4.4 Galaxy Collisions

In their eloquent article from the 1980s, Schroeder and Comin [14] came up with a very simple, yet amazingly useful model for interacting disk galaxies. They proposed to treat a disk galaxy as gravitating point mass surrounded by a disk of test particles representing stars, assuming that all the mass of a galaxy is concentrated in the center and gravitational interactions between stars are negligible. There is no dark matter, no interstellar gas and dust, and stars only experience the gravity of the central mass (or central masses when dealing with a system of two galaxies). Although this is a very crude picture, it is able to reproduce some basic properties seen in interacting galaxies. Of course, it is far from being competitive to a full N-body simulation, which will be the subject of the next section. Since the computation is so much cheaper, it nevertheless serves its purpose as a pedagogical model that can be run in a matter of seconds on any PC or laptop.

In the following, we will reimplement the model of Schroeder and Comin with some extra features in Python.[22] Our version is contained in the module `galcol`, which is part of the zip archive for this chapter. As a first step, we need to initialize a galactic disk. The function `galcol.init_disk()` listed below accepts a dictionary of basic parameters, which can be generated with `galcol.parameters()`:

```
1  # excerpt from galcol.py
2  def init_disk(galaxy, time_step=0.1*unit.Myr):
3      '''
4      initializes galaxy by setting stars in random positions
5      and Keplerian velocities half a time step in advance
6      (Leapfrog scheme)
7
8      args: dictionary of galaxy parameters,
9            numerical time step
10     '''
11
12     # width of a ring
13     dr = (1 - galaxy['softening'])*galaxy['radius']/ \
14          galaxy['N_rings']
15     N_stars_per_ring = int(galaxy['N_stars']/galaxy['N_rings'])
16
17     # rotation angle and axis
18     norm = np.sqrt(galaxy['normal'][0]**2 +
19                    galaxy['normal'][1]**2 +
20                    galaxy['normal'][2]**2)
21     cos_theta = galaxy['normal'][2]/norm
22     sin_theta = np.sqrt(1-cos_theta**2)
23     u = np.cross([0,0,1], galaxy['normal']/norm)
24     norm = np.sqrt(u[0]**2 + u[1]**2 + u[2]**2)
25
26     if norm > 0:
```

[22] The original program GC3D (Gallactic Collisions in 3D) was written in BASIC and later adapted in [4].

```
        u /= norm # unit vector

        # rotation matrix for coordinate transformation
        # from galactic plane to observer's frame
        rotation = \
            [[u[0]*u[0]*(1-cos_theta) + cos_theta,
              u[0]*u[1]*(1-cos_theta) - u[2]*sin_theta,
              u[0]*u[2]*(1-cos_theta) + u[1]*sin_theta],
             [u[1]*u[0]*(1-cos_theta) + u[2]*sin_theta,
              u[1]*u[1]*(1-cos_theta) + cos_theta,
              u[1]*u[2]*(1-cos_theta) - u[0]*sin_theta],
             [u[2]*u[0]*(1-cos_theta) - u[1]*sin_theta,
              u[2]*u[1]*(1-cos_theta) + u[0]*sin_theta,
              u[2]*u[2]*(1-cos_theta) + cos_theta]]

        # print angels defining orientation of galaxy
        phi = np.arctan2(galaxy['normal'][1],
                         galaxy['normal'][0])
        theta = np.arccos(cos_theta)
        print("Plane normal: ",
              "phi = {:.1f} deg, theta = {:.1f} deg".\
              format(np.degrees(phi), np.degrees(theta)))

    else:
        rotation = np.identity(3)

    galaxy['stars_pos'] = np.array([])
    galaxy['stars_vel'] = np.array([])

    # begin with innermost radius given by softening factor
    R = galaxy['softening']*galaxy['radius']
    for n in range(galaxy['N_rings']):

        # radial and angular coordinates
        # in center-of-mass frame
        r_star = R + \
            dr * np.random.random_sample(size=N_stars_per_ring)
        phi_star = 2*np.pi * \
            np.random.random_sample(size=N_stars_per_ring)

        # Cartesian coordinates in observer's frame
        vec_r = np.dot(rotation,
                       r_star*[np.cos(phi_star),
                               np.sin(phi_star),
                               np.zeros(N_stars_per_ring)])
        x = galaxy['center_pos'][0] + vec_r[0]
        y = galaxy['center_pos'][1] + vec_r[1]
        z = galaxy['center_pos'][2] + vec_r[2]

        # orbital periods and angular displacements
        # over one timestep
```

```
78          T_star = 2*np.pi * ((G*galaxy['mass'])**(-1/2) * \
79                   r_star**(3/2)).to(unit.s)
80          delta_phi = 2*np.pi * time_step.to(unit.s).value / \
81                      T_star.value
82
83          # velocity components in observer's frame
84          # one half of a step in advance (Leapfrog scheme)
85          vec_v = np.dot(rotation,
86              (r_star.to(unit.km)/time_step.to(unit.s)) * \
87               [(np.cos(phi_star) - np.cos(phi_star-delta_phi)),
88                (np.sin(phi_star) - np.sin(phi_star-delta_phi)),
89                np.zeros(N_stars_per_ring)])
90          v_x = galaxy['center_vel'][0] + vec_v[0]
91          v_y = galaxy['center_vel'][1] + vec_v[1]
92          v_z = galaxy['center_vel'][2] + vec_v[2]
93
94          if galaxy['stars_pos'].size == 0:
95              galaxy['stars_pos'] = np.array([x,y,z])
96              galaxy['stars_vel'] = np.array([v_x,v_y,v_z])
97          else:
98              galaxy['stars_pos'] = \
99                  np.append(galaxy['stars_pos'],
100                           np.array([x,y,z]), axis=1)
101             galaxy['stars_vel'] = \
102                 np.append(galaxy['stars_vel'],
103                           np.array([v_x,v_y,v_z]), axis=1)
104
105         R += dr
106
107     # units get lost through np.array
108     galaxy['stars_pos'] *= unit.kpc
109     galaxy['stars_vel'] *= unit.km/unit.s
110
111     # typical velocity scale defined by Kepler velocity
112     # at one half of the disk radius
113     galaxy['vel_scale'] = np.sqrt(G*galaxy['mass']/(0.5*R)).\
114                                   to(unit.km/unit.s)
```

We follow [14] in subdividing the disk into a given number of rings, `galaxy['N_rings']`. The radial width of each ring `dr` is computed from the outer disk radius `galaxy['radius']` (lines 13–14). The disk also has an inner edge which is specified as fraction of the disk radius. Since this fraction is used to avoid a boundless potential near the center when computing the orbits of test particles, it is called softening factor. While stars are set at constant angular separation in each ring in the original model, we place the stars at random positions. This is done in code lines 62 to 65, where the function `random_sample()` from NumPy's `random` module is used to define the star's radial and angular coordinates within each ring. The sample returned by `random_sample()` is drawn from a uniform distribution in the interval [0, 1] and has to be scaled and shifted to obtain values in the desired range. The number of random values is of course given by the number of stars per ring defined

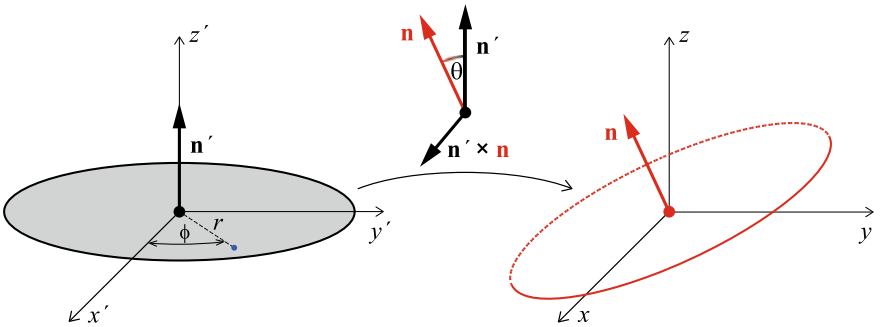

Fig. 4.12 Schematic view of a disk in a coordinate frame aligned with the disk plane and its normal (left) and in another frame with arbitrary orientation relative to the disk (right). The frames are related by a rotation by the angle θ around the axis given by $\mathbf{n}' \times \mathbf{n}$ (middle)

in line 15. The resulting coordinates are defined in a polar coordinate system in the disk plane.

The next problem is transferring the polar coordinates of the stars in the disk to a three-dimensional Cartesian coordinate system (we will refer to this coordinate system as the observer's frame) in which the disk can have arbitrary orientation. The orientation is defined by the normal vector $\mathbf{n} = (n_x, n_y, n_z)$ of the galactic plane, which is defined as a 3-tuple `galaxy['normal']` in the galaxy's dictionary. This requires several steps. First we need to convert the polar coordinates (r, ϕ) into Cartesian coordinates (x', y', z'), where the x' and y' axes are aligned with the disk plane and the z' axis points in the direction perpendicular to the plane (see Fig. 4.12).

$$x' = r\cos\phi, \tag{4.60}$$
$$y' = r\sin\phi, \tag{4.61}$$
$$z' = 0 \tag{4.62}$$

The disk's normal is given by $\mathbf{n}' = (0, 0, 1)$ in this coordinate system. To align the coordinate axes with the observer's frame, the normal direction has to be rotated by an angle θ given by

$$\cos\theta = \mathbf{n}' \cdot \mathbf{n} = n_z \ , \tag{4.63}$$

assuming that $|\mathbf{n}| = 1$. To ensure that $\mathbf{n}$ is a unit vector, the vector `galaxy['normal']` is divided by its length when defining the variable `cos_theta` in line 21. The rotation axis is then given by the normalized cross product

$$\mathbf{u} = \frac{\mathbf{n}' \times \mathbf{n}}{|\mathbf{n}' \times \mathbf{n}|} \ , \tag{4.64}$$

as shown in the middle of Fig. 4.12. The cross product can be evaluated by applying the function `np.cross()` (see line 23). Having defined the rotation angle θ and

axis $\mathbf{u}$, we obtain the coordinates of a star in the observer's frame, $\mathbf{r} = (x, y, z)^{\mathrm{T}}$,[23] by multiplying $\mathbf{r}' = (x', y', z')^{\mathrm{T}}$ with the rotation matrix R, i.e.

$$\mathbf{r} = \mathsf{R} \cdot \mathbf{r}' \,, \tag{4.65}$$

where

$$\mathsf{R} = \begin{bmatrix} u_x^2(1-\cos\theta) + \cos\theta & u_x u_y(1-\cos\theta) - u_z \sin\theta & u_x u_z(1-\cos\theta) + u_y \sin\theta \\ u_y u_x(1-\cos\theta) + u_z \sin\theta & u_y^2(1-\cos\theta) + \cos\theta & u_y u_z(1-\cos\theta) - u_x \sin\theta \\ u_z u_x(1-\cos\theta) - u_y \sin\theta & u_z u_y(1-\cos\theta) + u_x \sin\theta & u_z^2(1-\cos\theta) + \cos\theta \end{bmatrix}$$

This expression is is know as Rodrigues' rotation formula. The rotation matrix is defined in lines 31–40 as two-dimensional NumPy array of shape `(3,3)` provided that the norm of the cross $\mathbf{n}' \times \mathbf{n}$ is positive (line 26) and the right-hand-side of Eq. (4.64) is mathematically defined. If the normal vectors $\mathbf{n}'$ and $\mathbf{n}$ are aligned, the rotation angle is 0 and the rotation matrix is set equal to the identity matrix (line 51).

Equation (4.65) with x', y', and z' substituted by expressions (4.60)–(4.62) is coded in lines 68–74, using `np.dot()` for the product of a matrix and a vector. Since we work with NumPy arrays of length given by the number of stars per ring, this is done inside the **`for`** loop through all rings beginning in line 58. The advantage of using `dot()` is that it can be applied not only to a single vector but also to an array of vectors. (You might find it instructive to figure out the shapes of the variables in lines 68–71 for a particular example). Finally, the x, y, and z coordinates are shifted by the position of the disk center in the observer's frame (lines 72–74).

In the block of code starting at line 85, the orbital velocities of the stars are computed through similar transformations as the positions. Since we are going to solve the equations of motion using a Leapfrog scheme, we need to initialize the velocity of each star by its Keplerian velocity one half of a time step earlier than the initial positions. In the Leapfrog scheme, this is the mean velocity $\mathbf{v}'(t_{-1/2})$ over the time interval $[t_0 - \Delta t, t_0]$:

$$\mathbf{v}'(t_{-1/2}) \simeq \frac{\mathbf{r}'(t_0) - \mathbf{r}'(t_0 - \Delta t)}{\Delta t} \tag{4.66}$$

For an orbital period T given by Kepler's third law (lines 78–79; see also Sect. 2.2), the angular shift corresponding to Δt is given by (lines 80–81)

$$\Delta\phi = 2\pi \, \frac{\Delta t}{T} \,. \tag{4.67}$$

Thus, the velocity components in the plane of the disk can be expressed as

[23] The superscript T indicates that $\mathbf{r}$ is a column vector, which is important in the context of matrix multiplication.

$$v'_x = r\,[\cos\phi - \cos(\phi - \Delta\phi)]\,/\Delta t, \tag{4.68}$$
$$v'_y = r\,[\sin\phi - \sin(\phi - \Delta\phi)]\,/\Delta t, \tag{4.69}$$
$$v'_z = 0 \tag{4.70}$$

Applying the frame rotation, $\mathbf{v} = \mathsf{R} \cdot \mathbf{v}'$, and adding the translation velocity of the disk center yields the velocities in the observer's frame.

In lines 94–103, the computed stellar positions and orbital velocities are accumulated in the arrays `galaxy['stars_pos']` and `galaxy['stars_vel']`, respectively. Once all rings are filled and the loop terminates, the resulting arrays have a shape corresponding to three spatial dimensions times the number of stars in the disk. Finally, Astropy units of kpc and km/s are attached to positions and velocities, respectively. This is necessary because NumPy's `array()` function strips any units from its argument (the reason is explained in Sect. 2.1.3). To compute numerical values that are consistent with the chosen units, radial distance is converted to km and time to s in line 86. It is also noteworthy that we make an exception of our rule of explicitly returning the output of a function here. The position and velocity data are added as new items to the dictionary `galaxy`, which is an argument of `galcol.init_disk()`. Such changes persist outside of a function call.[24] As a result, there is no need to return a complete copy of `galaxy` at the end of the function body.

After having defined initial data, the functions `galcol.evolve_disk()` and `evolve_two_disks()` can be applied to compute the time evolution of a single disk or a pair of disks, respectively. Studying a single disk is left as an exercise (see Exercise 4.11). The definition of `evolve_two_disks()` for the simulation of collisions of galaxies is listed in the following.

```
1  # excerpt from galcol.py
2  def evolve_two_disks(primary, secondary,
3                       time_step=0.1*unit.Myr,
4                       N_steps=1000, N_snapshots=100):
5      '''
6      evolves primary and secondary disk
7      using Leapfrog integration
8
9      args: dictionaries of primary and secondary galaxy,
10           numerical timestep, number of timesteps,
11           number of snapshots
12
13     returns: array of snapshot times,
14              array of snapshots
15              (spatial coordinates of centers and stars)
16     '''
17     dt = time_step.to(unit.s).value
18
19     r_min1 = primary['softening']*primary['radius'].\
```

[24] This is an example for Python's call by object reference mentioned in Sect. 3.1.2.

```
            to(unit.m).value
    r_min2 = secondary['softening']*secondary['radius'].\
            to(unit.m).value

    N1, N2 = primary['N_stars'], secondary['N_stars']

    # mass, position and velocity of primary galactic center
    M1 = primary['mass'].to(unit.kg).value
    X1, Y1, Z1 = primary['center_pos'].to(unit.m).value
    V1_x, V1_y, V1_z = primary['center_vel'].\
                       to(unit.m/unit.s).value

    # mass, position and velocity of secondary galactic center
    M2 = secondary['mass'].to(unit.kg).value
    X2, Y2, Z2 = secondary['center_pos'].to(unit.m).value
    V2_x, V2_y, V2_z = secondary['center_vel'].\
                       to(unit.m/unit.s).value

    # stellar coordinates of primary
    x = primary['stars_pos'][0].to(unit.m).value
    y = primary['stars_pos'][1].to(unit.m).value
    z = primary['stars_pos'][2].to(unit.m).value

    # stellar coordinates of secondary
    x = np.append(x, secondary['stars_pos'][0].\
                     to(unit.m).value)
    y = np.append(y, secondary['stars_pos'][1].\
                     to(unit.m).value)
    z = np.append(z, secondary['stars_pos'][2].\
                     to(unit.m).value)

    # stellar velocities of primary
    v_x = primary['stars_vel'][0].to(unit.m/unit.s).value
    v_y = primary['stars_vel'][1].to(unit.m/unit.s).value
    v_z = primary['stars_vel'][2].to(unit.m/unit.s).value

    # stellar velocities of secondary
    v_x = np.append(v_x, secondary['stars_vel'][0].\
                         to(unit.m/unit.s).value)
    v_y = np.append(v_y, secondary['stars_vel'][1].\
                         to(unit.m/unit.s).value)
    v_z = np.append(v_z, secondary['stars_vel'][2].\
                         to(unit.m/unit.s).value)

    # array to store snapshots of all positions
    # (centers and stars)
    snapshots = np.zeros(shape=(N_snapshots+1,3,N1+N2+2))
    snapshots[0] = [np.append([X1,X2], x),
                    np.append([Y1,Y2], y),
                    np.append([Z1,Z2], z)]

```

```
    # number of steps per snapshot
    div = max(int(N_steps/N_snapshots), 1)

    print("Solving equations of motion for two galaxies",
          "(Leapfrog integration)")

    for n in range(1,N_steps+1):

        # radial distances from centers with softening
        r1 = np.maximum(np.sqrt((X1 - x)**2 +
                                (Y1 - y)**2 +
                                (Z1 - z)**2), r_min1)
        r2 = np.maximum(np.sqrt((X2 - x)**2 +
                                (Y2 - y)**2 +
                                (Z2 - z)**2), r_min2)

        # update velocities of stars
        # (acceleration due to gravity of centers)
        v_x += G.value * (M1*(X1 - x)/r1**3 +
                          M2*(X2 - x)/r2**3) * dt
        v_y += G.value * (M1*(Y1 - y)/r1**3 +
                          M2*(Y2 - y)/r2**3) * dt
        v_z += G.value * (M1*(Z1 - z)/r1**3 +
                          M2*(Z2 - z)/r2**3) * dt

        # update positions of stars
        x += v_x*dt
        y += v_y*dt
        z += v_z*dt

        # distance between centers
        D_sqr_min = (r_min1+r_min2)**2
        D_cubed = \
            (max((X1 - X2)**2 + (Y1 - Y2)**2 + (Z1 - Z2)**2,
                 D_sqr_min))**(3/2)

        # gravitational acceleration of primary center
        A1_x = G.value*M2*(X2 - X1)/D_cubed
        A1_y = G.value*M2*(Y2 - Y1)/D_cubed
        A1_z = G.value*M2*(Z2 - Z1)/D_cubed

        # update velocities of centers
        # (constant center-of-mass velocity)
        V1_x += A1_x*dt; V2_x -= (M1/M2)*A1_x*dt
        V1_y += A1_y*dt; V2_y -= (M1/M2)*A1_y*dt
        V1_z += A1_z*dt; V2_z -= (M1/M2)*A1_z*dt

        # update positions of centers
        X1 += V1_x*dt; X2 += V2_x*dt
        Y1 += V1_y*dt; Y2 += V2_y*dt
        Z1 += V1_z*dt; Z2 += V2_z*dt
```

```
122
123         if n % div == 0:
124             i = int(n/div)
125             snapshots[i] = [np.append([X1,X2], x),
126                             np.append([Y1,Y2], y),
127                             np.append([Z1,Z2], z)]
128
129         # fraction of computation done
130         print("\r{:3d} %".format(int(100*n/N_steps)), end="")
131
132     time = np.linspace(0*time_step, N_steps*time_step,
133                        N_snapshots+1, endpoint=True)
134     print(" (stopped at t = {:.1f})".format(time[-1]))
135
136     snapshots *= unit.m
137
138     return time, snapshots.to(unit.kpc)
```

Parameters and initial data of the two galaxies are defined by the arguments `primary` and `secondary`, which have to be dictionaries prepared by `galcol.init_disk()`. For the numerical solver in the **`for`** loop through all timesteps (the total number of steps is specified by the optional argument `N_steps`), parameters and data from the dictionaries are converted to simple float values in SI units (lines 17–62). For example, the coordinates `X1`, `Y1`, and `Z1` of the center of the primary galaxy in units of meters are initialized in line 28. The conversion from Astropy objects to numbers avoids some overhead in the implementation and improves efficiency of the computation, while we have the full flexibility of using arbitrary units in the input and output of the function.

The positions of stars in both disks are joined into arrays `x`, `y`, and `z` via `np.append()` in lines 39–49. As a result, the length of the three coordinate arrays equals the total number of stars in the primary and secondary disks. This allows us to apply operations at once to all stars. The coordinates are updated for each time step in lines 97–99, where the velocity components are computed from the accelerations in the gravitational field of the two central masses M_1 and M_2 (lines 89–94):

$$\mathbf{x}(t_{n+1}) = \mathbf{x}(t_n) + \mathbf{v}(t_{n+1/2})\Delta t\,, \tag{4.71}$$

$$\mathbf{v}(t_{n+1/2}) = \mathbf{v}(t_{n-1/2}) + \mathbf{a}(t_n)\Delta t\,, \tag{4.72}$$

where

$$\mathbf{a}(t_n) = \frac{GM_1}{r_1^3(t_n)}\left[\mathbf{X}_1(t_n) - \mathbf{x}(t_n)\right] + \frac{GM_2}{r_2^3(t_n)}\left[\mathbf{X}_2(t_n) - \mathbf{x}(t_n)\right] \tag{4.73}$$

is the gravitational acceleration of a test particle (star) at position $\mathbf{x}(t_n)$ at time $t_n = t_0 + n\Delta t$. The distances $r_{1,2}(t_n)$ from the two central masses are given by (for brevity, time dependence is not explicitly written here):

$$r_{1,2} = \sqrt{(X_{1,2} - x)^2 + (Y_{1,2} - y)^2 + (Y_{1,2} - y)^2}\,. \tag{4.74}$$

However, in the corresponding code lines 80–85 distances are given by the above expression only if the resulting distances are greater than some minimal distances `r_min1` and `r_min2`. NumPy's `maximum()` function compares values element-wise. In this case, it compares each element of the array `r1` to `r_min1` and returns the larger value (and similarly for `r2`). The variables `r_min1` and `r_min2` are defined in terms of the disk's softening factors in lines 19 to 22. This means that in the close vicinity of the gravitational centers, the potential is limited to the potential at the minimal distance. Otherwise stellar velocities might become arbitrarily high, resulting in large errors.

Of course, not only the stars move under the action of gravity, but also the two central masses M_1 and M_2. Since the stars are treated as test masses (i.e. the gravity of the stars is neglected), it is actually a two-body problem that needs to be solved for the central masses. We leave it as an exercise to study the implementation in lines 101–121 (see appendix of [4] for a detailed description). The expressions for the velocity updates of the secondary in lines 114–116 follow from momentum conservation.

The data for the positions of the centers and stars are stored in `snapshots`, a three-dimensional array that is initialized with `np.zeros()` in line 66. Since we want to record the evolution of the system, we need to store a sufficient number of snapshots to produce, for instance, an animation of the two disks. The most obvious choice would be to store the data for all timesteps. However, this would result in a very large array consuming a lot of memory. Moreover, the production of animations becomes very time consuming if the total number of frames is too large. The number of snapshots can be controlled with the optional argument `N_snapshots`. The shape of the array `snapshots` specified in the call of `np.zeros()` is the number of snapshots plus one (for the initial data) times the number of spatial dimensions (three) times the total number of stars plus two (for the two centers). The initial positions of centers and stars defines the first snapshot `snapshots[0]`, which is a two-dimensional subarray (lines 67–69). In principle, we could gradually extend `snapshots` by applying `np.append()` for each subsquent timestep, but this involves copying large amounts of data in memory. As a consequence, the program would slow down considerably with growing size of `snapshots`. For this reason, it is advantageous to define large arrays with their final size (you can check the size in the examples discussed below) and to successively set all elements in the aftermath of filling them with zeros. For the default values defined in line 4, one snapshot will be produced after every cycle of 100 timesteps (the number of steps per snapshot is assigned to the variable `div` in line 72). Consequently, we need to check if the loop counter `n` (the current number of timesteps in the loop body) is a multiple of 10. This is equivalent to a zero remainder of division of `n` by 10. In Python, the remainder is obtained with `%` operator. Whenever this condition is satisfied (line 123), the position data are assigned to the snapshot with index given by `n/div` (lines 124–127). Remember that an array index must be an integer. Since Python performs divisions in floating point arithmetic, we need to apply the conversion function **`int`**`()` when calculating the snapshot index. After the loop over all timesteps has finished, `snapshots` is multiplied by `unit.m` and converted to kpc before it is returned. (Do you see why can we not just multiply by `unit.kpc`?)

The function also returns an array `time` with the instants of time corresponding to the snapshots, which can be used to label visualizations of the snapshots.

Let us do an example:

```
import galcol
import astropy.units as unit

galaxies = {
    'intruder' : galcol.parameters(
        # mass in solar masses
        1e10,
        # disk radius in kpc
        5,
        # coordinates (x,y,z) of initial position in kpc
        (25,-25,-5),
        # x-, y-, z-components of initial velocity in km/s
        (-75,75,0),
        # normal to galactic plane (disk is in xy-plane)
        (0,0,1),
        # number of rings (each ring will be randomly
        # populated with 1000/5 = 200 stars)
        5,
        # total number of stars
        1000,
        # softening factor defines inner edge of disk
        0.025),
    'target' : galcol.parameters(
        5e10, 10, (-5,5,1), (15,-15,0), (1,-1,2**0.5),
        10, 4000, 0.025),
}
```

First, we need to import the modules `galcol` and `astropy.units`. Then an intruder and a target galaxy are defined as items in the dictionary named `galaxies`. To help you keep an overview of the parameters, comments are inserted in the argument list of `galcol.parameters()` for the intruder (to see the definition of the function `parameters()`, open the file `galcol.py` with an editor). Compared to the target, the intruder has a five times smaller mass and is also smaller in size. It approaches the intruder with a relative velocity of 128 km/s from an initial distance of 30 kpc under an angle of 45° in the xy-plane and a separation of 6 kpc in transversal direction. The initial positions and velocity vectors of the two disks are chosen such that their center of mass resides at the origin of the coordinate system (since we are dealing with a two-body problem with central forces, the center of mass is stationary).

Test particles are produced by invoking `galcol.disk_init()` for both disks:

```
165 galcol.init_disk(galaxies['intruder'])
166 galcol.init_disk(galaxies['target'])
```

If you output the contents of `galaxies['intruder']` after the call above, you will find arrays containing the particle's initial positions and velocities under the keywords `'stars_pos'` and `'stars_vel'`, respectively (with different numbers produced by your random number generator):

```
{'mass': <Quantity 1.e+10 solMass>,
 'radius': <Quantity 5. kpc>,
 'center_pos': <Quantity [ 25., -25.,  -5.] kpc>,
 'center_vel': <Quantity [-75.,  75.,   0.] km / s>,
 'normal': (0, 0, 1),
 'N_rings': 5,
 'N_stars': 1000,
 'softening': 0.025,
 'stars_pos': <Quantity [[ 25.27909275,  24.83823236,  25.05526353, ...,  26.70027151,
             28.50078551,  22.24242959],
            [-25.164745  , -24.60920606, -24.43662745, ..., -21.34249645,
             -22.12864831, -20.94120214],
            [ -5.        ,  -5.        ,  -5.        , ...,  -5.        ,
              -5.        ,  -5.        ]] kpc>,
 'stars_vel': <Quantity [[ 127.78537167, -374.04767989, -348.54051012, ...,
             -168.58250706, -136.72561385, -152.49071953],
            [ 377.39033859,  -35.49029092,  108.72755079, ...,
              118.65298369,  150.42575762,   22.46279061],
            [   0.        ,    0.        ,    0.        , ...,
                0.        ,    0.        ,    0.        ]] km / s>,
 'vel_scale': <Quantity 131.16275798 km / s>}
```

The next step is to compute the time evolution of the combined system of intruder and target, starting for the initial data produced above:

```
167 t, data = galcol.evolve_two_disks(
168                 galaxies['target'], galaxies['intruder'],
169                 N_steps=10000, N_snapshots=500,
170                 time_step=0.05*unit.Myr)
```

While the integration proceeds, progress is printed in percent of the total number of time steps, which is 10000 in this case. This is achieved by a print statement in the main loop (see line 130 in the code listing of `evolve_two_disks()`); the format option `end=""` prevents a new line after each call and, owing to the carriage return `"\r"`, printing starts over at the beginning of the same line). The function completes with

```
Solving equations of motion for two galaxies (Leapfrog integration)
100 % (stopped at t = 500.0 Myr)
```

The final time $t = 500$ Myr follows from the chosen timestep, $\Delta t = 0.05$ Myr, times the total number of timesteps.

Various options for visualization are available in `galcol` (in the exercises, you have the opportunity to explore their capabilities). For example, to produce a three dimensional scatter plot of the stars for a particular snapshot, you can use

```
171 i = 100
172 galcol.show_two_disks_3d(data[i,:,:],
173                          galaxies['target']['N_stars'],
174                          [-15,15], [-15,15], [-15,15], t[i],
175                          'two_disks')
```

This call was used to produce the upper left plot for $t = 100$ Myr in Fig. 4.13. From the parameters in lines 169–170 you an calculate that the snapshot index i simply corresponds to the time in Myr. In the call above, the slice `data[i,:,:]` for the snapshot with index `i` is passed as first argument of `show_two_disks_3d()`. The second argument allows the function to infer the number of stars in the two disks, which is needed to display stars belonging to the intruder and target galaxies in blue and red, respectively. The following arguments set the x, y, and z-range of the plot in units of kpc. The snapshot time $t[i]$ is required for the label on top of the plot (if it is omitted, no label is produced). The last argument is also optional and specifies the prefix of the filename under which the plot is saved (the full filename is composed from the prefix and the snapshot time). The rendering is based on the `scatter()` function from `pyplot`. In contrast to a surface plot (see Sect. 3.2.1), a scatter plot shows arbitrarily distributed data points. The function `scatter()` can also be used for two-dimensional plots, i.e. points in a plane. As you can see from the source code in the file `galcol.py`, we use `Axes3D` from `mpl_toolkits.mplot3d` to create three-dimensional coordinate axes.

Figure 4.13 shows a sequence of plots illustrating the evolution of the two disks ranging from $t = 100$ Myr, where the intruder is still approaching the largely unperturbed target, to 450 Myr, where the remnants of the galaxies are moving apart (the centers of mass follow hyperbolic trajectories). The intruder is strongly disrupted during its slingshot motion through the potential well of the target's larger central mass. The plots in the middle show that a large fraction of the intruder's stars are ejected, while the intruder triggers spiral waves in the target disk through tidal forces. Although some stars in the outer part of the target disk are driven away from the center, the effect is less dramatic because the stars are bound more tightly. There is also function `anim_two_disks_3d()` for producing an animation of the snapshots in the module file `galcol.py` . While we do not discuss the details here, you can find an example in the online material for this chapter. The animation is saved in MP4 format and you can view it with common movie players.

The Whirlpool Galaxy M51 is a well known example for a close encounter of two galaxies (see also Sect. 5.4). There are other types, for example, the Cartwheel galaxy (see Exercise 4.13). In Fig. 4.14, you can see an optical image of a pair of galaxies with prominent tidal tails made by the Hubble Space Telescope (HST). Our simulation resembles such galaxies although the underlying model is very simple and completely ignores the gas contents of galaxies and the dark matter halos (see also the discussion in [14]). However, one should keep in mind that it is unrealistic in some important aspects:

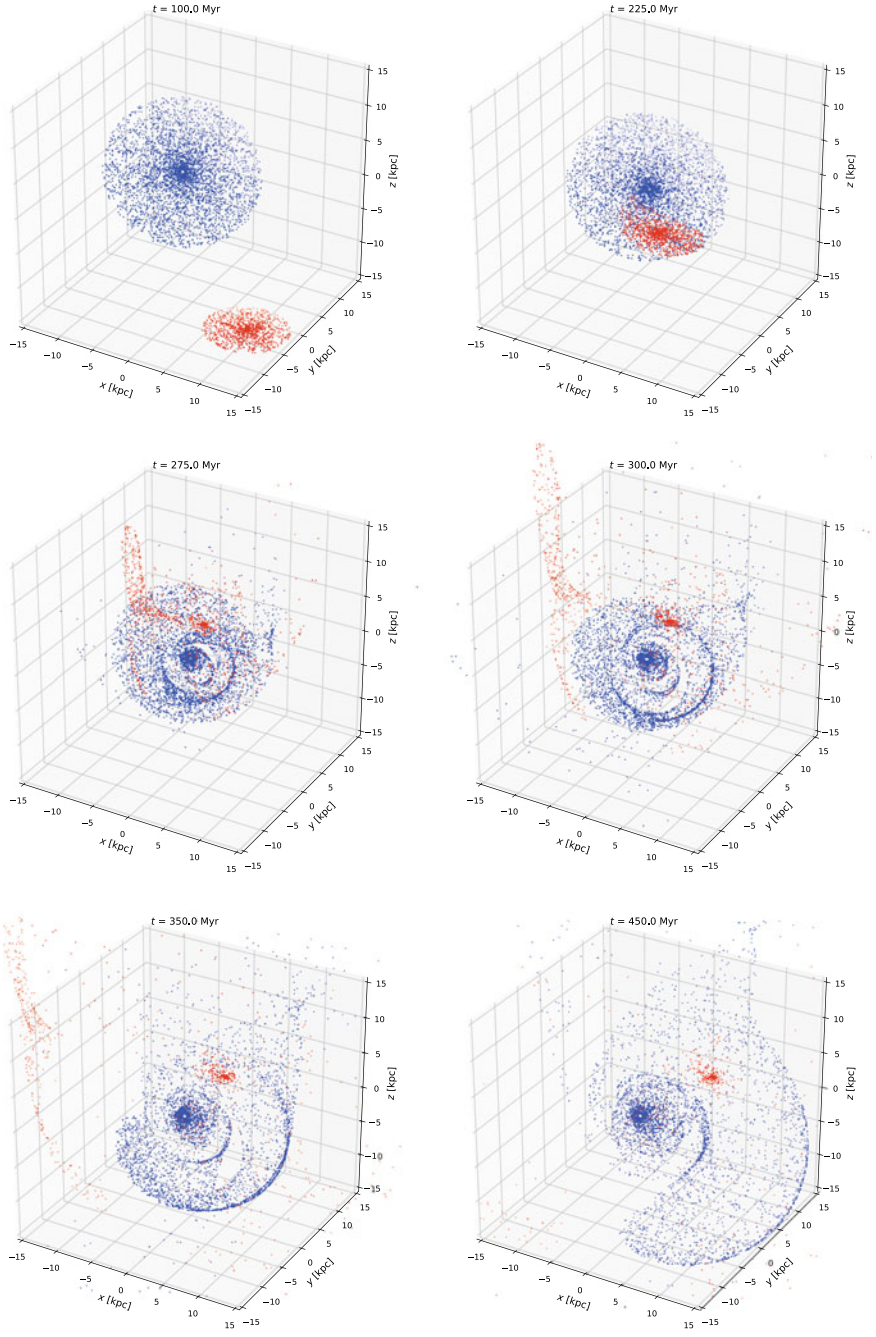

Fig. 4.13 Different stages of the collision of two galaxies (time in Myr is indicated on top of each plot). Stars of the target galaxy are shown in blue, stars of the intruder in red. The target disk has an inclination of 45° relative to the horizontal (xy) plane

Fig. 4.14 Hubble image of interacting galaxies Arp 87. Image credit: NASA, ESA, and The Hubble Heritage Team (STScI/AURA)

- The collision of real galaxies is not governed by two-body dynamics. The dark matter halos with their embedded disks of baryonic matter can even merge into a single galaxy.[25] It is believed that galaxies typically undergo several merges in the course of cosmic history.
- The ejection of stars through the slingshot effect, which is caused by strong acceleration during the close flyby of a gravitating center, is prone to numerical errors. To compute the trajectories more accurately a very low timestep would be required, which in turn would substantially increase the computing time. You can investigate the impact of the timestep on different trajectories in Exercise 4.14.

Exercises

4.11 Compute the evolution of an isolated disk in the xy-plane centered at (0, 0, 0) using `galcol.evolve_disk()`.

1. First consider the case where the center is at rest. Visualize the time evolution of the disk with `galcol.anim_disk_2d()`. Does the behaviour of the disk meet your expectation?
2. Since there are no perturbations of orbital motion by a second disk, the stars should follow circular Keplerian orbits. The function `galcol.show_orbits()` allows you to plot the numerically computed orbits of individual stars (the indices of these stars are passed as elements of an array to the function). Choose stars in the different rings and compare their motion over a given interval of time. Compute the disk evolution with different numerical timesteps. How are the orbits affected by the timestep, particularly in the innermost ring?

[25] The term baryonic refers to elementary particles in the atoms of which gas and stars are composed.

4.12 The effect of the collision of two galaxies depends mainly on their relative velocity and the impact parameter b, which is defined as the perpendicular distance between the path of the intruder galaxy from infinity and the center of the target galaxy. If the separation of the two galaxies at time $t = 0$ is large enough, it can be assumed that the intruder is nearly unaffected by the gravity of the target and moves along a straight line through its center in the direction of its initial velocity vector. The impact parameter is then given by the normal distance of this line to the center of the target.

1. Calculate b in kpc for the scenario discussed in this section. Vary the impact parameter by changing the initial position of the intruder. Compute the resulting evolution of the two disks and interpret the results.
2. What is the effect of the relative velocity of the intruder and the target for a given impact parameter?
3. The relative orientation of the disks and the mass ratio also play a role in the interaction process. Investigate for one of the scenarios from above orientations of the target disk ranging from $\theta = 0°$ (planes of target and intruder are parallel) to 90° (target perpendicular to intruder) and compare the mass ratios 1 : 5 (example above) and 1 : 1 (equal masses). Discuss how the central masses and stars are affected by these parameters.

4.13 The head-on collision of two disks can result in a Cartwheel-like galaxy [4, 14]. The name refers to the large outer ring which gives the galaxy the appearance of a wagon wheel. In this case, the intruder moves in z-direction toward the target and its normal is aligned with the direction of motion. The plane of the target disk is parallel to the intruder's disk. Vary the relative velocity and the impact parameter. Can you produce a post-collision galaxy of a similar shape as the Carthwheel Galaxy?

4.14 Analyze trajectories of ejected stars in some of the simulations from Exercise 4.12 or 4.13. Since you cannot predict which star in the initial disks will be ejected, take random samples of stars and plot their orbits with `galcol.show_orbits_3d()`.

1. Compute and plot the time-dependent specific orbital energy

$$\epsilon(t) = \frac{1}{2}v(t)^2 - \frac{GM_1}{r_1(t)} - \frac{GM_2}{r_2(t)}\,, \tag{4.75}$$

   for your sample of stars. The distances $r_{1,2}$ from the two galaxy centers are defined by Eq. (4.74). To compute the kinetic energy per unit mass, $v^2/2$, you need to modify `galcol.evolve_two_disks()` such that the position and velocity data are returned for each snapshot. Compare ejected stars to stars that remain bound to the target galaxy and describe the differences. Why is $\epsilon(t)$ in general not conserved?
2. How sensitive is $\epsilon(t)$ to the numerical time step?