Investigating Galactic Tidal Tails by Computer Modelling

Abstract

The behaviour and formation of tidal tails was investigated using computer modelling. Two galaxies were each modelled as a central heavy mass orbited by a number of light test masses, and the effect of the galaxies closely approaching each other was investigated using a fourth-order Runge-Kutta method. A number of parameters were varied, and the results were recorded. It was found that a prograde orbit of test particles relative to the heavy particles' trajectory lead to far more pronounced tidal tails than a retrograde orbit. The mass of one heavy particle was varied within a factor of ten of the mass of the other heavy particle. Decreasing the mass of the galaxy within this range led to test particles initially orbiting that galaxy to form more pronounced tidal tails, and to the tidal tails existing for longer. The effect of varying the closest approach distance of the two heavy masses was explored, and it was found that decreasing the distance of closest approach led to more test particles being more significantly perturbed and to more pronounced tidal tails within the range explored. [Word count: 2952 words]

1 Introduction

1.1 Background

Tidal tails are thin regions of galaxies that protrude from the rest of a galaxy, giving it an unconventional shape. The tails emerge due to the gravitational interaction between two closely approaching galaxies, and remain as the two galaxies move further apart. Further, tidal tails lead to various astrophysical phenomena: more than 1% of stars are formed in tidal tails [1], and it has been proposed that tidal tails are able to spawn dwarf galaxies[2].

Tidal tails were first studied in detail in 1953 by Zwicky [3], and were generally believed to result from gravitational interaction. However, it was unclear whether gravity alone was sufficient to form tidal tails (for example, Halton Arp expressed doubts in his *Atlas of Perculiar Galaxies* [4]). In 1972, Toomre and Toomre used a computer simulation to show that tidal tails could be caused by gravity alone [5]. Their simulation detailed how tidal tails could form due to gravity, and how their appearance and behaviour varied depending on the relative trajectory of the two galaxies.

1.2 Goals of this Report

This report seeks to verify and extend Toomre and Toomre's [5] work, using modern computing resources to verify that gravity by itself can be responsible for tidal tails. It explores some of the behaviour of tidal tails - in particular, the dependence on the galaxy masses, closest approach distance, and orbit direction.

2 Analysis

2.1 Physics of the System

To reduce computation time, each galaxy was modelled as a single heavy mass orbited by a number of test particles whose own gravity was considered negligible.

Particle motion is governed by the equation of acceleration due to gravity:

$$a = -\sum_h \frac{Gm_h}{r_h^3}\mathbf{r}_h \tag{1}$$

where the index h is used to represent each heavy mass, and $\mathbf{r}_h$ is the corresponding radius, G is the gravitational constant, and m_h is the mass of the heavy mass at index h.

Initially, the test particles are in circular orbits around one of the heavy masses, ignoring the other heavy mass. The initial condition of the system must therefore satisfy the requirement for circular

motion:

$$a = \frac{v^2}{r} \tag{2}$$

for each test particle. Here, a is the acceleration of a particle, v is its velocity, and r is its radius.

Unless otherwise specified, I only considered situations in which the heavy masses travel in a parabolic orbit - i.e. the total energy of the two heavy masses was zero. To implement this, the total energy of each heavy mass was set to zero.

2.2 Rescaling

To reduce errors, physical constants were rescaled such that all number variables would have similar orders of magnitude. I therefore set the value of G and the masses of the two heavy masses to unity.

2.3 Ordinary Differential Equation Solvers

I considered in depth two methods of solving the Ordinary Differential Equation (ODE): Verlet integration and Runge-Kutta integration. Verlet integration has the benefit of being symplectic (and so would follow conservation laws well), being numerically stable, and being fast[6]. Runge-Kutta integration has the benefits of introducing smaller errors, reliability, and being implemented in the vast majority of ODE solving libraries[7]. The computation used SciPy's Solve_IVP function, which included Runge-Kutta integration but not Verlet integration. For these reasons, Runge-Kutta integration was selected.

This implementation allowed easy switching between different order Runge-Kutta methods. An nth order Runge-Kutta method introduces an $O(h^{n+1})$ error in each step of size h but takes $O(n)$ time in each step. As such, I could use lower order integration for preliminary measurement, and higher order integration for more important, precise calculations.

2.4 Details of Runge-Kutta Integration

In this subsection, the operation of the Runge-Kutta method is explained.

The Runge-Kutta method works as an extension of the Euler method, where each variable y is calculated by

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{3}$$

where h is the size of the timestep and $f(t_n, y_n)$ is the time derivative of y_n when $t = t_n$. The Euler method has an error of size $O(h^2)$ (per step)[7].

The Euler method may be improved by using the gradient at the midpoint rather than the start of the step. Therefore, one can take a step of size $h/2$ as a "trial" step to calculate the values of y_n at the midpoint of t_n and t_{n+1}, and use this to calculate $f(t, y)$ at this midpoint. Using this, one can take a further step using the Euler method:

$$k_1 = hf(t_n, y_n) \tag{4}$$

$$k_2 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \tag{5}$$

$$y_{n+1} = y_n + k_2 \tag{6}$$

This is the Midpoint method. Here, k_1 is the increase due to the Euler method of a step of size h, and k_2 is the increase due to the Midpoint method. Note that k_2 relies on the value of k_1.

This method has errors of order $O(h^2)$ (per step)[7]. It can be extended further by using its result as a further trial step. Indeed, any extension of this form can be extended further by treating its result as an additional trial step. This is the basis of the Runge-Kutta method. For this experiment, a fourth-order Runge-Kutta method was used. This uses the same k_1 and k_2 as above, and also introduces:

$$k_3 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \tag{7}$$

$$k_4 = hf(t_n + h, y_n + k_3) \tag{8}$$

It can then be shown that

$$y_n + 1 = y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 \tag{9}$$

introduces only $O(h^5)$ errors per timestep[7]. Note that k_1 and k_4 are approximations of the full step whereas k_2 and k_3 are approximations of a half step.

2.5 Time Complexity

While the program was required to generate particles, calculate their motion, and plot the result in a meaningful way, motion calculation took the most time to run by far. This is because the Runge-Kutta function calculation was repeated for a large number of time steps and for a large number of particles. The best way to ensure the program ran quickly was therefore to ensure that a step in the Runge-Kutta method took as little time as possible - this part of the code was therefore optimised to run as quickly as possible.

The model includes a large number of parameters (positions and velocities of two heavy masses and a large number of test masses), but most of these are independent of each other. For improved performance, I ensured the computation did not waste time considering the dependence of independent parameters. Indeed, for n particles, a naive implementation would consider the relationship between all pairs of particles. There are $O(n^2)$ such pairs, so each step would take $O(n^2)$ time. However, it is feasible to calculate each step in $O(n)$ time as every particle's motion can be calculated in constant time and there are $O(n)$ such particles. As n could be large, it was crucial to ensure that each step in the computation only took $O(n)$ time.

2.6 ODE Iteration

One method of ensuring that the computation time is $O(n)$ would be to write an ODE solver that solves the two-body problem of two heavy masses, and an ODE solver that computes the path of one test particle (using the resulting positions of the heavy masses). Both of these operations would take $O(1)$ time and therefore running them on all ($O(n)$) relevant particles would take $O(n)$ time. Further, it would be very readable and easy to fault-find.

This method has some problems, however. Runge-Kutta integration involves evaluating the value of the ODE function at various times t, and this depends on the distance between the test particle being considered and the heavy particles. As such, it would be necessary to quickly find the position of heavy particles at any given time t. If the stepsizes of both the heavy particle solver and the test particle solver were guaranteed to be the same, this would not be a problem as a list could simply be constructed such that the heavy particle positions at some time t could be looked up in constant time. However, in order to allow the stepsize to be varied, it would be necessary to approximate the positions of heavy particles at times that were not calculated in the heavy particle motion calculation. This would introduce additional errors. Further, some optimisation involved in calculating all particle trajectories at the same time would not be possible.

Therefore, I decided to use one array for all particle positions and velocities. This was implemented in such a way that each particle's motion calculation took constant time, so the total calculation took $O(n)$ time.

2.7 Optimisation

To optimise the speed of the Runge-Kutta integration, I used a few additional means of optimising the code.

Where possible, vectorised code (optimised for performing the same operation on all elements of a large list or array) was used.

I went through it investigating how long different lines took. I found that some NumPy functions were slowing down the code substantially. In particular, calls to numpy.linalg.norm caused substantial delays. I developed functions to replace the offending NumPy functions and this resulted in a substantially faster execution (computation time was reduced by a factor of five).

2.8 Infinities Arising due to Collisions

A test particle that approaches a heavy mass will experience a gravitational force that approaches infinity. This is a problem as infinities arising will lead to large errors.

To avoid this, collisions were first modelled by multiplying the force on a particle due to a heavy mass's gravity by a negative constant when a particle came within a predetermined radius of the heavy particle. After some experimentation with different values, I decided to set the constant to 0 as this resulted in energy conservation. This may result in the individual collision appearing unrealistic as particles pass through anything they collide with, but ensures that the effect of collisions are realistic to the simulation as a whole. In particular, particles cannot gain any energy due to collisions and infinities cannot arise.

A predetermined radius of 0.5 au was found to be useful as this ensured that gravitational forces were kept sufficiently low for the errors in the gravitational forces to be acceptable and for particles entering the radius to be a rare effect (thereby minimising the effect on the simulation as a whole).

3 Implementation

3.1 Overview of Process

Initially, test particles were programmed to orbit one heavy mass at a fixed position. I checked that this led to satisfactory results and that stable circular orbits could be maintained. In order to check

this, a program to plot and animate the result of the computation was written.

However, I found that Runge-Kutta integration was unstable for particles at low (2 to 3 au) radii. This resulted from numerical errors propagating such that the particles fell in to the centre of the galaxy despite starting with the correct velocity for circular motion. Particles at larger radii (4 au or more) had stable orbits within the timescale used, and so Runge-Kutta integration was used for computations as long as they were still at stable radii and timescales. For computations at radii or timescales that would cause Runge-Kutta integration to be unstable, I decided to use Radau integration. This is because Radau integration is designed for stiff problems[8], and the instability and large errors that were found are indicative of the problem being a stiff problem. Further, Radau integration is included in the SciPy solve_IVP function that was used for the computation.

Having verified that the motion of one galaxy was computed correctly, I added a second heavy mass and increased the number of test particles. To generate quantitative results, I wrote an additional program to count the number of perturbed particles at different points of time. As the program grew, I decided to split it up into separate files for parameters, generating the initial conditions, and running the integration. I also spent time going through the bottlenecks in the computation ensuring that it ran as quickly as possible (discussed further in section 2.7).

3.2 Performance

For a system with 3,500 test particles, the program took 5.5±0.1 minutes to run using an Intel i5-7300HQ processor with 8 GB RAM. Of this, saving results to file took 0.5 ± 0.1 minutes with the rest of the time being taken up by the computation itself.

3.3 Experiments Made

First, the computation was run to verify that tidal tails could form. Tidal tail dependence on the direction of galactic orbit, galaxy masses, and distance of closest approach were then tested and explored. Finally, I ran some computations of non-parabolic orbits.

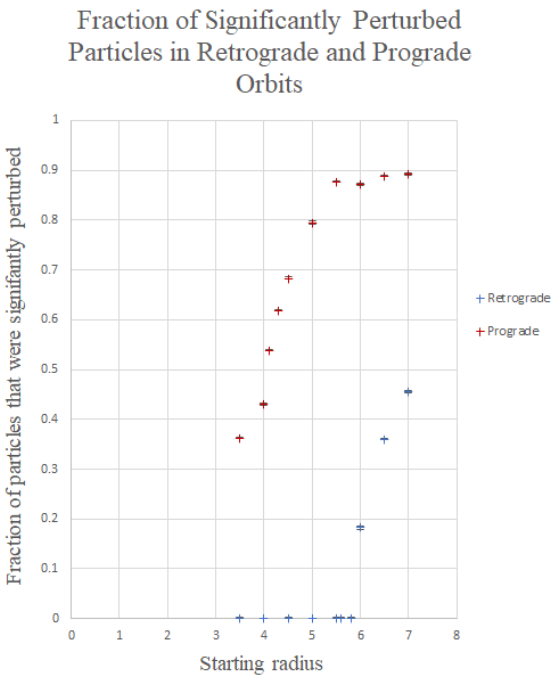

Figure 1: Plot of the fraction of test particles that are significantly perturbed in prograde and retrograde galactic orbits. A particle is considered to be significantly perturbed if the radius between the centre of the galaxy and the particle position is changed by at least a factor of 1.5.

4 Results and Discussion

4.1 Tidal Tail Formation

In the simulations, tidal tails appeared. This verifies that the existence of tidal tails can be attributed to gravity by itself.

4.2 Prograde and Retrograde Orbits

It was found that tidal tails are more pronounced in a galaxy that spins in a prograde direction relative to the trajectory of the heavy masses than in a galaxy that spins in a retrograde direction. Figure 1 shows how the fraction of test particles that were perturbed significantly varies with the spin direction of a galaxy. A particle is considered to be significantly perturbed if the radius between the centre of the galaxy and the particle position is changed by at least a factor of 1.5. For this test, the initial conditions were set such that heavy masses started at positions of (0 au,0 au) and (15 au,-35 au) respectively. The heavy particles' closest approach was at t = 80.21 ± 0.01 au, and figure 1 shows data for t= 140 au. Figures 2, 3, and 4 show visually how the system develops over time. They

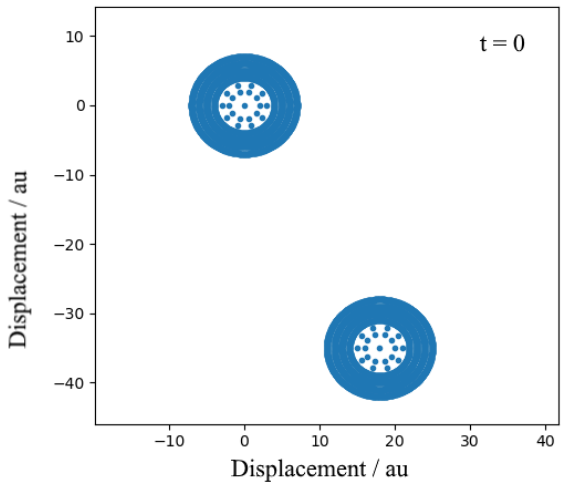

Figure 2: Plot of two galaxies spinning in opposite directions, at t = 0 au.

show a system of two galaxies spinning in opposite directions.

Qualitatively, viewing an animation of the simulation gives an impression of why prograde motion leads to more pronounced tidal tails. During prograde motion, a particle that is close to the perturbing galaxy during the time of closest approach will remain close to the perturbing galaxy for longer. It therefore has more time to be perturbed by a stronger gravitational force, leading to more pronounced effects.

These results are in agreement with Toomre and Toomre's results[5].

4.3 Changing the Masses of Galaxies

The mass of the heavy particle at the galaxy, m1, was varied within the range 10 au $\geq$ m1 $\geq$ 0.1 au while the mass of the perturbing galaxy was kept constant at 1 au. The two heavy galaxy centres were initially positioned at (0 au,0 au) and (18 au, 35 au) with a parabolic trajectory. Plots of the resulting perturbed galaxy are shown in figure 5. As illustrated, varying the mass m1 leads to large changes in the resulting galaxies. For larger values of m1, the part of the tidal tail between the two galaxies contains fewer particles. Larger values of m1 also led to tidal tails appearing for less time, as the gravitational force acts more quickly. For m1 $\leq$ 1 au, part of the tidal tail was found to protrude in a direction away from the perturbing galaxy. This effect increased for decreasing m1.

Also in this range, decreasing m1 resulted in more test particles becoming unbound to either of the heavy particles. The resulting tidal tails would grow and, given time, eventually separate from the heavy masses.

4.4 Closest Approach Distances

Next, I varied the distance of closest approach between the two heavy masses and measured the results. As expected, a decrease of the closest approach distance led to more pronounced tidal tail effects. A plot of the proportion of test particles that were significantly perturbed (a change of radius of a factor of at least 1.5) against the distance of closest approach is shown in figure 6. The simulation used 1000 particles at initial radii r = 5, 6, and 7 au, and 500 particles at initial radius 4 au.

The plot illustrates that a larger proportion of particles were perturbed at higher initial radii. This is unsurprising: particles at higher radii experience a lower gravitational force from their initial galaxy and a larger gravitational force from the perturbing galaxy, and therefore are affected more by the perturbing galaxy.

4.5 Non-Parabolic Orbits

It has been verified that tidal tails can be formed by gravity when two galaxies approach with parabolic trajectories. However, this does not necessarily mean that tidal tails can be formed by gravity in roughly parabolic approaches, and, as different galaxies are not likely to be directly related, it is unlikely that two closely-approaching galaxies will have exactly parabolic orbits. Therefore, in order to verify that gravity can cause tidal tails in real galaxies, the computation was modified such that the two galaxies approached each other in roughly parabolic orbits (the velocity of one of the heavy particles was modified by a factor of 1.05). Tidal tails still appeared, confirming that gravity is able to form tidal tails in these conditions.

4.6 Errors

4.6.1 Start and End of Integration

The computation only considered a certain period of time, and the effect of gravity outside this time period was ignored. Because of this, I made sure that the initial and final conditions included that the heavy particles were sufficiently far away from each other other for the ignored effects to be insignificant.

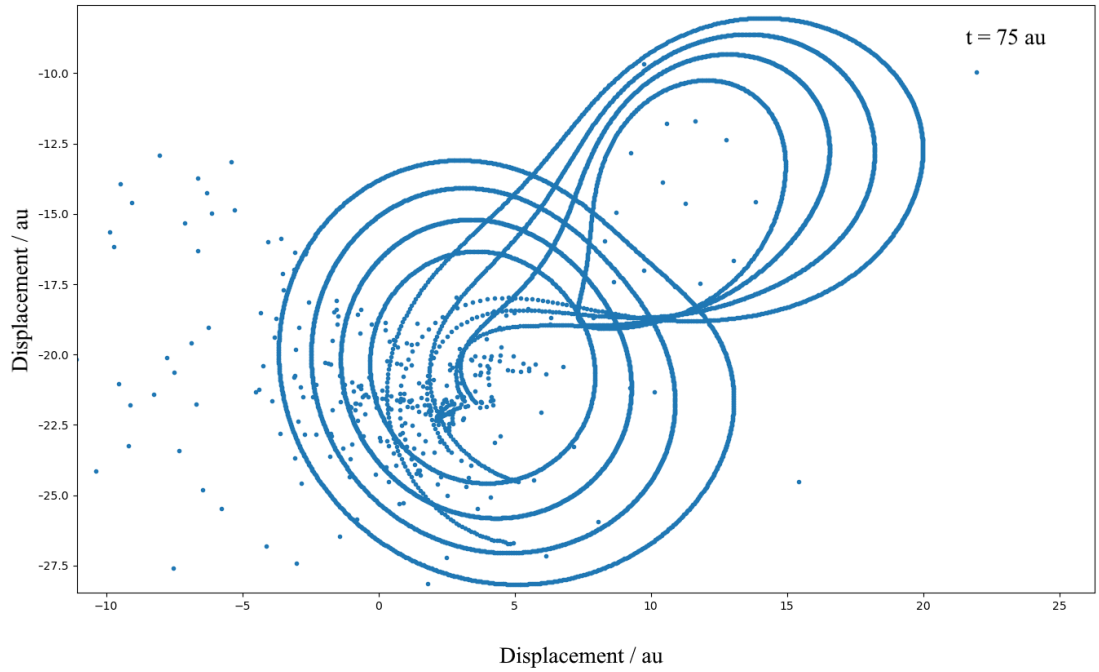

Figure 3: Plot of the same two galaxies as in figure 2 at t = 75 au, shortly before the time of closest approach between the two heavy masses. Note that the axis scales have changed. Note also that the galaxy in the top-right (with prograde spin) is far more perturbed than the galaxy in the bottom-left (with retrograde spin).

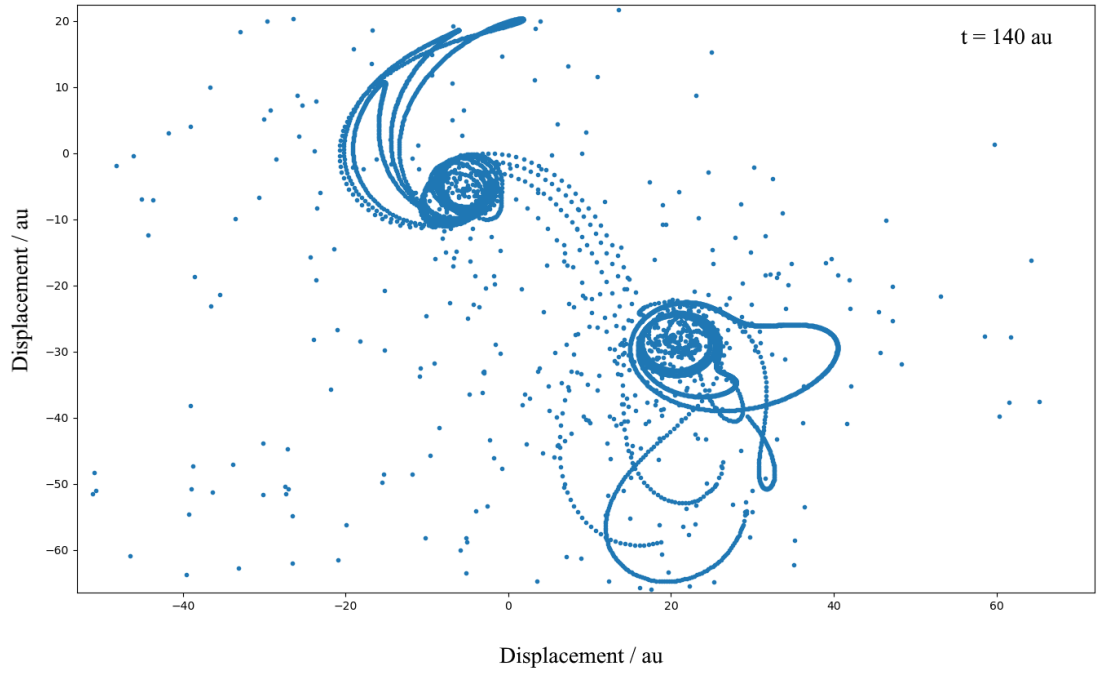

Figure 4: Plot of the same two galaxies as in 2 and 3 at t = 140 au. Note that the axis scales have changed. Note also that the galaxy in the top-left (with prograde spin) is far more perturbed than the galaxy in the bottom-right (with retrograde spin). The tidal tails on the bottom-right galaxy were originally gravitationally bound to the other galaxy, but were perturbed so significantly that they ended bound to the bottom-right galaxy.

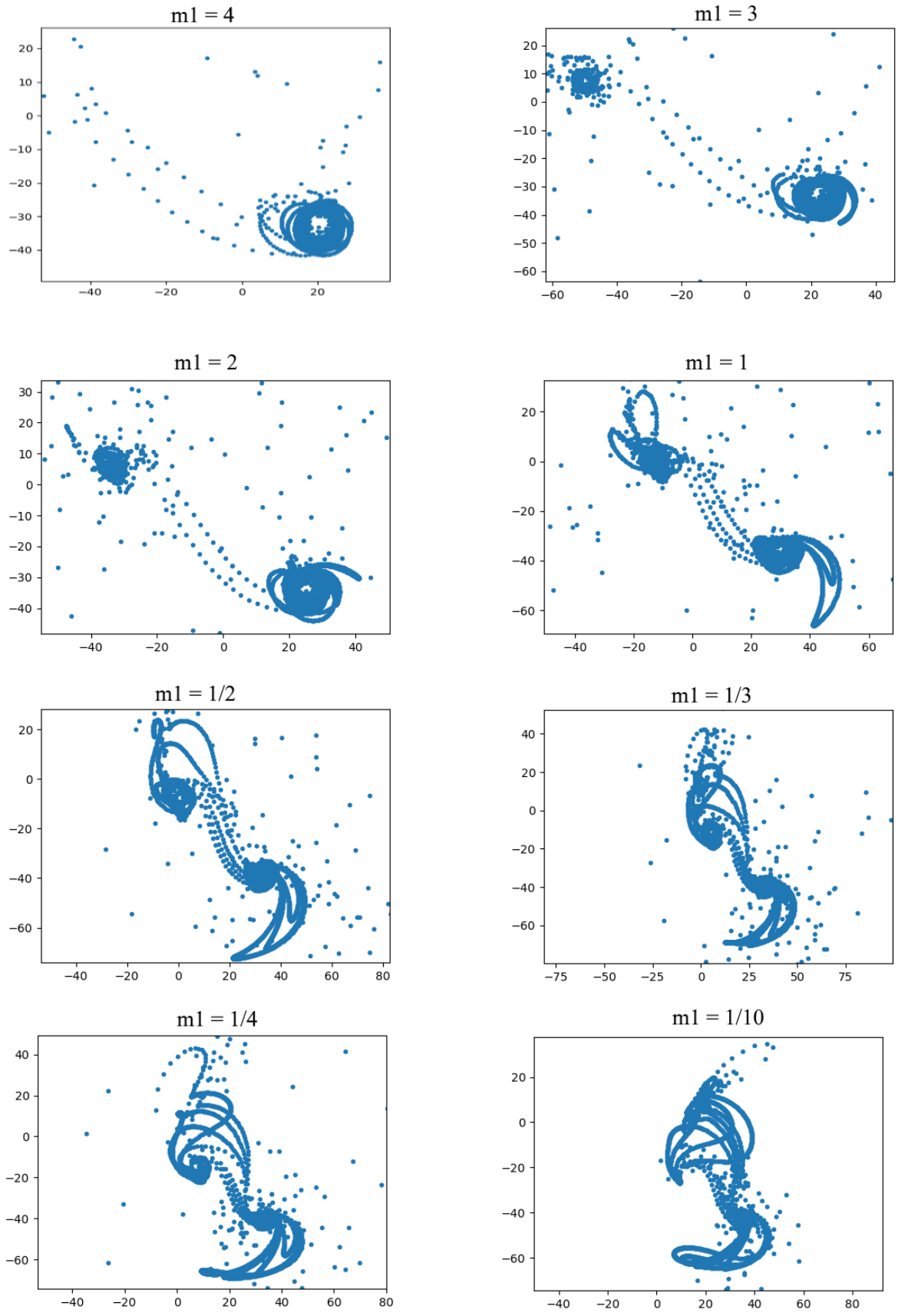

Figure 5: Plots of test particles of the perturbed galaxy at t = 200 au for varying values of the mass of the perturbed galaxy, m1. The perturbed galaxy is at the bottom right of these plots; the perturbing galaxy is in the top left.

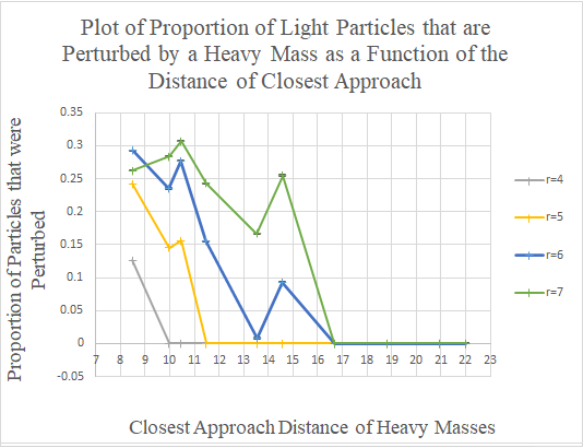

Figure 6: Plot of the proportion of particles that were significantly perturbed as a function of the closest approach distance. Different initial radii are plotted in different colours. The error, while not clearly visible in the plot, is one part in 1,000 for r=5,6,7 and one part in 500 for r=4.

4.6.2 Dependence on Unconsidered Variables

A potential problem in modelling a physical system is that it may involve a number of interdependent parameters. This problem is present in the present system, and this makes it challenging to assess the exact impact of varying only one parameter.

For example, figure 6 shows that the fraction of perturbed masses at different radii have a relatively weak correlation with the distance of closest approach but that a strong correlation exists between the fraction of perturbed masses at different radii. This may indicate that an interdependent parameter has an effect on the results. I believe that a likely cause is that the change of the distance of closest approach also changes the path along which the two heavy particles travel. The particular path, characterised by more than just the distance of closest approach, may have had observable effects on the tidal tails.

Changing the mass of particles also changes other parameters. For example, the initial particle positions were kept constant between varying mass measurements in section 4.3. However, changing the mass of particles also changed the path and with it the distance of closest approach. This may have had a noticeable effect. Further, changing the mass of the heavy particle changes the velocity of any particle in a circular orbit.

These factors are likely to impact the formation of tidal tails, so may introduce systematic errors. However, more detailed research work could further investigate the exact effects of interdependent parameters.

5 Conclusion

The two galaxies were successfully modelled as a central heavy mass surrounded by test particles in initial circular orbits. It was found that prograde orbits of the test particles with respect to the galaxy trajectory led to significantly more pronounced tidal tails than retrograde orbits. As the distance of closest approach is decreased, more particles are more significantly perturbed. Further, particles initially more distant from their initial galaxy are more significantly perturbed. The effect of changing (by a factor of up to 10) the mass of the galaxy that test particles initially orbit was investigated. I found that a greater mass led to fewer particles in part of the tidal tail joining the two galaxies, and that tidal tails were more pronounced for lower initial masses within the range investigated. Increasing the mass also led to the tidal tails appearing for less time as the tidal tails were destroyed more quickly by the larger gravitational force.

6 References

1. Naeye, R. (2007, December 18). 'Shot in the Dark' Star Explosion Stuns Astronomers. Retrieved April 24, 2020, from https://www.nasa.gov/centers/goddard/news/topstory/2007/intergalatic_shot.html

2. Barnes, J. E., Hernquist, L. (1992). Formation of Dwarf Galaxies in Tidal Tails. *Nature*, 360, 715–717. doi: 10.1038/360715a0

3. Zwicky, F. (1953). Luminous and dark formations of intergalactic matter. *Physics Today*, 6(4), 7–11. doi: 10.1063/1.3061224]

4. Arp, H. (1966). Atlas of Peculiar Galaxies. *The Astrophysical Journal Supplement Series*, 14, 1. doi: 10.1086/190147

5. Toomre, A., Toomre, J. (1972). Galactic Bridges and Tails. *The Astrophysical Journal*, 178, 623. doi: 10.1086/151823

6. Verlet, L. (1967). Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones

Molecules. *Physical Review*, 159, 98–103. doi: 10.1103/PhysRev.159.98

7. Press, W et al. Cambridge University Press. (2007). 17. In *Numerical Recipes* (3rd ed.). Cambridge.

8. Hairer, E., Wanner, G. (2010). Sec. IV.8. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg

Appendix

Helper_Functions.py

```
#The Particle class stores Particles in a tidy, readable way. It can't be used in the ODE solver (which
# requires a 1d array-like of numbers) but is useful for particle generation.
class Particle:
    """The Particle class stores Particles in a tidy, readable way. It can't be used in the ODE solver (which
    requires a 1d array-like of numbers) but is useful for particle generation."""
    def __init__(self, position, velocity):
        self.position = position
        self.velocity = velocity

    def getPosition(self):
        return self.position

    def getVelocity(self):
        return self.velocity
```

Parameters.py

```
import numpy as np

#This file contains all variable parameters

#G, the gravitational constant
G = 1

#Parameters corresponding to all heavy masses.
number_of_heavy_masses = 2
heavy_mass_masses = (1,1)
heavy_mass_starting_positions = [[0,0,0],[18,-35,0]]
heavy_mass_starting_velocity_directions= [np.array([0,-1,0]),np.array([0,1,0])]
heavy_mass_radii = [0.5,0.5] #these radii are the radii below which light particles are considered to collide with the heavy particle

light_particle_radii = (4,5,6,7)
points_at_each_radius = (500,1000,1000,1000)


max_time = 200

save_file_name = "data.csv"

def check_parameters():

    if (number_of_heavy_masses != len(heavy_mass_masses)):
        raise Exception("The number of heavy masses should be the same as the length of the heavy mass masses tuple.")
    elif (number_of_heavy_masses != len(heavy_mass_starting_positions)):
        raise Exception("The number of heavy masses should be the same as the length of the heavy mass starting positions list.")
    elif (number_of_heavy_masses != len(heavy_mass_starting_velocity_directions)):
        raise Exception("The number of heavy masses should be the same as the length of the heavy mass starting velocities list")
    elif (number_of_heavy_masses != len(heavy_mass_radii)):
        raise Exception("The number of heavy masses should be the same as the length of the heavy mass radii list")
    elif (len(light_particle_radii) != len(points_at_each_radius)):
        raise Exception("The length of the radii and points_at_each_radius lists should be the same length.")
    else:
        return True
```

Particles_Generator.py

```
#Using the parameters from Parameters, this file generates a list of particles formatted such that they can be solved by the ODE solver.

```

```
import numpy as np
from Parameters import *
from Helper_Functions import Particle

def circular_orbit_initial_condition(particles, relative_position, orbitted_particle_index, clockwise = True):
    """Returns a Particle object set to circularly orbit a heavy particle. particles should be a list of Particle objects corresponding to all the particles in the system (where the first number_of_heavy_masses particles are heavy). relative_position is the relative position of the particle being set from the particle it is orbiting, and orbitted_particle_index is the index of the particle being orbitted. clockwise determines if the orbit is clockwise or anticlockwise."""
    speed = np.sqrt(G * heavy_mass_masses[orbitted_particle_index] / np.linalg.norm(relative_position))

    relative_velocity = np.cross(relative_position,[0,0,1]) * (speed / np.linalg.norm(relative_position))
    if not(clockwise):
        relative_velocity = relative_velocity * -1
    return Particle(np.add(heavy_mass_starting_positions[orbitted_particle_index], relative_position),np.add(particles[orbitted_particle_index].velocity, relative_velocity))

def circular_polar_to_cartesian(radius, theta):
    """Converts a position from circular polar coordinates to 3d cartesian coordinates. radius is the distance from 0,0,0 and theta is the angle to the x axis."""
    return [radius * np.cos(theta), radius * np.sin(theta), 0]

def calculate_parabolic_orbit_particle_speed(particles, particle_index):
    '''Calculates the speed required for a particle to have a total energy of 0, given the particles list (of Particle objects) and the index of the particle.'''
    gravitational_potential_energy = 0
    for i in range(number_of_heavy_masses):
        if(i != particle_index):
            gravitational_potential_energy -= G *  heavy_mass_masses[i] / np.linalg.norm(np.subtract(particles[particle_index].position, particles[i].position))

    speed = np.sqrt(-1 * 2 * gravitational_potential_energy)

    return speed

def format_particles_for_solving(particles):
    """Reformats the array particles so that every Particle object in the initial array is replaced with three elements corresponding to its position then three elements corresponding to its velocity."""
    formatted_particles = []
    for i in range(len(particles)):
        particle = particles[i]
        formatted_particles.extend(particle.position)
        formatted_particles.extend(particle.velocity)

    return formatted_particles

def set_up_initial_conditions():
    """Set up the positions of all heavy masses. Then, using these, calculate the velocity of each heavy mass such that its kinetic energy plus its gravitational potenetial energy is zero.
    This ensures that the total energy is zero, i.e. the orbit is parabolic. Next, generate all light particles, and finally reformat them so they can be sent into solve_ode."""

    check_parameters()

    particles = []
    for j in range(number_of_heavy_masses):
        particles.append(Particle(heavy_mass_starting_positions[j], [0,0,0]))

    for j in range(number_of_heavy_masses):
```

```
        particles[j].velocity = (heavy_mass_starting_velocity_directions[j]) * calculate_parabolic_orbit_particle_speed(particles, j)

    #Set up light particles
    for k in range(1):
        for i in range(len(light_particle_radii)):
            r = light_particle_radii[i]
            for j in range(points_at_each_radius[i]):
                    particles.append(circular_orbit_initial_condition(particles, circular_polar_to_cartesian(r, 2*np.pi * j/points_at_each_radius[i]), k, clockwise = (k==1)))

    particles = format_particles_for_solving(particles)

    return particles
```

Computation.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

import time
import sys

from Parameters import *
from Particles_Generator import set_up_initial_conditions

squares_of_heavy_mass_radii = np.square(heavy_mass_radii)

def get_particle_attribute(particles, particle_start_index, is_velocity = False):
    """Gets some parameter - either the position (if is_velocity is false) or velocity (if is_velocity is true) - of a particle"""
    velocity_offset = 3 if is_velocity else 0
    answer = []
    for i in range(3):
        answer.append(particles[particle_start_index + i + velocity_offset])
    return answer

def get_particle_position(particles, particle_start_index):
    """Gets the position of a particle at particle_start_index. Requires the array particles to be formatted as an array of particles,
    first position then velocity."""
    return get_particle_attribute(particles, particle_start_index, False)

def get_particle_velocity(particles, particle_start_index):
    """Gets the velocity of a particle at particle_start_index. Requires the array particles to be formatted as an array of particles,
    first position then velocity."""
    return get_particle_attribute(particles, particle_start_index, True)


def gravity_force(t,particles):
    """Calculate gravitational force on all particles at some point in time t. particles should be an array of all the particles
    in the format [x,y,z, xdot, ydot, zdot]. An array containing the time derivatives of each of these variables is returned."""

    #First, find the positions of all heavy masses
    heavy_mass_positions = []
    for i in range(number_of_heavy_masses):
        heavy_mass_positions.append(get_particle_position(particles,6*i))

    answer = np.empty(len(particles))

    for particle_start_index in range(0, len(particles), 6):
        particle_position = get_particle_position(particles, particle_start_index)

        #find the distances to heavy particles
        #fast_subtract replaces np.subtract - it's slightly faster at the expense of being less generalised and elegant.
```

```
48          def fast_subtract(list_1, list_2):
49              return np.array([list_1[0]-list_2[0], list_1[1] - list_2[1], list_1[2] -
     list_2[2]])
50          heavy_particle_distances = [fast_subtract(particle_position,
    heavy_mass_positions[0]),
51          fast_subtract(particle_position, heavy_mass_positions[1])]
52
53          def calculate_acceleration(heavy_particle_distances, particle_start_index):
54              """Calculate the total acceleration by adding all acceleration_term
    values for every heavy particle in the system."""
55              def acceleration_term(heavy_particle_distances, heavy_mass_index):
56                  """Calculate (and return) the acceleration on a particle due to the
    heavy mass with index heavy_mass_index and at position r (relative to the
    particle
57                  whose acceleration is being calculated"""
58
59                  #r squared is stored because finding the square root was found to be
     significantly more computationally expensive
60                  r_squared = heavy_particle_distances[heavy_mass_index][0]**2 +
    heavy_particle_distances[heavy_mass_index][1]**2 + heavy_particle_distances[
    heavy_mass_index][2]**2
61
62                  magnitude = (-1 * G * heavy_mass_masses[heavy_mass_index] /
    r_squared**(3/2))
63                  if (r_squared < squares_of_heavy_mass_radii[heavy_mass_index]):
64                      magnitude = 0
65                  direction = heavy_particle_distances[heavy_mass_index]
66
67                  return (magnitude * direction)
68
69              acceleration = 0
70              for heavy_mass_index in range(number_of_heavy_masses):
71                  if (particle_start_index != 6*heavy_mass_index): #Don't consider the
     gravitational force on a particle due to itself
72                      acceleration += acceleration_term(heavy_particle_distances,
    heavy_mass_index)
73              return acceleration
74
75
76          acceleration = calculate_acceleration(heavy_particle_distances,
    particle_start_index)
77
78          for j in range(3):
79              answer[particle_start_index + j] = particles[particle_start_index + j+3]
     #The time derivative of a particle's position is just its velocity, which is
    stored three elements along in the array
80              answer[particle_start_index + j + 3] = acceleration[j] #The time
    derivative of velocity is the acceleration, which has been calculated.
81      return answer
82
83  def heavy_particles_closest_approach_event(t, particles):
84      """Returns the dot product of the relative position and velocity of the (first)
    two heavy particles. This will equal
85      zero at a extremal point of particle distance."""
86      relative_position = np.subtract(get_particle_position(particles,6),
    get_particle_position(particles,0))
87      relative_velocity = np.subtract(get_particle_velocity(particles,6),
    get_particle_velocity(particles, 0))
88
89      return np.dot(relative_position, relative_velocity)
90
91  def get_closest_approach_values(integral_answer):
92      """Returns the time and distance of closest approach of the (first) two heavy
    particles. The argument, integral_answer, should be the returned value from a
    call
93      to solve_ivp, where solve_ivp's events arguments some function that crosses 0 at
     the particle's closest approach."""
94      closest_approach_y_events = integral_answer.y_events
95      if (len(closest_approach_y_events[0]) == 0):
96          raise Exception("find_distance_of_closest_approach failed; no extremal point
```

```
     was reached.")

    elif (len(closest_approach_y_events[0]) > 1):
        raise Exception("find_distance_of_closest_approach failed; multiple extremal
     points were reached.")
    else:
        closest_approach_time = integral_answer.t_events[0][0]
        particles_at_closest_approach = closest_approach_y_events[0][0]
        closest_approach_distance = np.linalg.norm(np.subtract(get_particle_position
    (particles_at_closest_approach,6), get_particle_position(
    particles_at_closest_approach, 0)))#make this line more legible

        return closest_approach_time, closest_approach_distance

def solve_ode(initial_condition):
    """Given an array of particles and velocities initial_condition, this uses SciPy
     solve_ivp to calculate the motion of the particles.
    The return value from the call to SciPy's solve_ivp is returned."""
    tsToPlot = np.linspace(0,max_time, 1000)
    integral_answer = solve_ivp(gravity_force, [0,max_time], initial_condition,
    events = heavy_particles_closest_approach_event, method = "RK45", t_eval =
    tsToPlot, first_step = 0.01)
    if (integral_answer.success == False):
        print("ODE Solver failed")
    try:
        closest_approach_time, closest_approach_distance =
    get_closest_approach_values(integral_answer)
        print("Closest approach: time  = ",closest_approach_time," and distance =",
    closest_approach_distance)
    except Exception as err:
        print(err)

    return integral_answer

def export_computation_to_csv(ode_solution):
    """Formats the result from solve_ivp, ode_solution, such that it can be saved to
     a csv file then saves it to a csv file."""
    data = [ode_solution.t,]
    for i in range(len(ode_solution.y)):
        data.append(ode_solution.y[i])
    print("Saving to file, time =", time.time())
    try:
        np.savetxt(save_file_name, data, delimiter=',')
    except:
        input("Saving to file failed. Check that it isn't open.")
        np.savetxt(save_file_name, data, delimiter=',')

def solve_ode_with_timestamps(particles):
    """Runs solve_ode (with argument particles) and prints timestamps at the start
    and end."""
    start_time = time.time()
    print("Start time: ",start_time)
    answer = solve_ode(particles)
    export_computation_to_csv(answer)
    finish_time = time.time()
    print("Finish time: ",finish_time, ", Difference: ", finish_time - start_time)

    return answer

def run():
    check_parameters()
    return solve_ode_with_timestamps(set_up_initial_conditions())
```