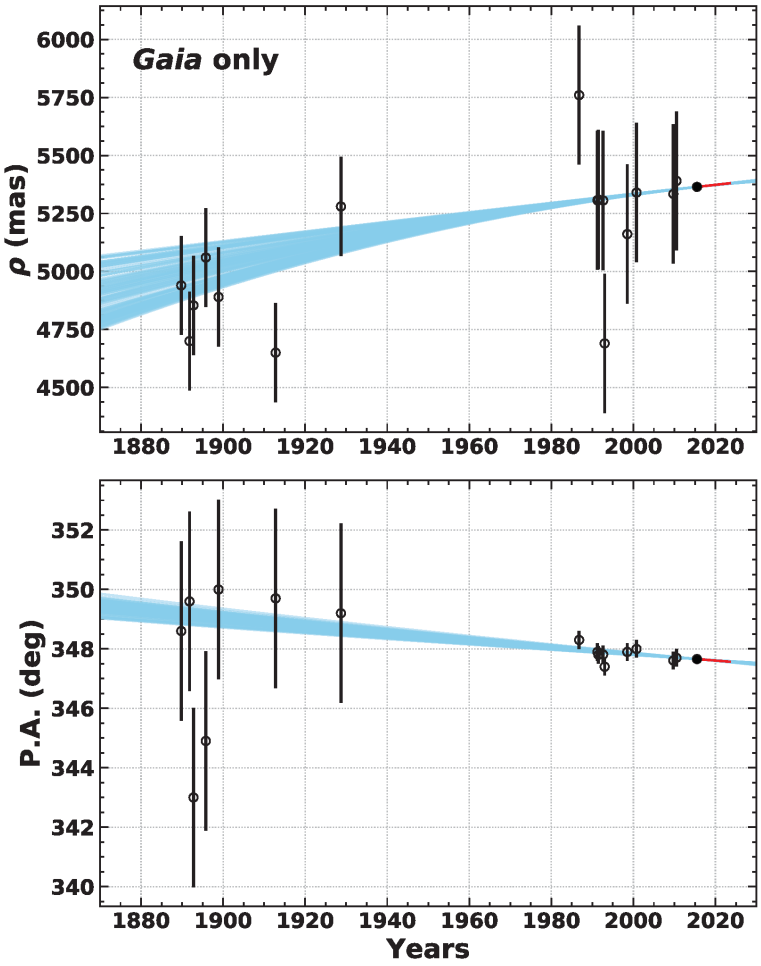


Figure 2 (from Figure 1 of Pearce+ 2020): Orbit fit results for DS Tuc B relative to DS Tuc A. The black points and error bars are the WDS astrometric measurements (not used in fit), the red line shows the *Gaia* DR2 velocity vector, and the blue lines are orbits from the posterior distribution of the orbit fit.

Pearce et al. 2020 includes a very important exploration of when *Gaia* astrometry should not be used to produce accurate or reliable posterior orbits for a system. Stars with unresolved inner companions, significant acceleration during *Gaia* observations, or close enough to be subject to psf overlap will have poor solutions and should not be relied upon for fitting their orbits. A check of the re-normalized unit weight error (RUWE) parameter is essential before using *Gaia* (RUWE $\approx$ 1.0 is desired; see Pearce+ 2020 Sec 2.2 for discussion)

Orbit fitting with *Gaia* DR2

Gaia DR2 measurements provide a very lose constraint on orbital parameters, so an MCMC fitting algorithm was not practical, because the posterior distribution of Keplerian orbital parameters were not much different from the priors. For our orbit fitting in Newton+ 2019 and Pearce+ 2020, we used a modified version of the Orbits for the Impatient (OFTI) rejection sampling algorithm. For a detailed discussion of the specifics of OFTI, see Blunt et al. 2017 (https://arxiv.org/pdf/1703.10653.pdf (https://arxiv.org/pdf/1703.10653.pdf)), and for a discussion of

when to use OFTI vs MCMC, see Blunt et al. 2019 (https://ui.adsabs.harvard.edu/abs/2019ascl.soft10009B/abstract (https://ui.adsabs.harvard.edu/abs/2019ascl.soft10009B/abstract)).

OFTI is a rejection sampling algorithm. Briefly, rejection sampling samples the parameter space by randomly generating a set of trial orbits from the prior probability distribution for each orbital parameter, then testing each trial orbit by determining a likelihood of the orbit given the data, and accepting the orbit if it is more probable than a randomly selected number on the interval (0,1). OFTI includes the additional speed up step of randomly generating 4 Keplerian parameters, then scaling semi-major axis and rotating position angle of nodes to match observation, eliminating much of the region of the parameter space that won't be likely.

The random generation of trial orbits allows OFTI to explore the full parameter space (without the risk of getting stuck in local minima), but can be prohibitively slow when the observations provide significant restraint, because the vast majority of parameter space will be excluded by observations. Fortunately for orbit fitting many systems with Gaia DR2, measurements were precise enough to allow us to say something meaningful about the orbit, but sufficiently imprecise to make OFTI a good choice for orbit fitting. Many of the fits in the Pearce+ 2020 paper were completed in less than 1 hour.

Orbit fitting with *Gaia* DR3

Gaia DR3 promises more precise measurements of many more objects and additional observational parameters including acceleration in the plane of the sky for some objects. For wide binaries without unresolved subsystems, the measured acceleration will provide further constraint on the binary orbit. (Again, it must be stressed that this will only apply when there are not inner subsystems of sufficient size to influence the observed acceleration. Later Gaia data releases will include accelerations precise enough to discover new planets and constrain masses of companions, which exciting. But that will render this orbit fitting method ineffective. This can only be done in the absence of subsystems.)

The addition of two additional measurement constraints (accel in RA and accel in DEC) makes OFTI a less practical option for finding posterior orbits. We should switch to an MCMC- based algorithm for accomplishing the fit.

In this report, I will show how I adapted the use of Gaia measrements of position, velocity, and acceleration to constrain Keplerian orbital elements for a wide stellay binary system using the `emcee` affine-invariant MCMC ensemble sampler python package .

First, I will describe the considerations for fitting Keplerian orbital elements using MCMC and a single time-point measurement of position and its derivatives, and the algorithm I developed to accomplish it. Then I will demonstrate my algorithm using a fictional system with a fictional "true" orbit. Finally I will compare the results of my algorithm run on DS Tuc AB (with fictional acceleration measurements) to results using the established OFTI fitter (modified to accept accelerations).

Algorithm + Test orbit

Coordinate system

I followed the same coordinate system of Pearce+ 2020 for which +X = +DEC, +Y = +RA, +Z = towards observer, forming a right handed system for which +Dec is the reference direction as in astrometry (however this is contrary to the radial velocity convention of +Z = away from observer, so radial velocities retrieved from *Gaia* must have applied the opposite sign).

Parameterization

Keplerian (elliptical) orbits are described by 7 parameters:

- semi-major axis (a)
- period (P)
- eccentricity (e)
- inclination relative to reference plane (i)
- argument of periastron (ω)
- longitude of nodes (Ω)
- epoch of periastron passage (t_0)

Where here the reference plane is the plane of the sky and the reference direction in +Dec

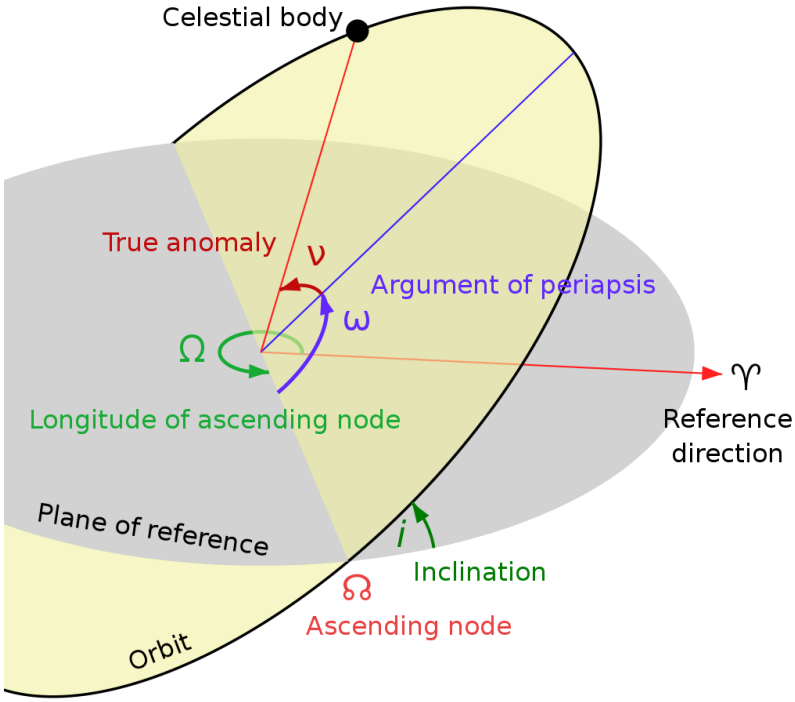


Figure 3: Keplerian orbital elements. Image credit: Lasunncty at the English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8971052

If we know the total mass of the system, we can eliminate the period through Kepler's 3rd law. This requires use of an observational mass obtained from an external observation, and we will include the observational uncertainties in our fit.

Likewise, we need to be able to transform between physical separation (AU) and projected separation (arcsec), so we need to include the distance to the system. Which we can obtain from *Gaia*, but must include its observational uncertainties in the fit as well.

Additionally, I have chosen to parameterize the epoch of periastron passage through the orbit phase, τ, at observation date, rather that the epoch of periastron passage, because this simplifies the prior significantly.

So, our 8-dimensional parameter array for fitting will be:

$\theta = \{a, e, i, \omega, \Omega, \tau, mass, dist\}$

Test orbit:

I will generate a fictional "true" orbit to develop and test the algorithm:

In [1]:
```
import numpy as np
import astropy.units as u

#A choice of mass, distance, and observation date:
dist = (44.1, 0.063) # pc
mtot = (1.2,0.1) # Msun
obsdate = 2015.5 # Gaia DR2 reference date

# sma in arcsec
a_true = (2.5,0.005) # arcsec
# epoch of periastron passage in decimal year:
to_true = (1965.5,0.3)
# ecc:
e_true = (0.21,0.01)
# incl:
i_true = (np.radians(48.2),np.radians(1.02))
# arg of periastron:
w_true = (np.radians(13.4),np.radians(0.95))
# long of nodes:
O_true = (np.radians(346.9),np.radians(2.3))

# Compute period from Kepler's 3rd law (Monte Carlo for errors):
# convert sma to AU:
a_au_true = np.random.normal(a_true[0],a_true[1],1000)*np.random.normal
a_au_true = (np.mean(a_au_true),np.std(a_au_true))
# period in years:
P_true = np.sqrt((np.random.normal(a_au_true[0],a_au_true[1],1000)**3)/
P_true = (np.mean(P_true),np.std(P_true))
```

```
This is what the orbit looks like projected onto the plane of the sky. The grey indictes the effect of the parameter uncertainties, and color displays the orbit phase $\tau$, with $\tau$ = 0 occuring at t = 2015.5, indicated by the blue circle.  This is the location of the fictional companion at the *Gaia* DR2 observation epoch

<img src="LPearce_report_images/true_orbit.png" width=700>
Figure 4: Trial orbit described by the above parameters projected onto the plane of the sky.  Grey shading shows effect of uncertainties in parameters.  Color indicates phase, or fraction of orbit completed at that location, with phase = 0 at the *Gaia* DR2 reference date of 2015.5
```

Next I will generate a fictional set of observations from the true orbit and its uncertainties. I will use functions I wrote for the OFTI-based orbit fitter, called `lofti_gaiaDR2`, to compute the position, velocity, and acceleration of a companion on this orbit at t = 2015.5. And Kepler's equation solver as well. (The functions can be found here: https://github.com/logan-pearce/orbittools/blob/master/orbittools/orbittools.py (https://github.com/logan-pearce/orbittools/blob/master/orbittools/orbittools.py))

In [2]:
```
from orbittools.orbittools import calc_XYZ, calc_velocities, calc_accel

# Generate simulated measurements from those orbital parameters:
# Make a Monte Carlo array of orbits:
n = 1000
true_array = np.array([np.random.normal(a_true[0],a_true[1],n),
              np.random.normal(P_true[0],P_true[1],n),
              np.random.normal(to_true[0],to_true[1],n),
              np.random.normal(e_true[0],e_true[1],n),
              np.random.normal(i_true[0],i_true[1],n),
              np.random.normal(w_true[0],w_true[1],n),
              np.random.normal(O_true[0],O_true[1],n)])

# Compute values for each orbit in the array:
# Position in arcsec:
pos = calc_XYZ(*true_array,obsdate, solvefunc = danby_solve)
# velocities in km/s
vel = calc_velocities(*true_array,obsdate,dist[0], solvefunc = danby_so
# accelerations in m/s/yr
acc = calc_accel(*true_array,obsdate,dist[0], solvefunc = danby_solve)
```

Now we'll generate fake *Gaia* measurements for this system:
(Note: calc_XYZ etc. returns results for X, Y, Z directions in that order)

In [3]:
```
# Take mean and std deviation
dRA = (np.mean(pos[1]),np.std(pos[1]))
dDec = (np.mean(pos[0]),np.std(pos[0]))
dpmRA = (np.mean(vel[1]),np.std(vel[1]))
dpmDec = (np.mean(vel[0]),np.std(vel[0]))
dRV = (np.mean(vel[2]),np.std(vel[2]))
daccRA = (np.mean(acc[1]),np.std(acc[1]))
daccDec = (np.mean(acc[0]),np.std(acc[0]))
print('delta RA',dRA)
print('delta DEC',dDec)
print('delta pmRA',dpmRA)
print('delta pmDEC',dpmDec)
print('delta RV',dRV)
print('delta accel RA',daccRA)
print('delta accel DEC',daccDec)
```

```
delta RA (0.4801829264921518, 0.07822318024634159)
delta DEC (1.7015502545869723, 0.04271057926018868)
delta pmRA (2.4886855822904366, 0.1262251009114835)
delta pmDEC (-1.6503988357426214, 0.16820319292637315)
delta RV (2.29655068591654, 0.0842791094524388)
delta accel RA (-6.792653454442777, 1.4135892166666952)
delta accel DEC (-23.92996197013056, 1.8103136094877714)
```

In [4]:
```
# put data into one container:
obs = np.array([dRA,dDec, dpmRA, dpmDec, dRV, daccRA, daccDec])
data = obs[:,0]
error = obs[:,1]
```

Define a few functions to make working with stuff easier:

In [5]:

```
def t0_to_tau(t0, period, ref_epoch = 2015.5):
    """
    Convert epoch of periastron passage (t0) to orbit fraction (tau)
    Args:
        t0 (float or np.array): value to t0 to convert, decimal years
        ref_epoch (float or np.array): reference epoch that tau is def
        period (float or np.array): period (in years) that tau is defi
    Returns:
        tau (float or np.array): corresponding taus
    """
    tau = (ref_epoch - t0)/period
    tau %= 1

    return tau

def tau_to_t0(tau, period, ref_epoch = 2015.5, after_date = None):
    """
    Convert tau (epoch of periastron in fractional orbital period afte
    T0 in decimal years.  Will return as the first periastorn passage
    Args:
        tau (float or np.array): value of tau to convert
        ref_epoch (float or np.array): date that tau is defined relati
        period (float or np.array): period (in years) that tau is nora
        after_date (float): T0 will be the first periastron after this
    Returns:
        t0 (float or np.array): corresponding T0 of the taus in decima
    """

    t0 = ref_epoch - (tau * period)

    if after_date is not None:
        num_periods = (after_date - t0)/period
        num_periods = int(np.ceil(num_periods))

        t0 += num_periods * period

    return t0

def period(sma,mass):
    """ Given semi-major axis in AU and mass in solar masses, return t
        Kepler's third law.
        Written by Logan Pearce, 2019
    """
    import numpy as np
    import astropy.units as u
    # If astropy units are given, return astropy unit object
    try:
        sma = sma.to(u.au)
        mass = mass.to(u.Msun)
        period = np.sqrt(((sma)**3)/mass).value*(u.yr)
    # else return just a value.
    except:
        period = np.sqrt(((sma)**3)/mass)
    return period

def physical_separation(d,theta):
```

```
    """
    Returns separation between two objects in the plane of the sky in A
    Distance and parallax must be astropy unit objects.
    Args:
        d (float): distance
        theta (float): parallax
    Return:
        separation in AU
    Written by: Logan Pearce, 2017
    """
    from astropy import units as u
    d = d.to(u.pc)
    theta = theta.to(u.arcsec)
    a = (d)*(theta)
    return a.to(u.au, equivalencies=u.dimensionless_angles())

def angular_separation(d,a):
    """
    Returns separation between two objects in the plane of the sky in a
    physical separation in AU.
    Distance and separation must be astropy unit objects.
    Args:
        d (float): distance
        a (float): separation
    Return:
        theta in arcsec
    Written by: Logan Pearce, 2017
    """
    from astropy import units as u
    d = d.to(u.pc)
    a = a.to(u.au)
    theta = a / d
    return theta.to(u.arcsec, equivalencies=u.dimensionless_angles())

def logminmax(minAU, maxAU, dist):
    """
    Given a minumum and maximum desired distance in AU, and
    distance to system in parsecs, return those values in log
    space in arcsec

    Args:
        minAU, maxAU (float): min/max separations in AU
        distance (float): distance in parsec
    Return:
        logmin, logmax (float): min/max separations in log arcseconds
    """
    # Convert to arcsec:
    minAS = angular_separation(dist*u.pc,minAU*u.au)
    maxAS = angular_separation(dist*u.pc,maxAU*u.au)
    # Take log:
    logmin = np.log(minAS.value)
    logmax = np.log(maxAS.value)
    return logmin, logmax
```

Set up the MCMC sampler

Priors

Priors for each parameters are as follows:

- sma: Log Uniform (Jefferys prior), or 1/x probability distribution
- ecc: Uniform on (0,1)
- inc: Sin prior on (0,π)
- ω: Uniform on (0, 2π)
- Ω: Uniform on (0, 2π)
- τ: Uniform on (0,1)
- mass: Gaussian with μ = observational value, σ = observational uncertainty
- distance: Gaussian with μ = observational value, σ = observational uncertainty

(these are standard priors for Keplerian orbital elements)

In [6]:

```
# Define Prior functions:
def UniformPrior(x, xmin = -10., xmax = 10.):
    lnprob = 0. if xmin < x < xmax else -np.inf
    return lnprob

def SinPrior(x, interval = [0.,np.pi]):
    lnprob = np.log(np.sin(x))
    if (x < interval[0]) or (x > interval[1]):
        lnprob = -np.inf
    return lnprob

def LogUniformPrior(x, logmax=6, logmin=0):
    lnprob = -np.log((x))
    if (np.log(x) < logmin) or (np.log(x) > logmax):
        lnprob = -np.inf
    return lnprob

def GaussianPrior(x, mu = 0, sigma = 1):
    lnprob = -0.5*np.log(2.*np.pi*sigma) - 0.5*((x - mu) / sigma)**2
    return lnprob


# Establish the priors specific to our system:
def LogPriors(params):
    ''' Compute the prior probability of a proposed set of
        Keplerian orbit fitting parameters.
        Parameters:
        -----------
        params : 1d array
            proposed values for Keplerian orbital elements in order:
                sma [as] : semi-major axis in arcsec (as)
                ecc : eccentricity
                inc [rad] : inclination in radians
                argp [rad] : arguement of periastron in radians
                lon [rad] : longitude of nodes in radians
                tau : orbit phase, or fraction of orbit completed at re
                mass [Msun] : total system mass in solar masses
                distance [pc] : distance of system from Earth in parsec
        Returns:
        --------
        logprior : float
            log(prior probability) of proposed values for parameters
    '''
    logprior = 0.
    # sma:
    logmin, logmax = logminmax(0.001,1e7, dist[0])
    logprior += LogUniformPrior(params[0], logmin = logmin, logmax = lo
    # ecc:
    logprior += UniformPrior(params[1], xmin = 0., xmax = 1.)
    # incl:
    logprior += SinPrior(params[2])
    # argp:
    logprior += UniformPrior(params[3], xmin = 0., xmax = 2*np.pi)
    # lon:
    logprior += UniformPrior(params[4], xmin = 0., xmax = 2*np.pi)
    # tau:
```

```
    logprior += UniformPrior(params[5], xmin = 0., xmax = 1.)
    # mass:
    logprior += GaussianPrior(params[6], mu = mtot[0], sigma = mtot[1])
    # distance:
    logprior += GaussianPrior(params[7], mu = dist[0], sigma = dist[1])

    return logprior
```

Likelihood function:

In [7]:

```
def ComputeModel(params):
    ''' Compute the 3d value of position, velocity, and acceleration
        at reference date for proposed parameter values.  +X = DEC, +Y
        +Z = towards observer
        Parameters:
        -----------
        params : 1d array
            proposed values for Keplerian orbital elements in order:
                sma [as] : semi-major axis in arcsec (as)
                ecc : eccentricity
                inc [rad] : inclination in radians
                argp [rad] : arguement of periastron in radians
                lon [rad] : longitude of nodes in radians
                tau : orbit phase, or fraction of orbit completed at re
                mass [Msun] : total system mass in solar masses
                distance [pc] : distance of system from Earth in parsec
        Returns:
        --------
        pos : 1x3 array
            X, Y, Z position of test particle on proposed orbit in arcs
        vel : 1x3 array
            dotX, dotY, dotZ velocities of test particle in km/s
        acc : 1x3 array
            ddotX, ddotY, ddotZ accelerations in m/s/yr
    '''
    # Unpack parameters array:
    a,e,inc,w,O,tau,m,d = params
    # Compute Period:
    P = period(physical_separation(d*u.pc,a*u.arcsec), m).value
    # Compute to:
    to = tau_to_t0(tau,P)
    # Position in arcsec:
    pos = calc_XYZ(a,P,to,e,inc,w,O,obsdate, solvefunc = danby_solve)
    # velocities in km/s
    vel = calc_velocities(a,P,to,e,inc,w,O,obsdate,d, solvefunc = danby
    # accelerations in m/s/yr
    acc = calc_accel(a,P,to,e,inc,w,O,obsdate,d, solvefunc = danby_solv
    return pos, vel, acc

def LogLikelihood(params, data, error):
    ''' Compute the likelihood of the proposed model given observations
        Parameters:
        -----------
        params : 1d array
            proposed values for Keplerian orbital elements in order:
                sma [as] : semi-major axis in arcsec (as)
                ecc : eccentricity
                inc [rad] : inclination in radians
                argp [rad] : arguement of periastron in radians
                lon [rad] : longitude of nodes in radians
                tau : orbit phase, or fraction of orbit completed at re
                mass [Msun] : total system mass in solar masses
                distance [pc] : distance of system from Earth in parsec
        data : 1d array
            observations in the following order:
                dRA [as] : relative separation of two bodies in RA dire
```

```
                dDec [as] : rel separation in DEC direction in arcsec
                dpmRA [km/s] : rel difference in proper motion in RA in
                dpmDec [km/s] : rel difference in proper motion in Dec
                dRV [km/s] : rel difference in radial velocity
                daccRA [m/s/yr] : rel difference in acceleration in RA
                daccDec [m/s/yr] : rel difference in acceleration in De
        error : 1d array
            error in observations in the same order as data

        Returns:
        --------
        chi2lnlike : float
            log likelihood of model given observations
    '''
    # Compute model for proposed values:
    pos, vel, acc = ComputeModel(params)
    # take the model results that correspond to observation
    # dimensions in the correct order:
    model = [pos[1],pos[0],vel[1],vel[0],vel[2],acc[1],acc[0]]
    residual = (data - model)
    chi2lnlike = -0.5 * np.sum(residual**2 / error**2 - np.log(np.sqrt(

    return chi2lnlike
```

Total probability function:

In [8]:

```
def LogProbability(params, data, error):
    ''' Compute the probability of the proposed model given observation
        Parameters:
        -----------
        params : 1d array
            proposed values for Keplerian orbital elements in order:
                sma [as] : semi-major axis in arcsec (as)
                ecc : eccentricity
                inc [rad] : inclination in radians
                argp [rad] : arguement of periastron in radians
                lon [rad] : longitude of nodes in radians
                tau : orbit phase, or fraction of orbit completed at re
                mass [Msun] : total system mass in solar masses
                distance [pc] : distance of system from Earth in parsec
        data : 1d array
            observations in the following order:
                dRA [as] : relative separation of two bodies in RA dire
                dDec [as] : rel separation in DEC direction in arcsec
                dpmRA [km/s] : rel difference in proper motion in RA in
                dpmDec [km/s] : rel difference in proper motion in Dec
                dRV [km/s] : rel difference in radial velocity
                daccRA [m/s/yr] : rel difference in acceleration in RA
                daccDec [m/s/yr] : rel difference in acceleration in De
        error : 1d array
            error in observations in the same order as data

        Returns:
        --------
        lnp + lnlike : float
            log probability of model given observations
    '''
    # Priors:
    lnp = LogPriors(params)
    # If any proposal is outside the parameter's range, computing the m
    # will fail and return a "nan", so if any prior returns -inf,
    # we skip the likelihood step and return -inf:
    if ~np.isfinite(lnp):
        return -np.inf
    # Else, compute likelihood of model given data:
    lnlike = LogLikelihood(params, data, error)
    return lnp + lnlike

```

Initialize the sampler object

In [9]:
```
nwalkers = 1000 # number of walkers
ndim = 8 # number of fit parameters
steps = 5000 # how many samples per chain

import emcee
sampler = emcee.EnsembleSampler(nwalkers,
                                ndim,
                                LogProbability, # function for determin
                                args=(data, error) # args that go into
                               )
```

Initial position for walkers

The `emcee` docs example initializes initial positions for the walkers by choosing random values that are close to the "true" answer. For my purposes, I won't know a close answer because I can't use a process like MLE to estimate a close value. Additionally, my observational constraints, even with the accelerations, aren't going to be sufficient to arrive at anything like a Gaussian posterior (a "true" asnwer) for the orbtial parameters. *Gaia* only provides one constraint in the Z direction (RV; it will not be precise enough to constrain line of sight position, and will not include a line of sight acceleration), and the orbital parameters aren't independent, there will be some correlation among them. I am interested in placing lose constraints on wide binary orbits, so I need the MCMC to fully explore the entire parameter space, to find minima with some probability away from the "right" answer. So, I initialized the walkers by drawing initial positions randomly from the prior distributions for the parameters.

In [10]:
```
def DrawSamples(N):
    """ Draw random values for Keplerian orbit fit parameters from appr
        Parameters:
        -----------
        N : int
            Number of random samples to draw

        Returns
        -------
        sma,ecc,inc,argp,lon,tau,m,d : 1xN arrays
            random values for each parameter in units [as,n/a,rad,rad,r

    """
    # sma:
    logmin, logmax = logminmax(0.001,1e7, dist[0])
    sma = np.random.uniform(logmin, logmax, N)
    sma = np.exp(sma)
    # ecc:
    ecc = np.random.rand(N)
    # incl:
    cosi = np.random.uniform(-1.0,1.0,N)
    inc = np.arccos(cosi) % np.pi
    # argp:
    argp = np.random.uniform(0.,2*np.pi,N)
    # lon:
    lon = np.random.uniform(0.,2*np.pi,N)
    # tau
    tau = np.random.rand(N)
    # mass:
    m = np.random.normal(mtot[0],mtot[1],N)
    # distance:
    d = np.random.normal(dist[0],dist[1],N)

    return sma,ecc,inc,argp,lon,tau,m,d

initial_pos = DrawSamples(nwalkers)
initial_pos = np.stack(initial_pos).T
initial_pos.shape
```

Out[10]: (1000, 8)

Run sampler

In [ ]:
```
sampler.run_mcmc(initial_pos, steps, progress='notebook')
# Note - I will not actually run the fitter here because it takes ~1.5
# a trial run while developing the architecture.
```

Some things I learned about using emcee.

- The probability function and likelihood function **must** take as inputs the parameter array first, and then whatever you give for the `args` keyword. At first I tried computing the model outside of the likelihood function, which meant I only needed to give the likelihood function the data and errors arrays and not the parameter array. The sampler ran fine, but produced posteriors that were identical to the priors. Whatever emcee was doing under the hood did not allow it

understand computing the model outside the likelihood function without passing the parameter array to the likelihood function.

- Also, I tried using the 2d `obs` array of data as one column and error as the other in the likelihood and probability functions, and setting `args = (obs)`, and that also did not give correct results. Again, the fitter ran, but the results looked like the priors. Separating data and error into different arrays is the only way it worked. Not sure why that wasn't successful.
- This led to another problem. My model function is much more complex the the example in the emcee docs. I kept running into the problem that the sampler would run just fine for a while then crash out saying the probability function had returned a nan. But it's not possible for my probability function to give a nan? I finally figured out that when the sampler chose values outside the domain of my parameters, the prior function returned -inf, but the values were still being fed into my model function, even though they were outside the range of the parameters. If eccentricity > 1.0, the model function breaks and returns a nan. There was no mechanism to prevent feeding outside values into the model function. So I had to add a statement to the likelihood function to skip computing the model if a parameter is outside its allowed range. Which is better anyway because there is no need to take the CPU time to compute a model if the log(probability) will be -inf anyway from the prior function.
- For future upgrades: My model function is expensive to compute, each iteration took ~1 sec. The most expensive part is solving the Kepler equation, which I do 3 times in the model function. To speed it up, I should rewrite the computations to only solve for E once, and write the solver such that it uses C to do the computation rather than python.

Results

```
#### Chains as a function of walker step:
<img src="LPearce_report_images/testrun_chains_largerinit.png">
Figure 5: chain values as a function of walker step.  Each black line
shows the value of one of the 1000 walkers at each step.  It is clear
that the walkers found regions with highest probability, but found
some other regions also consistent with observations but with lower
probabilities.  There are orbits of high semi-major axis and high
eccentricity that might also produce our observed velocity and
acceleration.  The chains eventually converge near the known true
answer, but I am interested in exploring the full parameter space,
even areas with low probability, so I took 2000 steps as burn in.

#### Corner plot:
Full corner plot with 2000 step burn in:
<img src="LPearce_report_images/testrun_corner_largerinit.png">
Figure 6: Joint posterior distribution for fit parameters.  Contours
mark 16th, 50th, and 84th quantile (default for corner.py), shading
increases with density.  The blue squares and lines mark the "true"
orbital parameters.  We see that the true parameter values fall in the
highest density regions of the posterior, but there are there are more
extreme regions of parameter space with some probability.

There are two distinct families of orbits consistent with
observations, most readily visible in the inclination and eccentricity
joint posterior plots.  The true value falls within the most dense
regions.

```

```
Corner plot with most extreme family of orbits remove to make sma dist more readable:
<img src="LPearce_report_images/testrun_corner_largerinit_chop.png">
Figure 7: same as Figure 6 with exteme values removed for readability

The semi-major axis (sma)/ eccentricity joint distribution is correlated, which makes sense physically - a more circular smaller sma and more elliptical larger sma produce the same projected orbit. We also see that the ecc/incl joint dist. is also correlated.  This also makes sense physically, there should be a relation between eccentricity of the ellpise and its projection angle onto the sky - more eccentric and more inclined can look the same as less eccentric lecc inclined when projected onto the sky.  There is also a family of orbits with i ~ 1.6 rad/ e ~ 0.6 in the non-truncated plot, because they correspond to extreme value of sma (a > 1000 arcsec).
$\Omega$/$\tau$/ $\omega$ are also bimodal, which also make sense physcially in that changing the location of $\Omega$ and/or $\omega$ changes the location of periastron, which would change the orbit fraction, which is measured from periastron passage.  Although the highest density regions are where the true values are located.

In summary, the results of this test are consistent with expectations for a losely-constrained fit of orbital elements, and the MCMC algorithm is working as designed.
```

Test against literature with *Gaia*

DS Tuc AB is the best test case I have seen with *Gaia* DR2 astrometry, in that both have high quality astrometric solutions with RUWE ~ 1.0, and both objects have radial velocity measurements (this is the only wide binary I've seen in *Gaia* with RV for both). It also has published results using the OFTI algorithm with *Gaia* astrometry in Newton et al. 2019. So as an additional test, I ran the new MCMC algorithm on the DS Tuc system.

Since *Gaia* DR2 does not include acceleration terms, I created fictional acceleration terms in RA/DEC by taking accelerations from an orbit in the posterior distribution in Newton+ 2019.

Get *Gaia* observations

I will borrow from my public code for OFTI-based *Gaia* orbit fitting to retrieve astrometry from the *Gaia* archive and convert it into observational constraints on the binary orbit. For more details this code can be found here: https://github.com/logan-pearce/lofti_gaiaDR2 (https://github.com/logan-pearce/lofti_gaiaDR2)

In [19]:
```
from lofti_gaiaDR2.lofti import prepareconstraints

# DS Tuc AB source IDs:
DSTucA = 6387058411482257536
DSTucB = 6387058411482257280
# Masses from Newton+ 2019:
massA = (0.97, 0.04)
massB = (0.87, 0.04)

# This function queries the Gaia archive, retrieves the astrometry, and
# the values needed for orbit fitting:
deltaRA, deltaDec, pmRA_kms, pmDec_kms, deltarv, total_pos_velocity, to
    rho, pa, delta_mag, d_star, ruwe = prepareconstraints(DSTucA, DSTuc
# total mass:
mA, mAerr = np.float(massA[0]),np.float(massA[1])
mB, mBerr = np.float(massB[0]),np.float(massB[1])
mtot = (mA + mB, np.sqrt((mAerr**2) + (mBerr**2)))
print('Delta RA, err in mas:', deltaRA[0],'+/-', deltaRA[1])
print('Delta Dec, err in mas:', deltaDec[0],'+/-', deltaDec[1])
print()
print('pmRA, err in km/s:',pmRA_kms[0],'+/-', pmRA_kms[1])
print('pmDec, err in km/s:',pmDec_kms[0],'+/-', pmDec_kms[1])
if deltarv != 0.:
    print('deltaRV, err im km/s (pos towards observer):',deltarv[0],'+/
print()
print('Total relative velocity [km/s]:',total_velocity_kms[0],'+/-',tot
print('Total plane-of-sky relative velocity [mas/yr]:',total_pos_velocit
print()
print('sep,err [mas]',rho[0],'+/-',rho[1], 'pa,err [deg]:',pa[0],'+/-',
print('sep [AU]',(rho[0]/1000)*d_star[0])
print('D_star',d_star[0],'+\-',d_star[1])
print('Delta Gmag',delta_mag)
print('RUWE source 1:', ruwe[0][0])
print('RUWE source 2:', ruwe[1][0])
print('Total mass [Msun]', mtot)
# convert to as:
deltaRA = (deltaRA[0]/1000.,deltaRA[1]/1000.)
deltaDec = (deltaDec[0]/1000.,deltaDec[1]/1000.)
```

```
Delta RA, err in mas: -1146.6531704914323 +/- 0.016048657816882164
Delta Dec, err in mas: 5240.633805784656 +/- 0.03128627531346939

pmRA, err in km/s: -0.30173712008430653 +/- 0.02044456492818036
pmDec, err in km/s: 0.3543703626789322 +/- 0.012045661197205923
deltaRV, err im km/s (pos towards observer): 1.8793611665844168 +/- 0.737
675600888463

Total relative velocity [km/s]: 1.9361358521672773 +/- 0.7380571592599962
Total plane-of-sky relative velocity [mas/yr]: 2.2246301214145014 +/- 0.1
1376315413334326

sep,err [mas] 5364.611465895851 +/- 0.03080522544561477 pa,err [deg]: 34
7.6581527665023 +/- 0.00018107116029660936
sep [AU] 236.76196920386982
D_star 44.1340385429632 +\- 0.06336868730526682
Delta Gmag -1.0800505
RUWE source 1: 1.0344639
```

```
RUWE source 2: 1.0149378
Total mass [Msun] (1.8399999999999999, 0.0565685424949238)
```

Before proceeding, I want to check that the separation/position angle/distance match what I expected, that both objects have acceptable RUWE (RUWE $\lesssim$ 1.2), to be sure that using these values for orbit fitting will return reliable results. Everything looks good so I'll proceed.

The orbit fitting I did for Newton+ 2019 included a posterior distribution of the 3d accelerations, so I selected acceleration values and reasonable errors from there:

In [15]:
```
daccRA = (1.1,0.1) #m/s/yr
daccDec = (-4.8,0.1) #m/s/yr
```

In [16]:
```
# put data into one container:
obs = np.array([deltaRA,deltaDec, pmRA_kms, pmDec_kms, deltarv, daccRA,
data = obs[:,0]
error = obs[:,1]
```

I'll use the same walker set up as before, and randomly draw an initial walker position from the entire parameter domain as before.

In [ ]:
```
initial_pos = DrawSamples(nwalkers)
initial_pos = np.stack(initial_pos).T

# re-initialize the sampler
#(if you don't do this, or call sampler.reset(), it will add the new sa
# chain as before)
sampler = emcee.EnsembleSampler(nwalkers,
                                ndim, # number of fit parameters
                                LogProbability, # function for determin
                                args=(data, error) # args that go into
                               )

sampler.run_mcmc(initial_pos, 4000, progress='notebook')
```

Again I won't run it here because it takes ~1 hr to run. Here are the results for this configuration:

Chains as a function of walker step:

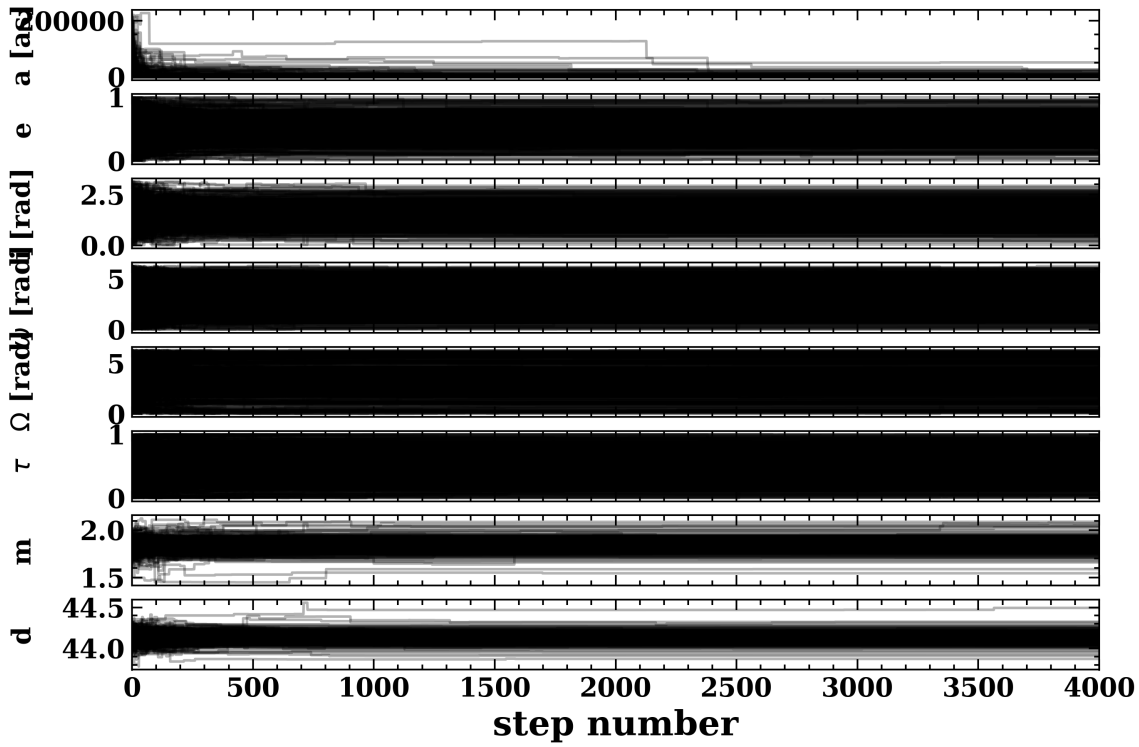
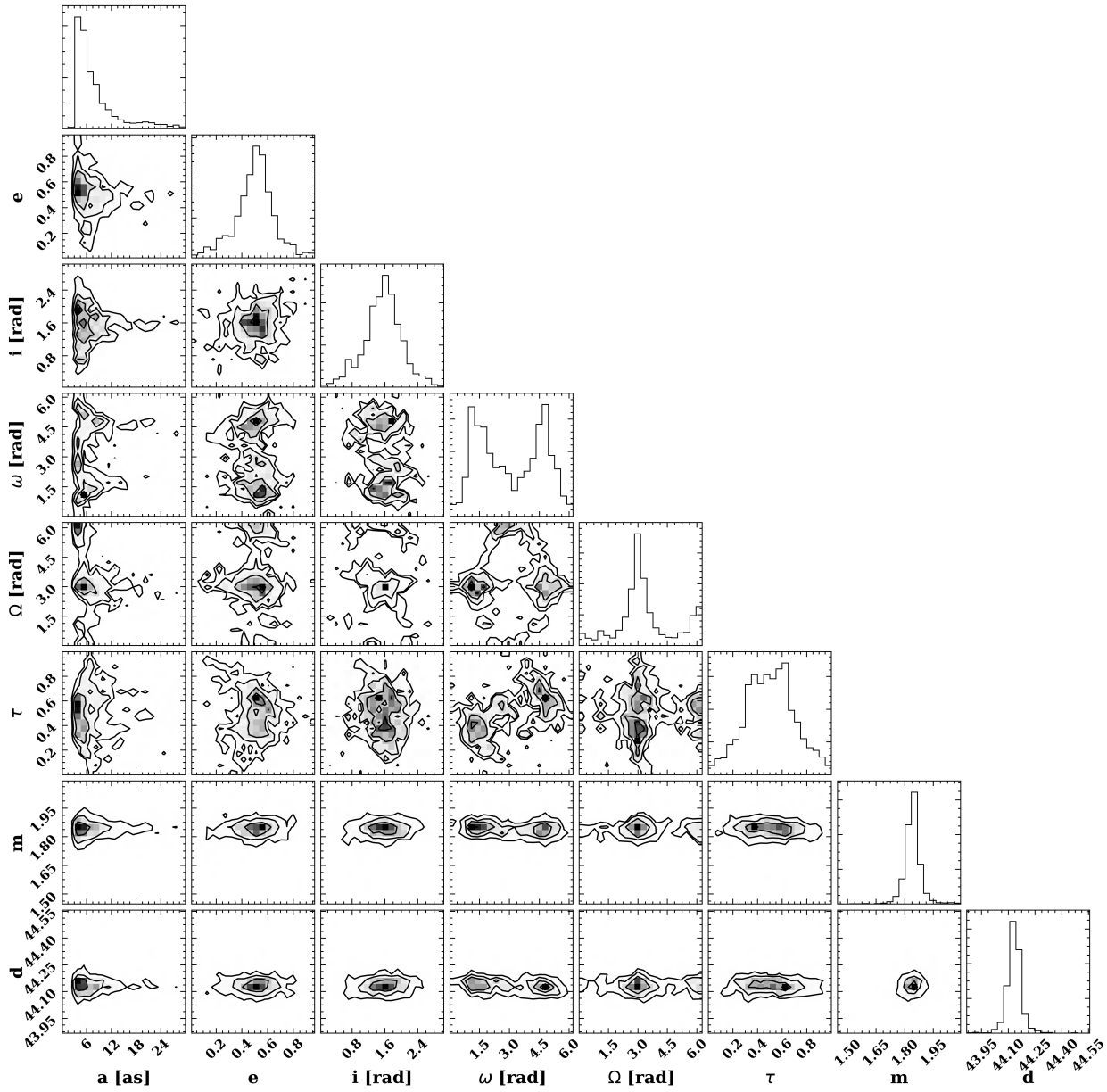


Figure 8: Chain values as a function of walker step. The chains never converge, as expected with this poorly-constrained parameter space, but they seem to settle into their final values around step 1000.

Corner plot:

Figure 9: Corner plot with 1000 step burn in, with most extreme semi-major axis values removed:



Again, there are some large outliers at extreme regions of parameter space (some excluded from this plot for readability), which I expect, there should be extreme orbits that can produce observed velocity and acceleration (well, "observed" acceleration). I am concerned that inclination values spanned pi/2. Inclinations 0 < i < 90 deg correspond to counterclockwise motion on the sky, and 90 < i < 180 deg clockwise motions, so inc spanning 90 deg would be two different motions on the plane of the sky. And since radial velocity is included as an observation, Ω should be constrained to one mode, which we don't see here; while most values are near π, there is another smaller mode near 0. ω being bimodal and there being a spread in τ seems the things that allows there to be such a spread in inclination, but this doesn't seem quite right to me given that the sign of RV is known. I tried to think about this physcially but I'm still not sure these results are reasonable and will need to think more on this.

Test using initial starting position:

Because of this strange result, I decided to run another test that is more typical of an MCMC exploration, to find a good fitting initial value and start all walkers near those values. For my initial starting position I chose values near the modes of the distributions for the OFTI fit from Newton+

2019:

In [21]:
```
a_init, e_init, i_init_deg, w_init_deg, O_init_deg, to_init =  (3.706,0
    0.5, 95.383, 245.73, 347.234, 1551.608
mtot = (1.839, 0.056)
a_au_init = np.random.normal(a_init[0],a_init[1],1000)*np.random.normal
a_au_init = (np.mean(a_au_init),np.std(a_au_init))
# period in years:
T_init = np.sqrt((np.random.normal(a_au_init[0],a_au_init[1],1000)**3)/
T_init = (np.mean(T_init),np.std(T_init))

tau_init = t0_to_tau(to_init,T_init[0])

params_init = [a_init[0], e_init, np.radians(i_init_deg), np.radians(w_
               np.radians(O_init_deg), tau_init, mtot[0], dist[0]]
params_init
```

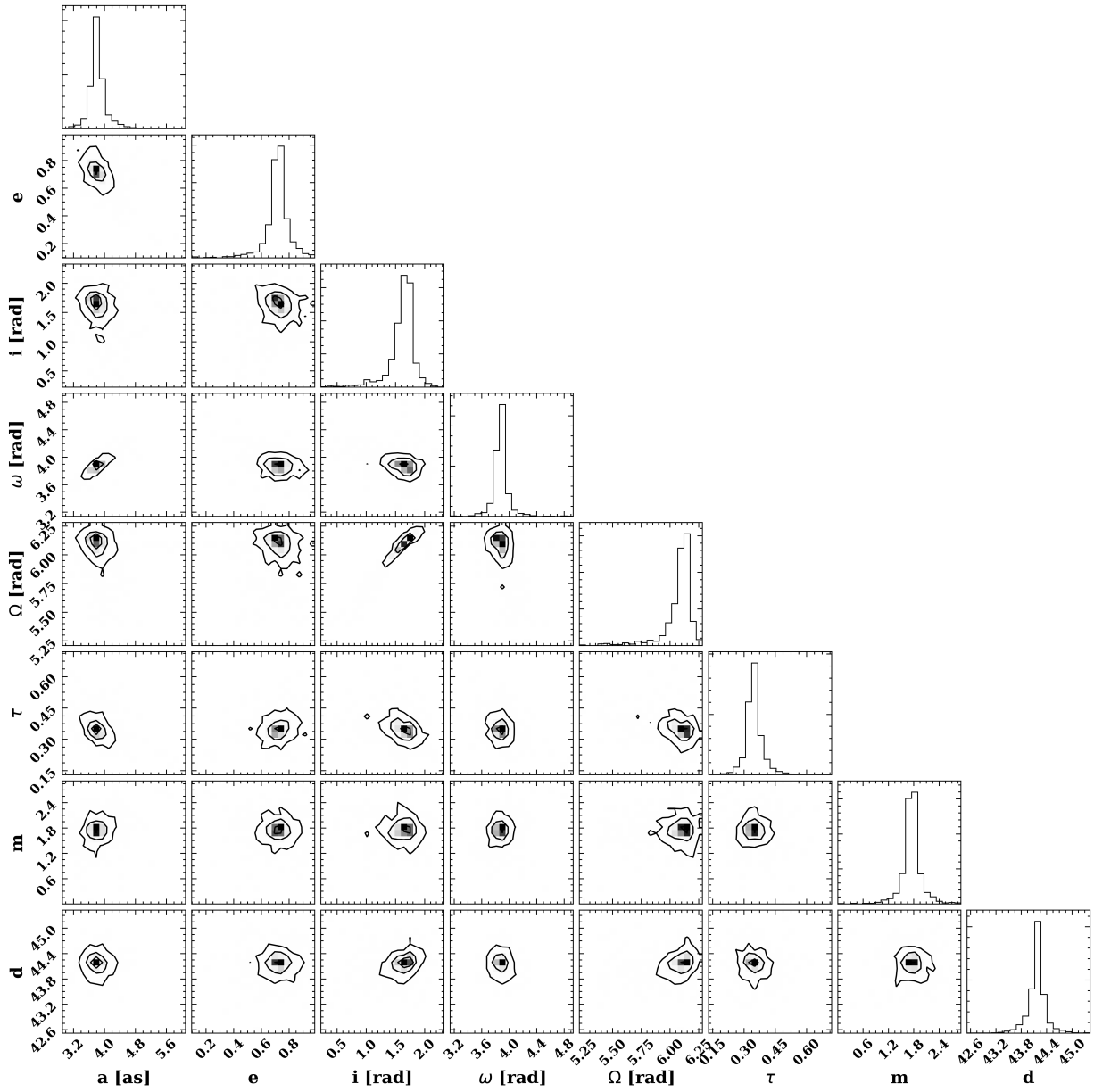
Out[21]:
```
[3.706,
 0.5,
 1.6647474004297513,
 4.288797570925666,
 6.06037657486999,
 0.30102190963201525,
 1.839,
 44.1]
```

In [ ]:
```
initial_pos = params_init + 1e-2 * np.random.randn(1000, 8)

sampler = emcee.EnsembleSampler(nwalkers,
                                ndim, # number of fit parameters
                                LogProbability, # function for determin
                                args=(data, error) # args that go into
                               )

sampler.run_mcmc(initial_pos, 4000, progress='notebook')
```

Results:

Corner plot:

Figure 10: Corner plot of samples initializing the walkers near modal parameter values, with 1000 step burn in, with most extreme semi-major axis values removed:



These results are much more confined, there are no real outliers in extreme regions of parameter space. Ω and ω are both constrained to one mode here, the one nearest the initial value I gave it. So the walkers did not find the other local minimum in those parameters. τ and inclination are noe much tighter as well.

Inclination is tightly confined near 90 degrees, which matches the findings of Newton+ 2019 that inclination was nearly aligned with the transiting planet around DS Tuc A. Although, inclination still spans across $\pi/2$. I still do not understand that. Eccentricity is now confined to a rather extreme value (and far from the initial value of 0.5; I chose rather large values of acceleration), though sma is close to the initial value.

Comparison to OFTI fit

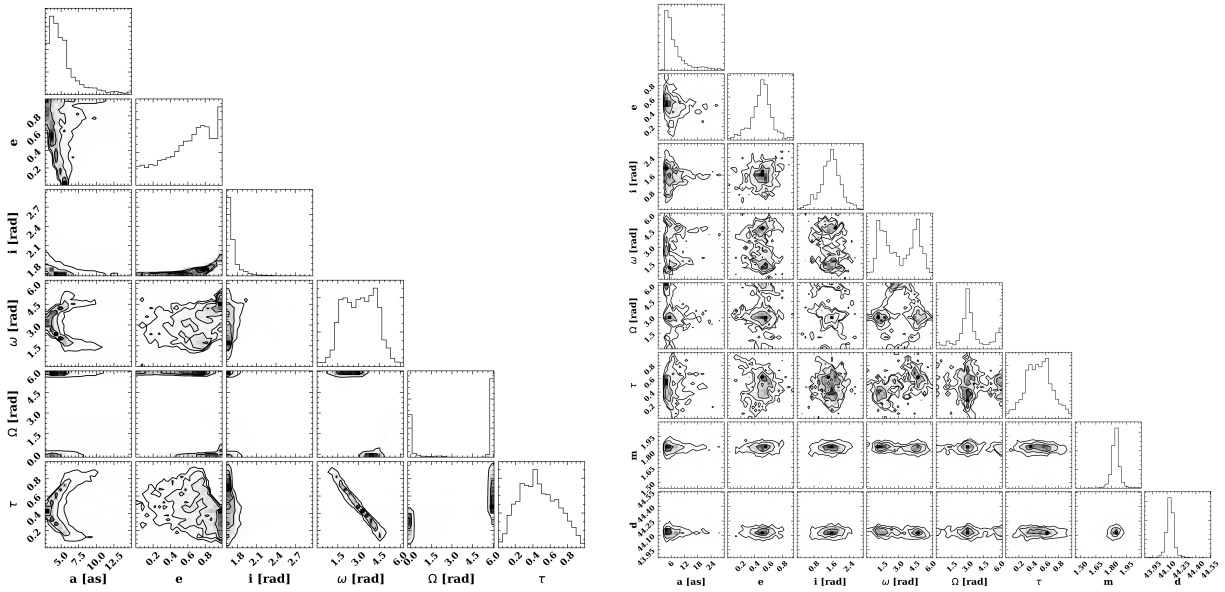
For comparison, I ran an OFTI fit to these same observational constraints by modifiying my existing architecture (https://github.com/logan-pearce/lofti_gaiaDR2 (https://github.com/logan-pearce/lofti_gaiaDR2)) to accomodate additional acceleration terms. A corner plot is shown below.

(Mass and distance are not fit parameters in OFTI, but their observational uncertainties are accounted for in the "scale and rotate" step; I excluded them from this corner plot because they were not part of the fit).

Figure 11:

OFTI sample results

MCMC with walker initial values drawn from full paramerter space



I would expect these results to be similar to the random initial value emcee results, and to some degree they are. High eccentricity is preferred, compared to the , and ω shows a range of values rather than being bimodal; τ shows a similar shape. Ω has only one mode however, which is interesting. Inclination is confined to be only >90, which is what I had expected from the velocity vector. The modes are similar (with the exception of Ω, which is off by pi), but the posterior shapes are different.

Discussion and Conclusion

Using the relative position, velocity, and acceleration at a single time point, measurements expected to be provided by *Gaia* DR3, can be successfully used to meaningfully constrain orbital elements for wide stellar binary systems for which both objects are resolved by *Gaia*. The ease of access to *Gaia* measurements makes this technique a boon to observational studies of angular momentum vector alignments in star and planet formation processes. This level of observational constraint makes rejection sampling too inefficient to be practical, and an MCMC-based algorithm is a better choice for fitting orbital elements. I constructed a fitting algorithm that uses the `emcee` python package in its default settings, an ensemble sampler with stretch moves using default parameters.

I found that the MCMC algorithm was successful at sampling the paramter space in a way consistent with known physics of Keplerian orbit and given the relatively lose constraints provided by *Gaia*-like measurements. When applied to a real physical system, the resulting posteriors were sensitive to the assumptions made when doing the fitting, to be expected with a Bayesian

algorithm. The most apparently successful result came when I used OFTI to find modes in parameter space, then initialized the walkers from those modes (Figure 10); this resulted in approximately Gaussian posteriors for orbital parameters. However, this may not be the physcially correct orbit. It excluded more extreme regions of parameter space that still had some amount of probability and could produce the observed orbit. Given the lose measurement constraints, a more robust exploration of parameter space is justified, and Gaussian posteriors should not necessarily be expected. I suggest initializing the walkers from the full prior distribution is perhaps the better choice for this application. Careful consideration of what is physcially most likely for wide stellar binaries is necessary before deciding on a course of action here.

It is useful to compare this to how other orbit fitting algorithms do this. The best comparison is to the new pyton package `orbitize!` (Blunt et al. 2019; https://ui.adsabs.harvard.edu/abs/2019ascl.soft10009B/abstract (https://ui.adsabs.harvard.edu/abs/2019ascl.soft10009B/abstract)), which is designed to fit orbits of time-series astrometry for directly images exoplanets. `orbitize!` allows the user to select from either OFTI or MCMC fitting algorithms, depending on the level of their measurement constraints. The MCMC `orbitize!` fitter is similar to what I've designed here, and uses ensemble sampling with stretch moves via `emcee`. They initialize their walkers from the full distribution of parameter space rather than finding any sort of best fit first. On the contrary, Dupuy et al. 2016 (https://ui.adsabs.harvard.edu/abs/2016ApJ...817...80D/abstract (https://ui.adsabs.harvard.edu/abs/2016ApJ...817...80D/abstract)) fit the orbit of the spectroscopic binary Kepler-444 BC around Kepler-444 A using many epochs of astrometry and radial velocity measurements. They used a LM least squares estimate first (using Thile-Innes constants to parameterize the orbit as they vary linearly) to obtain a best fit, then used MCMC to obtain approximately Gaussian posteriors for orbital parameters. So this is very much up to the assumptions of the users. As expected of a Bayesian system.

In conclusion, *Gaia* DR3 can be useful in studies of star and planet formation by constributing to angular momentum vector alignment studies, as long as the system's astrometric solution is sufficiently precise and there are no unresolved subsystems. If you find an applicable binary system, I suggest making use of MCMC fitting to accomplish the fit, and initializing the walkers from the full prior probabiliy paramter space. If you must find something like a "best fit" first, a short OFTI run could be an effective means to do this.

Future directions for improvement

- I am still uncomfortable with the fact the inclination spans both sides of 90 degrees. I was unable to reconcile this in my mind, and it deserves further thinking about.
- The model computation steps is very expensive. On average I saw between 1-2 sec per iteration during fitting, and since I was asking ~4000 steps this meant fits took 1-2 hours each. The most expensive part of the model is solving Kepler's equation. I modified the model function to only do this once (rather than three times as origially written) but did not see much speed up. I could move the model step to be done in C rather than python (`orbitize!` saw significant speed up when they did this). Additionally, I can use parallel processing under the hood to increase speed.
- I did not explore the Parallel Tempering version of `emcee` (`ptemcee`), or the Metropolis Hastings moves in `emcee`. MH would be advantgeous because the user can control how the walkers move around the parameter space by setting a proposal function and step sizes.

- Originally I had wanted to try making this a Hamiltonian MCMC, and I think I would still like to try it. Given that the posterior aren't thin or with distinct shape, this may not be advantageous, but I would like ot learn how Hamiltonain MCMC works by building one.

In [ ]:
```
1
```