```
'''
Version 0.0
- The initial version of the old TSM_v11 that is now being tracked in Git.
'''
from __future__ import division
import numpy
import scipy.integrate
import pickle
import profiles
import time
import sys
import cosmo

# Constants and conversions
G = 4.3*10**(-9) #newton's constant in units of Mpc*(km/s)**2 / M_sun
c = 3e5 #speed of light km/s
sinGyr = 31556926.*10**9 # s in a Giga-year
kginMsun = 1.98892*10**30 # kg in a solar mass
kminMpc = 3.08568025*10**19 # km in a Megaparsec
minMpc = 3.08568025*10**22 # m in a Megaparsec
r2d = 180/numpy.pi # radians to degrees conversion factor

def randdraw(A,index=None):
    '''
    This function randomly draws a variable from either and input gaussian
    distribution or an array of random draws from a parent distribution function
    (e.g. bootstrap sampling).
    Input:
    A = [(float, float) or (1D array of floats)] If (float, float) is input then
        it is interpreted as (mu, sigma) of a Gaussian distribution. If (1D
        array of floats with length > 2) then it is interpreted as a random
        sample from a parent distribution.
    index = [int] rather than drawing a random value from the array the value
    Output:
    a = [float] a random draw from either the Gaussian or sample distribution.
    '''
    N_A = numpy.size(A)
    if N_A == 2: #then assumed (mu, sigma) format
        a = A[0]+A[1]*numpy.random.randn()
    elif N_A > 2: #Then assume distribution array format
        if index == None:
            index = numpy.random.randint(N_A)
        a = A[index]
    elif N_A <2:
        print 'MCMAC.randdraw: Error, parameter input array is not a valid size, exiting'
        sys.exit()
    return a, index

def vrad(z1,z2):
    '''
    Given the redshifts of the two subclusters this function calculates their
    relative line-of-sight velocity (km/s).
    Input:
    z1 = [float] redshift of subcluster 1
    z2 = [float] redshift of subcluster 2
    Output:
    v_los = [float; units:km/s] line-of-sight relative velocity of the two
        subclusters
    '''
    v1 = ((1+z1)**2-1)/((1+z1)**2+1)*c
    v2 = ((1+z2)**2-1)/((1+z2)**2+1)*c
    return numpy.abs(v1-v2)/(1-v1*v2/c**2)

def f(x,a,b):
    '''
    This is the functional form of the time since merger integral for two point
    masses.
    '''
```

```
    return 1/numpy.sqrt(a+b/x)
def TSMptpt(m_1,m_2,r_200_1,r_200_2,d_end,E):
    '''
    This function calculates the time it takes for the system to go from a
    separation of r_200_1+r_200_2 to d_end. It is based on the equations of
    motion of two point masses, which is valid in the regime where the
    subclusters no longer overlap.
    Input:
    m_1 = [float; units: M_sun] mass of subcluster 1
    m_2 = [float; units: M_sun] mass of subcluster 2
    r_200_1 = [float; units: Mpc] NFW r_200 radius of subcluster 1
    r_200_2 = [float; units: Mpc] NFW r_200 radius of subcluster 2
    d_end = [float; units: Mpc] the final separation of interest
    E = [float; units: (km/s)^2*M_sun] the total energy (PE+KE) of the two subcluster system
    Output:
    t = [float; units: Gyr] the time it takes for the system to go from a
        separation of r_200_1+r_200_2 to d_end
    '''
    d_start = r_200_1+r_200_2
    C = G*m_1*m_2
    mu = m_1*m_2/(m_1+m_2)
    if E < 0:
        integral = scipy.integrate.quad(lambda x: f(x,E,C),d_start,d_end)[0]
        t = numpy.sqrt(mu/2)*integral/sinGyr*kminMpc
    else:
        print 'TSMptpt: error total energy should not be > 0, exiting'
        sys.exit()
    if t < 0:
        print 'TSM < 0 encountered'
    return t


def PEnfwnfw(d,m_1,rho_s_1,r_s_1,r_200_1,m_2,rho_s_2,r_s_2,r_200_2,N=100):
    '''
    This function calculates the potential energy of two truncated NFW halos.
    Input:
    d = [float; units:Mpc] center to center 3D separation of the subclusters
    m_1 = [float; units:M_sun] mass of subcluster 1 out to r_200
    rho_s_1 = [float; units:M_sun/Mpc^3] NFW scale density of subcluster 1
    r_s_1 = [float; units:Mpc] NFW scale radius of subcluster 1
    r_200_1 = [float; units:Mpc] r_200 of subcluster 1
    m_2 = [float; units:M_sun] mass of subcluster 2 out to r_200
    rho_s_2 = [float; units:M_sun/Mpc^3] NFW scale density of subcluster 2
    r_s_2 = [float; units:Mpc] NFW scale radius of subcluster 2
    r_200_2 = [float; units:Mpc] r_200 of subcluster 2
    N = [int] number of mass elements along one coordinate axis for numerical
        integration approximation
    Output:
    V_total = [float; units:(km/s)^2*M_sun] total potential energy of the two
        subcluster system
    '''
    if d >= r_200_1+r_200_2:
        V_total = -G*m_1*m_2/d
    else:
        # mass element sizes
        dr = r_200_2/N
        dt = numpy.pi/N
        i,j = numpy.meshgrid(numpy.arange(N),numpy.arange(N))
        # distance of 2nd NFW halo mass element from center of 1st NFW halo
        r = numpy.sqrt(((2*i+1)*dr/2*numpy.sin((2*j+1)*dt/2))**2+
                       (d+(2*i+1)*dr/2*numpy.cos((2*j+1)*dt/2))**2)
        #mass elements of 2nd NFW halo
        m = 2*numpy.pi*rho_s_2*r_s_2**3*(numpy.cos(j*dt)-numpy.cos((j+1)*dt))*(1/(1+
(i+1)*dr/r_s_2)-1/(1+i*dr/r_s_2)+numpy.log(((i+1)*dr+r_s_2)/(i*dr+r_s_2)))
        #determine if 2nd halo mass element is inside or outside of 1st halo
        mask_gt = r>=r_200_1
        mask_lt = r<r_200_1
        # potential energy of each mass element (km/s)^2 * M_sun
        # NFW PE
```

```
        V_nfw = numpy.sum(-4*numpy.pi*G*rho_s_1*r_s_1**3/(r[mask_lt]+dr)*(numpy.log(1+
(r[mask_lt]+dr)/r_s_1)-(r[mask_lt]+dr)/(r_s_1+r_200_1))*m[mask_lt])
        # Point mass PE
        V_pt = numpy.sum(-G*m_1*m[mask_gt]/r[mask_gt])
        V_total = V_nfw+V_pt
    return V_total

def NFWprop(M_200,z,c):
    '''
    Determines the NFW halo related properties. Added this for the case of user
    specified concentration.
    Input:
    M_200 = [float; units:M_sun] mass of the halo. Assumes M_200 with
        respect to the critical density at the halo redshift.
    z = [float; unitless] redshift of the halo.
    c = [float; unitless] concentration of the NFW halo.
    '''
    # CONSTANTS
    rho_cr = cosmo.rhoCrit(z)/kginMsun*minMpc**3
    #calculate the r_200 radius
    r_200 = (M_200*3/(4*numpy.pi*200*rho_cr))**(1/3.)
    del_c = 200/3.*c**3/(numpy.log(1+c)-c/(1+c))
    r_s = r_200/c
    rho_s = del_c*rho_cr
    return del_c, r_s, r_200, c, rho_s

def MCengine(N_mc,M1,M2,Z1,Z2,D_proj,prefix,C1=None,C2=None,del_mesh=100,TSM_mesh=200):
    '''
    This is the Monte Carlo engine that draws random parameters from the
    measured distributions and then calculates the kenematic parameters of the
    merger.
    Input:
    N_mc = [int] number of successful Monte Carlo samplings to perform
    M1 = [(float, float) or (1D array of floats); units:M_sun] M_200 of
        subcluster 1.  If (float, float) is input then it is interpreted as
        (mu, sigma) of a Gaussian distribution. If (1D array of floats with
        length > 2) then it is interpreted as a random sample from a parent
        distribution.
    M2 = [(float, float) or (1D array of floats); units:M_sun] M_200 of
        subcluster 2. Must have same format as M1.
    Z1 = [(float, float) or (1D array of floats)] redshift of subcluster 1.
        Similar input format to M1.
    Z2 = [(float, float) or (1D array of floats)] redshift of subcluster 2.
        Similar input format to M1.
    D_proj = [(float, float, float) or (1D array of floats); units:Mpc]
        projected separation of the two subclusters.  If (float, float, float)
        is input then it is interpreted as (d_proj, c1_sigma, c2_sigma), where
        d_proj is the projected distance of the two halos and c1_sigma and
        c2_sigma are the centroid errors of each halo.  If (1D array of floats
        with length > 3) then it is interpreted as a random sample from the
        parent distribution of d_proj.
    prefix = [string] prefix to attach to all output
    C1 = [(float, float) or (1D array of floats); unitless] NFW concentration of
        subcluster 1.  If (float, float) is input then it is interpreted as
        (mu, sigma) of a Gaussian distribution. If (1D array of floats with
        length > 2) then it is interpreted as a random sample from a parent
        distribution. If None then Duffey et al. (2008) scaling relation is used
        to determine the concentration based on M1 value. Must have same format
        as M1.
    C2 = [(float, float) or (1D array of floats); unitless] NFW concentration of
        subcluster 2. Must have same format as M2.
    del_mesh = [int] number of mass elements along one coordinate axis for
        numerical integration approximation of potential energy
    TSM_mesh = [int] number of elements along the separation axis when
        performing the numerical integration appoximation of the TSM
    Output:
    In general the 1D arrays of floats are output for each of the following
    parameters.  The Nth element of each array corresponds to the system
    properties of the Nth viable solution.
```

```
    m_1_out = [units: M_sun] M_200 of halo 1
    m_2_out = [units: M_sun] M_200 of halo 1
    z_1_out = redshift of halo 1
    z_2_out = redshift of halo 2
    alpha_out = [units: degrees] merger axis angle with respect to the plane of
        the sky. 0 corresponds to in the plane of the sky, 90 corresponds to
        along the line-of-sight.
    d_proj_out = [units: Mpc] projected center to center separation of the
        halos in observed state
    d_3d_out = [units: Mpc] 3D center to center separation of the halos in
        observed state
    d_max_out = [units: Mpc] 3D maximum center to center separation of the two
        halo at the apoapsis
    v_rad_obs_out = [units: km/s] relative line-of-sight velocity of the two
        halos at the observed time
    v_3d_obs_out = [units: km/s] relative 3D velocity of the two halos at the
        observed time
    v_3d_col_out = [units: km/s] relative 3D velocity of the two halos at the
        collision time
    TSM_0_out = [units: Gyr] time it took the system to reach the observed state
        from the collision state.  Assuming the system has not reached its
        apoapsis since colliding.
    TSM_1_out = [units: Gyr] time it took the system to reach the observed state
        from the collision state.  Assuming the system has passed throught the
        apoapsis once since colliding.
    T_out = [units: Gyr] period of the system
    prob_out = This is the probability of observing the system with the randomly
       drawn paramers.  This accounts for the fact that it is more likely to
       observe the system near the apoapsis rather than with zero separation,
       due to the system spending more of its time at large separation.  Simply
       an effect of the velocity of the system.
    '''
    # Verify user input
    # Check to make sure mass inputs for each halo are the same
    N_M1 = numpy.size(M1)
    N_M2 = numpy.size(M2)
    if N_M1 != N_M2:
        print 'MCMAC.MCengine: Error, the mass inputs for the two halos must \
            be the same type and size. For example, you cannot mix input \
            format (mu, sigma) for halo1 with (1D array of floats) format \
            for halo 2. Nor can the size of the (1D array of floats) input \
            be different. This is to facilitate correct covariance handeling. \
            Exiting.'
        sys.exit()

    if C1 != None or C2 != None:
        N_C1 = numpy.size(C1)
        N_C2 = numpy.size(C2)
        if N_C1 != N_M1 or N_C2 != N_M2:
            print 'MCMAC.MCengine: Error, the concentration inputs for the two \
            halos must be the same type and size as the mass inputs. For \
            example, you cannot mix input format (mu, sigma) for M1 with \
            (1D array of floats) format for C1, or vise versa. Nor can the size\
            of the (1D array of floats) input be different. This is to \
            facilitate correct covariance handeling. Exiting.'
            sys.exit()
        elif N_M1 > 2:
            print 'MCMAC.MCengine: Assuming that the order and values of the \
            C1 and C2 (1D array of floats) are correlated with the order and \
            values of the M1 and M2 (1D array of floats). This is meant to \
            maintain proper covariance.'

    N_D_proj = numpy.size(D_proj)
    if N_D_proj == N_M1:
        print 'MCMAC.MCengine: Assuming that the order and values of the \
        D_proj (1D array of floats) are correlated with the order and values of\
        the M1 and M2 (1D array of floats). This is meant to maintain proper\
        covariance.'
```

```
    N_Z1 = numpy.size(Z1)
    if N_Z1 == N_D_proj:
        print 'MCMAC.MCengine: Warning. It is currently assumed that the halo\
        redshifts and projected separation estimates are uncorrelated. \
        Covariance will not be handled correctly.'
    if N_Z1 >2 and N_Z1 == N_M1:
        print 'MCMAC,MCengine: Warning. It is currently assumed that the halo \
        redshifts and mass estimates are uncorrelated. Covariance will not be \
        handled correctly.'

    i = 0
    # Create the output arrays
    m_1_out = numpy.zeros(N_mc)
    m_2_out = numpy.zeros(N_mc)
    z_1_out = numpy.zeros(N_mc)
    z_2_out = numpy.zeros(N_mc)
    d_proj_out = numpy.zeros(N_mc)
    v_rad_obs_out = numpy.zeros(N_mc)
    alpha_out = numpy.zeros(N_mc)
    v_3d_obs_out = numpy.zeros(N_mc)
    d_3d_out = numpy.zeros(N_mc)
    v_3d_col_out = numpy.zeros(N_mc)
    d_max_out = numpy.zeros(N_mc)
    TSM_0_out = numpy.zeros(N_mc)
    TSM_1_out = numpy.zeros(N_mc)
    T_out = numpy.zeros(N_mc)
    prob_out = numpy.zeros(N_mc)

    N_d = numpy.size(D_proj)
    t_start = time.time()
    while i < N_mc:
        # Draw random mass and redshift parameters
        m_1, index = randdraw(M1)
        m_2, index = randdraw(M2,index)
        if m_1 <= 0 or m_2 <=0:
            # Discard these unphysical draws, this can happen if a Gaussian
            # distribution is speficied since at least part of the tail will be
            # in the negative mass range.
            continue
        z_1, temp = randdraw(Z1)
        z_2, temp = randdraw(Z2)

        # Draw random projected separation
        if N_d == 3: #then assumed (D_proj, c1_sigma, c2_sigma)
            #draw del_r1, del_r2, phi1 and phi2, where phi is the
            del_r_1 = D_proj[1] * numpy.random.randn()
            del_r_2 = D_proj[2] * numpy.random.randn()
            phi_1 = numpy.random.rand()*2*numpy.pi
            phi_2 = numpy.random.rand()*2*numpy.pi
            #calculate the projected separation based on drawn centers
            d_proj = numpy.sqrt((D_proj[0]-del_r_1*numpy.cos(phi_1)+del_r_2*numpy.cos(phi_2))**2+
(del_r_1*numpy.sin(phi_1)+del_r_2*numpy.sin(phi_2))**2)
        elif N_d > 3: #then assume distribution array format
            if N_D_proj == N_M1:
                # assume D_proj array values and order correlated with M1 and M2
                # array values and order.
                d_proj, index = randdraw(D_proj,index)
            else:
                # assume D_proj is uncorrelated with M1 and M2
                d_proj = D_proj[numpy.random.randint(N_d)]
        elif N_d < 3:
            print 'MCMAC.MCengine: Error, D_proj parameter input array is not a valid size,
exiting'
            sys.exit()
        d_proj = abs(d_proj)

        # Define NFW halo properties
        if C1 == None:
            # Then use scaling relation to determine concentration
```

```
            del_c_1, r_s_1, r_200_1, c_1, rho_s_1 = profiles.nfwparam_extended(m_1/1e14,z_1)
        else:
            # Then user specified concentration
            # Draw random concentration parameter
            c_1, index = randdraw(C1,index)
            del_c_1, r_s_1, r_200_1, c_1, rho_s_1 = NFWprop(m_1,z_1,c_1)
        if C2 == None:
            del_c_2, r_s_2, r_200_2, c_2, rho_s_2 = profiles.nfwparam_extended(m_2/1e14,z_2)
        else:
            c_2, index = randdraw(C2,index)
            del_c_2, r_s_2, r_200_2, c_2, rho_s_2 = NFWprop(m_2,z_2,c_2)

        # Reduced mass
        mu = m_1*m_2/(m_1+m_2)

        # Calculate potential energy at time of collision
        PE_col = PEnfwnfw(0,m_1,rho_s_1,r_s_1,r_200_1,m_2,rho_s_2,r_s_2,r_200_2,N=del_mesh)

        # Calculate maximum 3D free-fall velocity
        # Note that this assumes that the clusters begin with infinite
        # separation and their observed mass. It also guarntees that all
        # solutions will be for a bound system.
        v_3d_max = numpy.sqrt(-2/mu*PE_col)

        # Calculate observed radial velocity
        v_rad_obs = vrad(z_1,z_2)

        # Draw a random alpha, merger axis angles with respect to the sky
        ALPHA = numpy.random.rand()*numpy.pi/2
        #Since not all alpha are equally likely.
        alpha = (1-numpy.cos(ALPHA))*numpy.pi/2

        # Calculate the 3D velocity at observed time
        v_3d_obs = v_rad_obs/numpy.sin(alpha)

        if v_3d_obs > v_3d_max:
            # then the randomly chosen alpha is not valid
            continue

        # Calculate the 3D separation
        d_3d = d_proj/numpy.cos(alpha)

        # Calculate the potential energy at observed time
        PE_obs = PEnfwnfw(d_3d,m_1,rho_s_1,r_s_1,r_200_1,m_2,rho_s_2,r_s_2,r_200_2,N=del_mesh)

        # Calculate the 3D velocity at collision time
        v_3d_col = numpy.sqrt(v_3d_obs**2+2/mu*(PE_obs-PE_col))

        if v_3d_col > v_3d_max:
            # then the randomly chosen alpha is not valid
            continue

        # Total Energy
        E = PE_obs+mu/2*v_3d_obs**2

        # Calculate PE from d = 0 to r_200_1+r_200_2
        del_TSM_mesh = (r_200_1+r_200_2)/(TSM_mesh-1)
        d = numpy.arange(0.00001,(r_200_1+r_200_2)+del_TSM_mesh,del_TSM_mesh)
        PE_array = numpy.zeros(TSM_mesh)
        for j in numpy.arange(TSM_mesh):
            PE_array[j] =
PEnfwnfw(d[j],m_1,rho_s_1,r_s_1,r_200_1,m_2,rho_s_2,r_s_2,r_200_2,N=del_mesh)

        # Calculate d_max
        if E >= -G*m_1*m_2/(r_200_1+r_200_2):
            # then d_max > r_200_1+r_200_2
            d_max = -G*m_1*m_2/E
        else:
            # d_max <= r_200_1+r_200_2
```

```
            ## find closet d value less than d_max
            #idx=(numpy.abs(PE_array-E)).argmin()
            #d_max = d[idx]
            # determine d_max, ensuring that E is always > PE
            mask_tmp = E >= PE_array
            d_max = d[mask_tmp][-1]

        # Calculate TSM_0
        if d_3d >= r_200_1+r_200_2:
            # then halos no longer overlap
            # calculate the time it takes to go from d=0 to r_200_1+r_200_2
            TSM_0a = numpy.sum(del_TSM_mesh/numpy.sqrt(2/mu*(E-PE_array))*kminMpc/sinGyr)
            # calculate th time it takes to go from d = r_200_1+r_200_2 tp d_3d
            TSM_0b = TSMptpt(m_1,m_2,r_200_1,r_200_2,d_3d,E)
            TSM_0 = TSM_0a+TSM_0b
        else:
            # then d_3d < r_200_1+r_200_2, halos always overlap
            # calculate the time it takes to go from d=0 to d_3d
            mask = d <= d_3d
            TSM_0 = numpy.sum(del_TSM_mesh/numpy.sqrt(2/mu*(E-PE_array[mask]))*kminMpc/sinGyr)

        # Check that TSM_0 < Age of Universe at (z_1+z_2)/2
        age = cosmo.age((z_1+z_2)/2)
        if TSM_0 > age:
            # unlikely that this system could occur
            continue

        # Calculate period
        if E >= -G*m_1*m_2/(r_200_1+r_200_2):
            # then d_max > r_200_1+r_200_2
            # calculate th time it takes to go from d = r_200_1+r_200_2 to d_max
            if d_3d < r_200_1+r_200_2:
                #then TSM_0a has not been previously defined
                TSM_0a = numpy.sum(del_TSM_mesh/numpy.sqrt(2/mu*(E-PE_array))*kminMpc/sinGyr)
            TSM_0b = TSMptpt(m_1,m_2,r_200_1,r_200_2,d_max,E)
            T = 2*(TSM_0a+TSM_0b)
        else:
            # then d_max < r_200_1+r_200_2
            # calculate the time it takes to go from d=0 to d_max
            mask = d <= d_max
            T = 2*numpy.sum(del_TSM_mesh/numpy.sqrt(2/mu*(E-PE_array[mask]))*kminMpc/sinGyr)

        # Calculate probability of d_3d
        prob = TSM_0/(T/2)

        # Calculate TSM_1
        TSM_1 = T-TSM_0

        if TSM_0 < 0:
            print 'TSM < 0 encountered'

        # Write calculated merger parameters
        m_1_out[i] = m_1
        m_2_out[i] = m_2
        z_1_out[i] = z_1
        z_2_out[i] = z_2
        d_proj_out[i] = d_proj
        v_rad_obs_out[i] = v_rad_obs
        alpha_out[i] = alpha*180/numpy.pi
        v_3d_obs_out[i] = v_3d_obs
        d_3d_out[i] = d_3d
        v_3d_col_out[i] = v_3d_col
        d_max_out[i] = d_max
        TSM_0_out[i] = TSM_0
        TSM_1_out[i] = TSM_1
        T_out[i] = T
        prob_out[i] = prob

        i+=1
```

```
        # Estimate calculation time
        if i== 10 or i == 100 or i%1000 == 0:
            del_t = time.time()-t_start
            t_total = N_mc*del_t/i
            eta = (t_total-del_t)/60
            print 'Completed Monte Carlo iteration {0} of {1}.'.format(i,N_mc)
            print '~{0:0.0f} minutes remaining'.format(eta)

    # Pickle the results of the MC analysis
    filename = prefix+'_m_1.pickle'
    F = open(filename,'w')
    pickle.dump(m_1_out,F)
    F.close()
    filename = prefix+'_m_2.pickle'
    F = open(filename,'w')
    pickle.dump(m_2_out,F)
    F.close()
    filename = prefix+'_z_1.pickle'
    F = open(filename,'w')
    pickle.dump(z_1_out,F)
    F.close()
    filename = prefix+'_z_2.pickle'
    F = open(filename,'w')
    pickle.dump(z_2_out,F)
    F.close()
    filename = prefix+'_d_proj.pickle'
    F = open(filename,'w')
    pickle.dump(d_proj_out,F)
    F.close()
    filename = prefix+'_v_rad_obs.pickle'
    F = open(filename,'w')
    pickle.dump(v_rad_obs_out,F)
    F.close()
    filename = prefix+'_alpha.pickle'
    F = open(filename,'w')
    pickle.dump(alpha_out,F)
    F.close()
    filename = prefix+'_v_3d_obs.pickle'
    F = open(filename,'w')
    pickle.dump(v_3d_obs_out,F)
    F.close()
    filename = prefix+'_d_3d.pickle'
    F = open(filename,'w')
    pickle.dump(d_3d_out,F)
    F.close()
    filename = prefix+'_v_3d_col.pickle'
    F = open(filename,'w')
    pickle.dump(v_3d_col_out,F)
    F.close()
    filename = prefix+'_d_max.pickle'
    F = open(filename,'w')
    pickle.dump(d_max_out,F)
    F.close()
    filename = prefix+'_TSM_0.pickle'
    F = open(filename,'w')
    pickle.dump(TSM_0_out,F)
    F.close()
    filename = prefix+'_TSM_1.pickle'
    F = open(filename,'w')
    pickle.dump(TSM_1_out,F)
    F.close()
    filename = prefix+'_T.pickle'
    F = open(filename,'w')
    pickle.dump(T_out,F)
    F.close()
    filename = prefix+'_prob.pickle'
    F = open(filename,'w')
    pickle.dump(prob_out,F)
```

```
    F.close()


    return
m_1_out,m_2_out,z_1_out,z_2_out,d_proj_out,v_rad_obs_out,alpha_out,v_3d_obs_out,d_3d_out,v_3d_col_
out,d_max_out,TSM_0_out,TSM_1_out,T_out,prob_out
```