Rijksuniversiteit Groningen

Bachelor Thesis

# Simulating Galaxy Collisions in Python for Astronomy Education

*Author:*
Thijs Verkade (s3479838)

*Supervisor:*
Dr. Jake Noel-Storr

July 3, 2020

rijksuniversiteit
groningen

## Abstract

In this thesis, the process behind producing an application in python which simulates galaxy collisions is explained. The aim of the application is for it to be used by students in order to introduce them to the concepts of galaxy collisions in an educational manner. The application is based upon a java application called GalCrash, created by Chris Mihos. The application is tested by students and the educational value of the application is then discussed based on the feedback provided.

## Acknowledgements

I would like to thank Dr. Jake Noel-Storr for supplying the thesis topic, as well as for their support and guidance in producing the application, as well as organizing the evaluation of the application. I would also like to thank the group of students, along with the teaching assistant, who tested the application, and provided valuable feedback that was used to improve the application.

# Contents

# 1 Introduction

Studying the interactions between galaxies play a fundamental role in understanding how galaxies have evolved into the shapes and sizes that are present today. The colliding and merging of galaxies was a much more common process in the Early Universe, where astronomers found that very young galaxies often resembled starburst galaxies with multiple nuclei and peculiar shapes (Fraknoi et al., 2016). Merging galaxies tend to induce massive bursts of star formation and completely reshape the galaxies on the scale of millions of years. To be able to understand the properties of current galaxies and how these came about, it is thus extremely important to study these galaxy interactions to understand how these changes occur. As galaxy interactions occur over a timescale of millions of years, through use of computer simulations, one can study the whole process of a galaxy collision or a merger without having to live an eternity.

Roughly 20 years ago, such an application was created by Chris Mihos that could run these types of simulations. The application, named GalCrash, can simulate the interaction of two spiral galaxies. In this application, the user can change a few of the properties of the galaxies such as their relative masses, their separation, and the orientation of each galaxy's disk. By altering these properties, one can create a variety of different simulations that produce different results to show the impact that each of these parameters have on each respective galaxy and the resultant merger. This can then also be used to recreate collisions between real interacting galaxies that we can observe in the sky. The code for this application was written in Java, and is fairly outdated, while also lacking information in terms of how the code is written. It gives little information to the reader as to what processes are being carried out in the code and why this is being done.

The goal of this project, is to re-write the GalCrash application in Python, and do this in such a way that it can be used as an education tool to teach students about galaxy interactions. By updating the GalCrash application to be run in Python, the code also becomes more accessible to students studying Astronomy, as Python is a much more common language being taught to students studying Astronomy. In this way, students can still simulate galaxy interactions, while also gaining more insight into how such an application is created. By writing the code in such a way that the steps carried out are described, it becomes much easier for students to understand the purpose of each line of code. The graphical user interface (GUI) for the Python application written was created using Qt Designer, a tool for designing and building GUIs with widgets.

In this report, we will discuss the physics behind the galaxy interactions being simulated, as well as walk-through the coding behind the simulation step-by-step, explaining what processes were used to develop the final application. The educational value of the application will also be evaluated, looking at the accessibility of the application and whether or not students find it to be a valuable resource. Further improvements and features that could be added in the future will also be discussed.

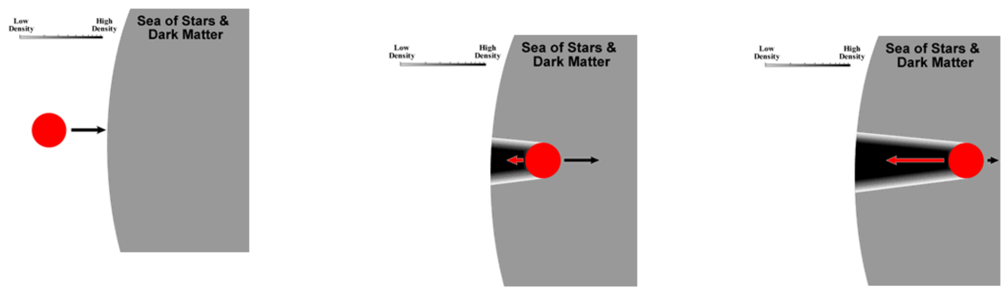
# 2 Theory

The simulations carried out in this project are done between two spiral galaxies. Spiral galaxies contain a disk of stars and gas arranged in a spiral pattern. Spiral galaxies usually contain a central bulge that resembles a small elliptical galaxy. The bulge consists of older stars while the disk is usually made up of a large range of ages, partly young stars (Gallagher & Sparke, 2007). According to a 2010 Hubble Space Telescope survey, around 70% of the galaxies observed were classified as spirals (Delgado-Serrano et al., 2010). Therefore, when looking at galaxy interactions, it makes sense to study the interaction between two spiral galaxies, as these have the highest likelihood to interact with one another due to the fact that they are so common.

The process of star formation within galaxies are linked to the growth of galaxies through galaxy interactions. The most common type of interaction between galaxies are that of a galaxy with a minor companion, due to the greater fractional abundance of low luminosity galaxies (Lambas et al., 2012). This is whats known as a minor merger. In this scenario, when the galaxies merge, the larger galaxy remains relatively unchanged, while the small galaxy is completely disrupted or its core comes to rest at the center of the larger galaxy.

A major merger occurs when the merging galaxies have similar masses. In such a case, due to the violently changing gravitational fields, the merger remnant does not resemble either of the original galaxies very well. In a major merger, the galaxies merge into a single steady-state system.

In both minor and major mergers, dynamical friction plays an important role in how galaxies collide. Dynamical friction is a pseudo-frictional force which occurs when a massive object moves through a sea of lower mass particles (Mihos et al., 1999). Due to the large gravitational force that the massive object is exerting on the lower mass particles, a high concentration of smaller particles start to form behind the massive object. This large concentration of small particles behind the massive object then exerts a collective gravitational force on the massive object, causing it to decelerate. This is illustrated in Figure 1 below.


Low Density
High Density
Sea of Stars & Dark Matter
Low Density
High Density
Sea of Stars & Dark Matter
Low Density
High Density
Sea of Stars & Dark Matter

(a) (b) (c)

Figure 1: (a), (b) and (c) show the effect of dynamical friction on a massive object due to the increasing density of the sea of stars and dark matter forming behind the object. The gray represents a low density region while the black represents a high density region. ("Dynamical Friction", n.d.)

Dynamical friction causes the galaxies to brake on their orbit and then merge. In our simulation, dynamical friction does not occur, due to the simplicity of our simulation. However, this friction can be analytically approximated using the Chandrasekhar formula. The Chandrasekhar formula states that galaxies should feel a frictional force due to dynamical friction proportional to the mass of the object times the density of the background sea divided by the velocity of the object squared (Mihos et al., 1999) (1).

$$F \propto \frac{M\rho}{v^2} \tag{1}$$

To understand this process, we look at the case of a minor merger. We consider a companion (small) galaxy with mass $M_c$, travelling through a population of field stars of individual mass $m_a$, where $m_a << M_c$. In this derivation, we assume that the companion galaxy acts as a point mass, even though we know that this is not the case. The field stars present are part of a much larger host galaxy of mass $M_g$ where $M_g >> M_c$. Here, the assumption is taken that the mass of the host galaxy is so much larger that it can be approximated as infinite and homogeneous. In such a scenario, then the dominant effect of the encounters is dynamical friction, which decelerates the companion galaxy at a rate of (Aceves & Colosimo, 2006):

$$\frac{dv_c}{dt} = -4\pi G^2 M_c m_a \ln\Lambda \int d^3 v_a f(v_a) \frac{\vec{v}_c - v_a}{|\vec{v}_c - v_a|^3} \tag{2}$$

Then, if the field stars have an isotropic velocity distribution, then equation 2 yields a simpler expression for the dynamical friction, namely Chandrasekhar's dynamical friction formula:

$$\frac{dv_c}{dt} = -16\pi^2 G^2 M_c m_a \ln\Lambda \left( \int_0^{v_c} dv_a v_a^2 f(v_a) \right) \frac{\vec{v}_c}{v_c^3} \tag{3}$$

In the case for which $v_c$ is sufficiently large, then our integral in 3 converges to a definite limit equal to the number density n divided by $4\pi$. This then provides us with the following formula for dynamical friction:

$$\frac{dv_c}{dt} = -4\pi G^2 M_c m_a n_a \ln\Lambda \frac{\vec{v}_c}{v_c^3} \tag{4}$$

This can then be written in terms of the density $\rho_a$ as follows,

$$\frac{dv_c}{dt} = -4\pi G^2 M_c \rho_a \ln\Lambda \frac{\vec{v}_c}{v_c^3} \tag{5}$$

This exact formula is not used in the simulation, but a very similar formula is used which mimics the Chandrasekhar expression is used. This expression can be found in the following section of the report.

# 3 Coding

## 3.1 Object-Oriented Programming

To generate the code of the new GalCrash application, a programming paradigm called Object-Oriented Programming (OOP) was used. OOP is an approach primarily used for modeling real-world things, where real-world entities are modeled as software objects. For this project, we have several different types of objects, each with their own properties. An example of an object present in this project is a star, which has the following properties: mass, position and velocity. For each such an object, as well as having properties, they also have functions which can be carried out, such as "MoveStar", which requires a time-step input, and will provide new positions and velocities so we can see the evolution of the properties of the star through time.

### 3.1.1 Classes

To be able to achieve this, we create whats called a class in Python. A class is essentially a blueprint for how something should be defined. Classes are generally used when you want to represent something much more complicated, with many different properties. Such classes can thus track properties about an object in an organized matter. Continuing with our example of stars, we created a class called star, in which we would like to know the mass of the star, its position in three dimensions, as well as its velocity in three dimensions. It is important to note that classes do not provide any real content themselves, and so even though we have defined specifications for a star, stating that a mass, position and velocity are necessary for defining a star, it will not actually state these properties.

### 3.1.2 Instances

With these classes, we use python objects called Instances. Simply put, instances are copies of the class with actual values. Instead of being a blueprint, an instance is now a fully realized object belonging to a specific class. You can create multiple instances, each with different values, but all containing the same information. This allows us to create multiple stars, each with their own unique properties, in such a manner that these properties can easily be called upon and used. Now that we understand the basics of OOP, and know what a class is and what an object of a class is, we must then look at how this is implemented in Python.

## 3.2 Implementing OOP in Python

### 3.2.1 Galaxy Class

To look at how we implement classes, let us take a look at an example from our application. In this particular example, we look at our Galaxy class, which includes several functions which alter our parameters.

```
class Galaxy:
    """A class used to define the initial parameters of a generated galaxy. It
    also contains functions that are used in calculating the evolution of the
    galaxy. Here we generally deal with the center of the galaxy when looking at
    parameters such as position."""
    def __init__(self,galmass,ahalo,vhalo,rthalo,galpos,galvel):
        self.galmass = galmass
        self.ahalo = ahalo
        self.vhalo = vhalo
        self.rthalo = rthalo
        self.galpos = galpos
        self.galvel = galvel
        self.galacc = np.full((3,1),0.)
```

Listing 1: Creating a class with attributes

The first thing we do when creating a class is to create attributes. Using the **init()** method, we specify an object's initial attributes and give them their default state. Our method has several attributes, as can be seen from Listing 1. galmass represents the galaxy mass, ahalo is the acceleration of the dark matter halo, vhalo is the velocity of the dark matter halo, rthalo is the radius of the dark matter halo, galpos and galvel are the three dimensional position and velocity of the galaxy respectively. Along with these attributes, a self variable is also required. The self variable refers to the object itself. The self variable is also an instance of our galaxy class, and instances of a class have varying values, allowing us to assign different values to different instances. The self variable helps keep track of individual instances of a class, allowing us to specify different parameters for each individual instance, such that we can have two galaxies with different properties.

Having now defined the basic blueprint for a galaxy, we can begin to perform operations with the attributes of our objects. To do this, we use so called instance methods, which is very similar to a function in Python. The first argument of an instance method is always self. A simple example of an instance method can be found in Listing 2. Looking at our instance method setPosVel, we can see that it simply changes the position and the velocity of the galaxy depending on the input. This particular instance is only used when initializing our system, as it gives the inital positions and velocities of the particular galaxy.

```
    def setPosvel(self,pos,vel):
        self.galpos = pos
        self.galvel = vel
```

Listing 2: Creating Instance Methods to perform operations on our galaxy

The next instance method used is called scaleMass and can be found in Listing 3. This instance is used to define the parameters of the companion galaxy, based on the input which is the ratio of the mass of the companion galaxy to the main galaxy. Both galaxies are initially created with the default parameters from Chris Mihos' java version. So essentially, we start by creating two identical galaxies with the same properties. Then, based on the input of the ratio of masses given by the user, we scale the parameters of the companion galaxy based on the scaleMass instance method. If the input of the mass ratio is given as 1 (default), then all parameters remain identical. Otherwise, the mass, ahalo, vhalo and rthalo all change according to the formula provided in the code.

```
    def scaleMass(self,massFact):
        """The Scale Mass Function calculates the parameters of the companion
    galaxy. This is done by taking the ratio of the mass of the companion galaxy
    to the main galaxy as the input."""
        self.galmass = self.galmass*massFact
        self.vhalo = 1.0*massFact**0.25
        self.ahalo = 0.1*massFact**0.5
        a2 = -self.galmass/(self.vhalo**2)
        a1 = -2.*self.ahalo*self.galmass/(self.vhalo**2)
        a0 = -self.galmass*(self.ahalo**2)/(self.vhalo**2)
        q = a1/3.0 - (a2**2)/9.0
        r = (a1*a2-3.*a0)/6.0 - (a2**3)/27.0

        s1 = (r + np.sqrt((q**3)+(r**2)))**0.333
        s2 = ((r - np.sqrt((q**3)+(r**2)))**0.333)

        self.rthalo=(s1+s2)-a2/3
```

Listing 3: Instance Method "Scale Mass" used to scale galaxy parameters based on user input

Another instance method used is namely the MoveGalaxy method shown in listing 4. As the name suggests, the MoveGalaxy method simply advances our galaxy forward in a given timestep dt. To be able to do this, the galaxies position, velocity and acceleration must be known. The positions and velocities of the galaxies are well known as these are attributes of the galaxy. The acceleration, however, needs to be calculated.

```
    def MoveGalaxy(self,dtime):
        """Move Galaxy evolves the position and velocity of the galaxy with input
    dtime"""
        newpos = self.galpos + self.galvel * dtime + 0.5 * self.galacc *(dtime**2)
        newvel = self.galvel + self.galacc * dtime

        self.galpos = newpos;
        self.galvel = newvel;
```

Listing 4: Instance method "Move Galaxy" which evolves the parameters of the galaxy as time progresses"

The acceleration can be calculated using Newton's well-known law of universal gravitation, in which we have that an object at a distance r from an object of mass M will fell an acceleration due to gravity equivalent to:

$$g = \frac{-GM}{r^2} \tag{6}$$

Using this method, we are treating our galaxies as if they have fixed shapes, when in reality, the shape of the galaxy is constantly changing as it interacts gravitationally with the other galaxy. If we wanted to calculate the acceleration in a self-consistent matter, then we would need to calculate all the gravitational forces between all the stars and dark matter particles which make up the galaxies, which is extremely expensive computationally, and therefore not feasible for a simulation of this nature. Having said this however, although we sacrifice some accuracy, using the Chandrasekhar dynamical friction formula (discussed in previous section), we can allow the galaxies to merge in a way that more closely resembles reality. We will come back to this concept later.

To calculate the acceleration, we have thus introduced an instance method with an input posin (see listing 5. posin is the position of the interacting object. For example, if we want to calculate the acceleration of our main galaxy M with respect to our companion galaxy C, then we would use the position of C as an input. From here, our method calculates the difference in position between our two galaxies, and uses this to calculate the gravitational acceleration from equation 6. In this method, we use a different instance method to calculate the mass called InteriorMass. How the interior mass is calculated within this instance method is discussed in the next paragraph. The next instance method calculates the gravitational potential energy (Listing 6) in a very similar manner to how the acceleration is calculated, the only difference being the formula for the gravitational potential 7

$$U = -\frac{GM}{r} \tag{7}$$

```python

    def Acceleration(self,posin):
        """Acceleration function takes the position of the object (star/galaxy) as
     input and returns the acceleration of the object."""
        G = 1.0
        dpos = posin - self.galpos
        #dx = dpos[0]
        #dy = dpos[1]
        #dz = dpos[2]
        r = np.sqrt(np.sum(dpos**2, axis = 0))
        #r = np.sqrt((dx**2)+(dy**2)+(dz**2))
        AccMag= -(G*self.InteriorMass(r))/(r**2)
        calcacc = (dpos*AccMag)/r

        return calcacc
```

Listing 5: Instance method "Acceleration" used to calculate acceleration of galaxy

```python
    def Potential(self, posin):
        """Potential calculates the potential of the object (star/galaxy) based on
     the input position given."""
        G=1.0;

        dpos = posin - self.galpos
        #dx = dpos[0]
        #dy = dpos[1]
        #dz = dpos[2]
        r = np.sqrt(np.sum(dpos**2, axis = 0))
        #r = np.sqrt((dx**2)+(dy**2)+(dz**2))
        pot= G*self.InteriorMass(r)/r

        return pot
```

Listing 6: Instance method "Potential" used to calculate the potential energy of our galaxy

The next instance method has already been mentioned previously, namely the InteriorMass instance method (Listing 7. This method returns the interior mass of the object dependent on the input r. r is also used in the potential and acceleration methods and is defined as the magnitude of the differential vector of the position of the star/galaxy with respect to the galaxy being called upon. From here, we simply have two formulas for calculating the interior mass, dependent on whether or not the magnitude r lies within the dark matter halo radius of the galaxy or not. If we consider the main galaxy M and the companion galaxy C again, if we wish to calculate the interior mass of our main galaxy, then r is the magnitude of the difference in position between C and M. Then, if this magnitude is less than the dark matter halo radius of M, this means that our companion galaxy lies within the dark matter halo radius of M. If this is the case, then the interior mass of our galaxy M is calculated using equation 8.

$$M_I = \frac{v_H^2 \times r^3}{(a_H + r)^2} \tag{8}$$

Otherwise, if we have that r is greater than the dark matter halo radius of our galaxy M, then we have that the companion galaxy is not close enough to significantly impact the interior mass, and the interior mass is then simply given by the mass of the galaxy M.

```python

    def InteriorMass(self, r):
        """Interior Mass returns the mass of the object (star/galaxy) based on its
     position in relation to the center of the galaxy"""


        indices = r < self.rthalo


        intmass = np.full(r.shape, 0.)

        if intmass[indices].shape != (0,):
        #If r< self.rthalo, then the interacting object (star/galaxy) is confined
    within the dark matter halo radius of the galaxy, and the interior mass
        # is calculated as follows
            intmass[indices] = (self.vhalo**2)*(r[indices]**3)/((self.ahalo+r[
    indices])**2)
        if intmass[~indices].shape != (0,):
        #Otherwise, the interior mass is simply the mass of the galaxy
            intmass[~indices] = self.galmass

        return intmass
```

Listing 7: Instance method "Interior Mass" used to calculate the mass of the interacting object

Next, we have an instance method called Density, which calculates the density of the surrounding area of the galaxy (the density of the area it is moving through). This density is then used to calculate the Dynamical Friction. The density instance method can be found in listing 8

```python
    def Density(self, r):
        """Determines the density based on the input r which is the position of
    the object in relation to the center of the galaxy."""
        rinner = r*0.99
        router = r*1.01
        minner = self.InteriorMass(r*0.99)
        mouter = self.InteriorMass(r*1.01)
        dm = (mouter-minner)
        vol=(4/3)*np.pi*((router**3)-(rinner**3))
        density=dm/vol

        return density
```

Listing 8: Instance method "Density" used to calculate the density of the surrounding medium being traversed

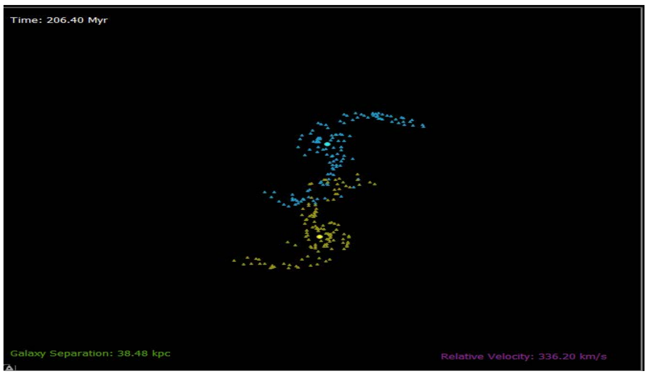
The last instance method we use in the Galaxy class is called DynFric (Listing 9). Here, an estimate is calculated of dynamical friction, through use of a slightly altered version of the Chandrasekhar formula given in equation 1. The friction force is then calculated using equation 9. In this equation, v represents the differential vector between the speeds of two objects, while dv represents the magnitude of v. It is also important to note that in this altered version of the Chandrasekhar formula, we use the following values G=1.0, and $\ln\Lambda = 3.0$

$$f = -\frac{4\pi G \ln \Lambda M \rho dv}{(1+v)^3} \tag{9}$$

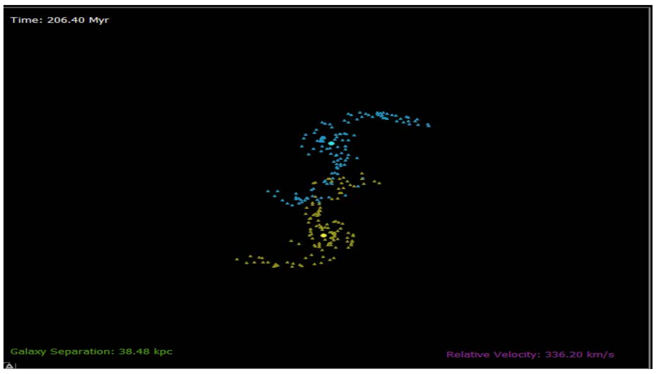
```
    def DynFric(self,pmass,ppos,pvel):
        """Dynamic Friction is used when considering friction in our model. It
    calculates the friction based on the mass position and velocity of the object.
     The resultant friction contributes to the overall acceleration of that object
    """

        G=1.0
        lnGamma=3.0
        dv = pvel - self.galvel
        v = np.linalg.norm(dv)
        dr = ppos - self.galpos
        r = np.linalg.norm(dr)
        galrho = self.Density(r)
        fricmag = 4.0*np.pi*G*lnGamma*pmass*galrho*v/((1+v)**3)
        friction = (-dv/v)*fricmag

        return friction
```

Listing 9: Instance method "DynFric" used to estimate Dynamical Friction

In figure 2, we can see the effects of dynamical friction, and how it affects the results of the simulation. In the figure, we see two instances of our simulation, one with dynamical friction (b) and one without (a). (If viewed digitally, the still image is replaced with an animation). These two images show the importance of dynamical friction in causing the galaxies to merge. In (a), we see that the galaxies have extended tidal tails, but the two galaxies are not being pulled together and will only be lightly disrupted. In (b), we see that the two galaxies are pulled towards each other, and the galactic centers start moving towards each other, and the galaxies merge into one larger galaxy.



(a)



(b)

Figure 2: (a) Simulation at around 200 Myr without dynamical friction, (b) Simulation at around 200 Myr with dynamical friction enabled. In digital versions, click to play the videos.

### 3.2.2 StarGalaxy Class

Along with the Galaxy class, two other classes are also present in the code for our simulation. The first of which is called StarGalaxy. The StarGalaxy class is a subclass of our galaxy class. A subclass inherits all of the attributes and methods associated with the parent class, while it can also have attributes and methods of its own. This can be seen by looking at listing 10. The super() function is used to give access to the methods and attributes of the Galaxy class to our new StarGalaxy class. We call all the attributes from the Galaxy class again, as can be seen in our initialize statement, as well as several new ones, diskSize, galtheta, galphi, and n.

```
class StarGalaxy(Galaxy):
    """A sub-class of galaxy, used to combine information from the stars and the
    galactic center to produce a galaxy with orbiting stars."""
    def __init__(self,galmass,ahalo,vhalo,rthalo,galpos,galvel,diskSize,galtheta,
    galphi,n):
        super().__init__(galmass,ahalo,vhalo,rthalo,galpos,galvel)
        self.diskSize = diskSize
        self.galtheta = galtheta
        self.galphi = galphi
        self.n = n

        #Define the star position, velocity, and acceleration in such a way that
    they can be called upon later more easily.
        self.starpos = np.full((3,self.n), 0.)
        self.starvel = np.full((3,self.n), 0.)
        self.staracc = np.full((3,self.n), 0.)
```

Listing 10: Creating the "StarGalaxy" class with inherited attributes

Most of these new attributes are parameters that can be altered by the user. Disksize is a fixed value which can not be altered by the user, and represents the radius in which stars are present within the galaxy. Galtheta and galphi give the orientation of the galaxy, more specifically, the inclination and slew of the galaxy in the orbital plane. The effects of changing the inclination and slew of a galaxy can be seen in figure 9. n represents the amount of stars being simulated. The stars are then evenly split between the two galaxies such that each galaxy has 0.5n stars.

At the end of listing 10, we also create star templates, in which we create n stars, each with the position velocity and acceleration set to 0. These will then be initialized in a later instance method.

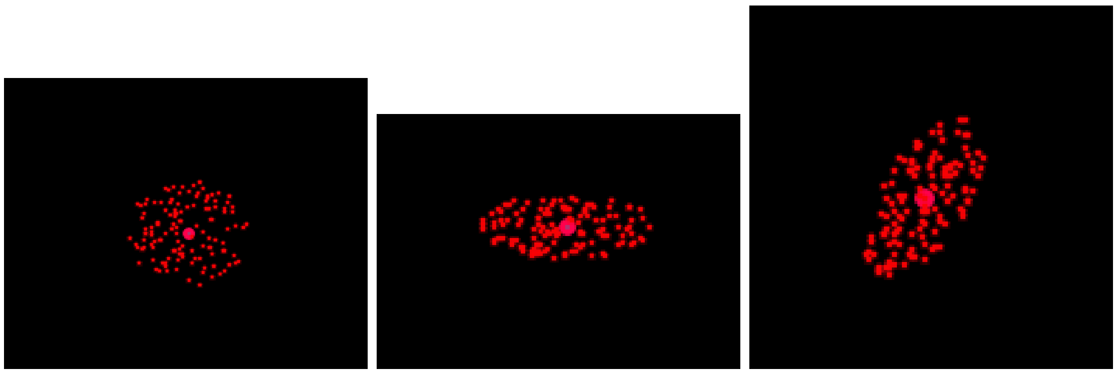


Figure 3: (a) Inclination and Slew of red galaxy are both 0°, (b) Inclination of red galaxy changed to 90°, (c) Inclination and Slew of red galaxy are both 90°

Now that we have gone over the new attributes defined in StarGalaxy, we can begin to look at what new instance methods are defined in this class. The StarGalaxy class contains an instance method called "MoveStars". The "MoveStars" method operates exactly the same as the "MoveGalaxy" method, with the only difference being that we are replacing the galaxy's position, velocity and acceleration with the star's position, velocity and acceleration. This instance method can be found in listing 11.

```
    def MoveStars(self,dtime):
        """Function for moving a star over time by calculating the new position and velocity"""
        newstarpos =  self.starpos + self.starvel * dtime + 0.5 * self.staracc * (dtime**2)
        newstarvel = self.starvel + self.staracc * dtime

        self.starpos = newstarpos
        self.starvel = newstarvel
```

Listing 11: Defining a new "MoveStars" method for our StarGalaxy class

The next instance method present in our StarGalaxy class is the InitStars method. This method initializes the position and velocity of the stars within the galaxy and can be found in listing 12.

```
    def InitStars(self):
        """InitStars initializes the stars in the galaxy. It generates n number of stars and gives them random positions within the disksize of the galaxy. The velocities of the stars are also calculated based on the positions as well as the input angles phi and theta"""
        cosphi = np.cos(self.galphi)
        sinphi = np.sin(self.galphi)
        costheta = np.cos(self.galtheta)
        sintheta = np.sin(self.galtheta)
        for i in range(self.n):
            bad = True
        while bad:
            xtry = self.diskSize*(1.-2.*np.random.random())
            ytry = self.diskSize*(1.-2.*np.random.random())
            rtry = np.sqrt(xtry**2+ytry**2)
            if (rtry < self.diskSize): bad = False

        ztry = 0.0
        xrot = xtry*cosphi + ytry*sinphi*costheta + ztry*sinphi*sintheta
        yrot = -xtry*sinphi + ytry*cosphi*costheta + ztry*cosphi*sintheta
        zrot = -ytry*sintheta + ztry*costheta
        rot = np.array([xrot,yrot,zrot])
        self.starpos[:,i] = rot + self.galpos.reshape(-1)

        vcirc = np.sqrt(self.InteriorMass(rtry)/rtry)

        vxtry = -vcirc*ytry/rtry
        vytry = vcirc*xtry/rtry
        vztry = 0.0

        vxrot = vxtry*cosphi + vytry*sinphi*costheta + vztry*sinphi*sintheta
        vyrot = -vxtry*sinphi + vytry*cosphi*costheta + vztry*cosphi*sintheta
        vzrot = -vytry*sintheta + vztry*costheta

        vrot = np.array([vxrot,vyrot,vzrot])
        self.starvel[:,i] = vrot + self.galvel.reshape(-1)
        self.staracc = np.full((1,3),0.)
```

Listing 12: Instance method "InitStars" used to initialize the positions and velocities of the stars within each galaxy

An initial position of the star is first taken by taking the diskSize attribute and multiplying it by a random number between -1 and 1. This is done for both the position in x as well as the position in y. The position in z is always simply set to 0. From here, we calculate the magnitude of our position vector. If the magnitude of the position vector of the star lies within the diskSize attribute, then we take this as our initial position. Otherwise, if the magnitude of our position vector lies outside our diskSize attribute, then the process is simply repeated until the star lies within it. From here, we work with our galtheta and galphi attributes to further workout the position of our stars depending on the orientation of our galaxy. The position of the star changes dependent on the rotation undergone due to the inclination and slew angle. The expressions for this can be found in equations 10 - 12.The derivation of these formulas can be found in Appendix A.

$$x_r = x\cos\phi + y\sin\phi\cos\theta + z\sin\phi\sin\theta \tag{10}$$

$$y_r = -x\sin\phi + y\cos\phi\cos\theta + z\cos\phi\sin\theta \tag{11}$$

$$z_r = -y\sin\theta + z\cos\theta \tag{12}$$

After accounting for the rotations of our axes, we then simply add these positions to that of the galaxy to obtain the positions of the stars within the galaxy. Next, we want to give these stars an initial velocity. The first step to this is to calculate an estimate of the escape velocity of the stars. This is done using an approximation of the escape velocity equation (equation 13). In this equation, M is the interior mass which we calculate using the instance method defined in the Galaxy Class. In our Galaxy class, we have also scaled all our equations such that we take the gravitational constant to be equal to 1. Therefore, we can simply exclude it from our equation. Furthermore, in the escape velocity equation, there is normally a factor 2 present in the numerator, which we do not include in our estimation of the escape velocity.

$$v_{esc} = \sqrt{\frac{GM}{r}} = \sqrt{\frac{M}{r}} \tag{13}$$

From here, we simply calculate velocities in x and y using equations 14 - 15. We assign the velocity in the z direction to be equal to 0.

$$v_x = \frac{-v_{esc}y}{\sqrt{x^2+y^2}} \tag{14}$$

$$v_y = \frac{v_{esc}x}{\sqrt{x^2+y^2}} \tag{15}$$

From here, we use equations 10 - 12 to once again, account for the rotation of our axes, this time replacing our position vector with the velocity vector. We then add our new velocity vector of our star with the velocity vector of our galaxy to obtain an initial velocity vector for a star within the galaxy. The initial acceleration of the stars is simply set to 0 for the time being.

The last new instance method present in our StarGalaxy class is another method called "Scale-Mass" which can be found in listing 13. This new method simply multiplies the disk size of our companion galaxy by the square root of the mass ratio between the companion galaxy and the main galaxy. It also contains the super() function, and inherits all the attributes associated with the ScaleMass function present in the Galaxy class.

```
    def scaleMass(self, massFact):
        """The function scalemass calculates the disk size of a companion galaxy
    based on the mass ratio of the companion galaxy to the main galaxy"""
        self.diskSize = self.diskSize*np.sqrt(massFact)
        super().scaleMass(massFact)
```

Listing 13: Instance method "ScaleMass" used to calculate a new diskSize

### 3.2.3 Orbit Class

The final class that we define is the orbit class. The orbit class calculates the initial position and velocity of the two galaxies based on the input masses of the galaxies. In our simulation, a parabolic orbit is assumed. The attributes used in the Orbit class are energy, rp & tp, which give the distances of closest approach, eccentricity, m1 & m2, which represent the masses of the galaxies, and the bod1pos/bod2pos as well as bod1vel/bod2vel which represent the position and velocities of each respective galaxy. For a parabolic orbit, we have that the energy of the trajectory is 0, and the eccentricity is 1. We then calculate the initial positions and velocities of the galaxies assuming a parabolic orbit as can be seen in listing 14.

```
class Orbit:
    """The Orbit class calculates initial position and velocity of the two
    galaxies based on a parabolic orbit"""
    def __init__(self,energy,rp,tp,eccentricity,m1,m2,bod1pos,bod2pos,bod1vel,
    bod2vel):
        self.energy = energy
        self.rp = rp
        self.tp = tp
        self.eccentricity = eccentricity
        self.m1 = m1
        self.m2 = m2
        self.bod1pos = bod1pos
        self.bod2pos = bod2pos
        self.bod1vel = bod1vel
        self.bod2vel = bod2vel
        self.initOrbit()



    def initOrbit(self):
        """InitOrbit initializes the orbit of the two galaxies based on the input
    masses of the galaxies
        This function assumes a parabolic orbit and returns the position and
    velocity of both galaxies"""
        #Parabolic Orbit
        mu = self.m1 + self.m2

        p = 2*self.rp
        nhat = np.sqrt(mu/(p**3))
        cots = 3.0 * nhat * self.tp
        s = np.arctan(1.0/cots)
        cottheta = (1./(np.tan(s/2.)))**0.3333
        theta = np.arctan(1./cottheta)
        tanfon2 = 2./np.tan(2.*theta)
        r = (p/2.)*(1+tanfon2**2)


        vel = np.sqrt(2.*mu/r)
        sinsqphi = p/(2.*r)
        phi = np.arcsin(np.sqrt(sinsqphi))
        f = 2.*np.arctan(tanfon2)
        xc = -r*np.cos(f)
        yc = r*np.sin(f)
        vxc = vel*np.cos(f+phi)
        vyc = -vel*np.sin(f+phi)
        xcom = self.m2 * xc/(self.m1+self.m2)
        ycom = self.m2 * yc/(self.m1+self.m2)
        vxcom = self.m2 * vxc/(self.m1 + self.m2)
        vycom = self.m2 * vyc /(self.m1+self.m2)

        self.bod1pos = np.array([[-xcom],[-ycom],[0.0]])
        self.bod1vel = np.array([[-vxcom],[-vycom],[0.0]])
        self.bod2pos = np.array([[xc-xcom],[yc-ycom],[0.0]])
        self.bod2vel = np.array([[vxc-vxcom],[vyc-vycom],[0.0]])
```

Listing 14: Initializing our "Orbit" class and the instance method "initOrbit" which calculates the initial positions and velocities of two galaxies

## 3.3 Creating the Graphical User Interface

Now that we have established all our classes and methods that will simulate the galaxy interaction, we must create a Graphical User Interface (GUI). The purpose of the GUI is to provide the user an easy way to interact with the free parameters, as well as visually see how the galaxies and stars interact with one another. Furthermore, the GUI is also used to provide important information such as the time of the simulation, the separation of the galaxies at that time as well as the relative velocity of the galaxies. Creating the User Interface is quite simple. With the help of an application called Qt Designer, one can simply drag and drop widgets on a blank canvas, and arrange them however they see fit. An blank canvas, along with the available widgets can be found in figure 4.

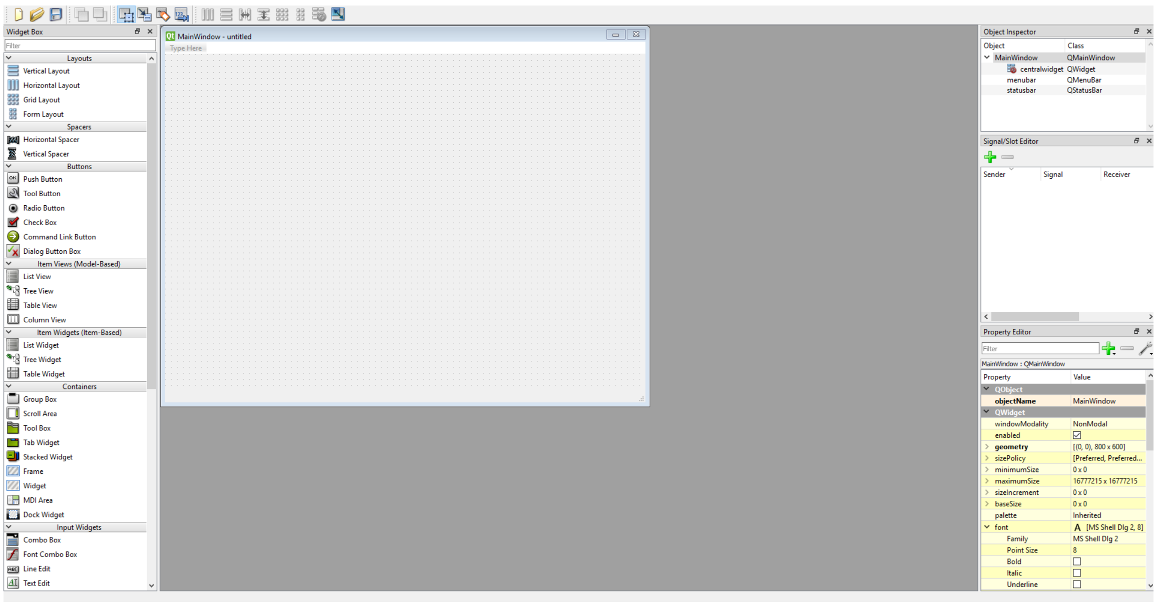
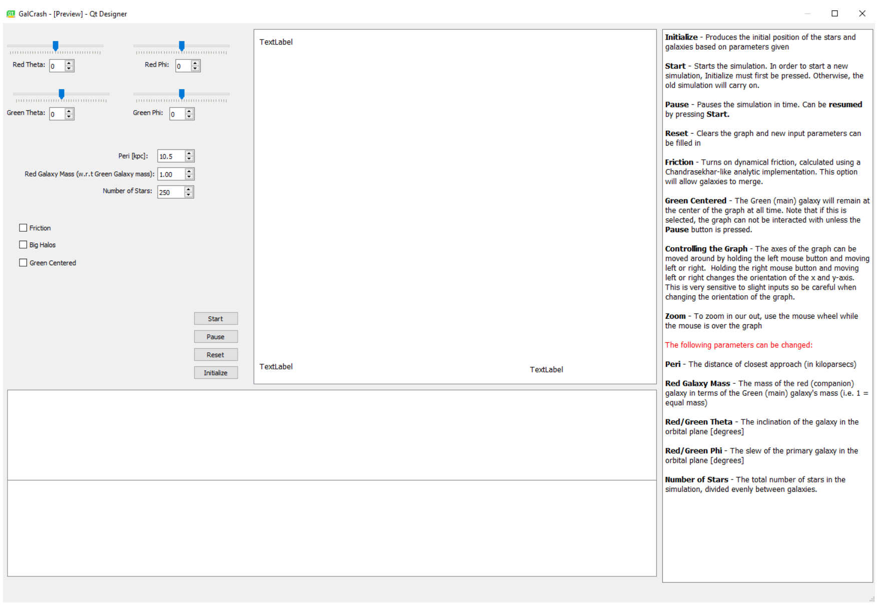


Figure 4: A blank canvas in the Qt Designer Application

Using Qt Designer, to design the widgets, and then launching the application, we obtain the GUI seen in figure 5



**Initialize** - Produces the initial position of the stars and galaxies based on parameters given

**Start** - Starts the simulation. In order to start a new simulation, Initialize must first be pressed. Otherwise, the old simulation will carry on.

**Pause** - Pauses the simulation in time. Can be **resumed** by pressing **Start.**

**Reset** - Clears the graph and new input parameters can be filled in

**Friction** - Turns on dynamical friction, calculated using a Chandrasekhar-like analytic implementation. This option will allow galaxies to merge.

**Green Centered** - The Green (main) galaxy will remain at the center of the graph at all time. Note that if this is selected, the graph can not be interacted with unless the **Pause** button is pressed.

**Controlling the Graph** - The axes of the graph can be moved around by holding the left mouse button and moving left or right. Holding the right mouse button and moving left or right changes the orientation of the x and y-axis. This is very sensitive to slight inputs so be careful when changing the orientation of the graph.

**Zoom** - To zoom in our out, use the mouse wheel while the mouse is over the graph

The following parameters can be changed:

**Peri** - The distance of closest approach (in kiloparsecs)

**Red Galaxy Mass** - The mass of the red (companion) galaxy in terms of the Green (main) galaxy's mass (i.e. 1 = equal mass)

**Red/Green Theta** - The inclination of the galaxy in the orbital plane [degrees]

**Red/Green Phi** - The slew of the primary galaxy in the orbital plane [degrees]

**Number of Stars** - The total number of stars in the simulation, divided evenly between galaxies.

Figure 5: The designed GUI for the GalCrash Python application

## 3.4 Combining the code with the GUI

Having created the GUI, we now need to link the GUI with the code such that pushing a button, or checking a box, actually has an impact. We need to write code which communicates between our GUI and our classes such that we can send inputs from our GUI to use this information for our parameters. The code should then calculate the relevant information about the stars and galaxies, and then send this information back to the GUI so that it can be presented to the user in an understandable way. To do this, we create a class called "GUI", which is created as a subclass of "QMainWindow". The QMainWindow widget is a built-in class from PyQt which we will use to link our widgets with our other classes.

```
   class GUI(QMainWindow):
 2     def __init__(self):
           super(GUI, self).__init__()
 4         uic.loadUi("GalCrashbeta.ui",self)
           self.show()
 6         #When the initialize button is clicked, we create the galaxy and stars as
       well as give an orbit using the functions MakeGalaxy and MakeOrbit
           self.buttoninit.clicked.connect(self.MakeGalaxy)
 8         self.buttoninit.clicked.connect(self.MakeOrbit)

10
           #When the start button is pressed, we start seeing an animated graph in
       which we see how the galaxy and stars evolve over time and how they collide
12         self.buttonstart.clicked.connect(self.update_plot)

14         #The reset button clears the graph and new parameters can be selected and
       the simulation can be run again
           self.restartbutton.clicked.connect(self.reset)
16         self.pausebutton.clicked.connect(self.pause)

18         self.time = 0.00
           #Creating lists for which values are appended to when update plot is
       active
20         #These are then used to create the plots of the Galaxy Seperation and
       Relative Velocity over Time
           # Cleared every time reset is pressed
22         self.Dist = []
           self.Vel = []
24         self.Time = []
```

Listing 15: Creating a GUI class as a subclass of QMainWindow

### 3.4.1 Initializing the galaxies and stars

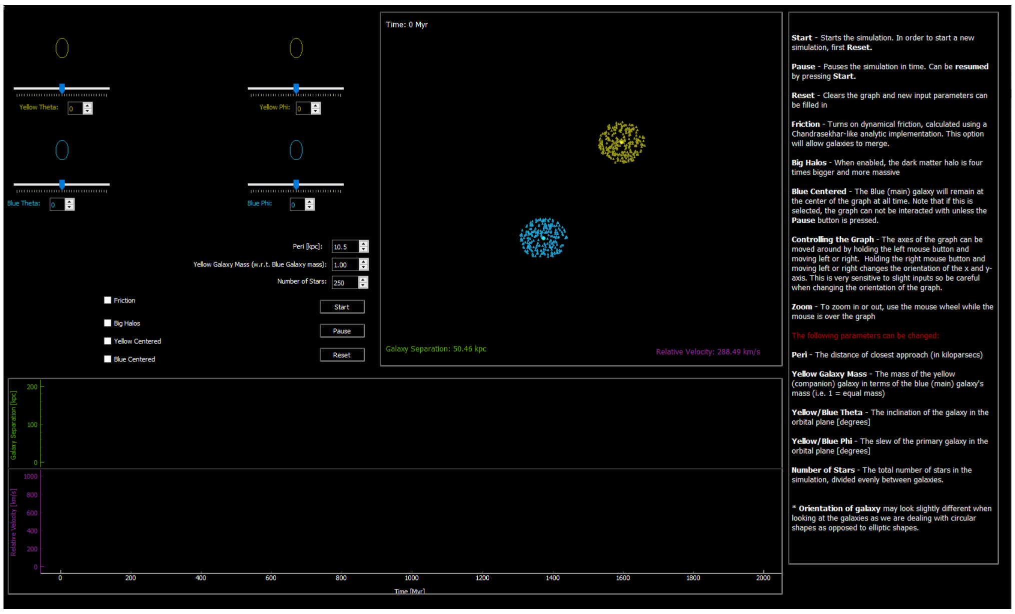
The first thing we do when creating our new GUI class is to load our GUI that we just built. This will then load the general layout of our GUI as well as all the widgets we have created. From here, we can start to send signals from the GUI to an instance method. Take for example, our Initialize button, the code in lines 6-8 of listing 15 essentially say that when the initialize button is pressed by the user, the "MakeGalaxy", as well as the "MakeOrbit" instance method should be carried out. This is done for all the other buttons as can be seen in the following lines of code.

Furthermore, we also set some initial variables at the top which are defined every time the program is run. self.time is set to zero when the application is started and will be advanced in the update plot function. self.Dist, self.Vel and self.Time are lists which will be filled with the galaxy seperation, relative velocity of the galaxies and the time so that this information can be plotted alongside the simulation.

Having connected all our buttons to instance methods, the next step is to set-up our graphs as well as our text labels. This is just a matter of defining axis limits and labelling our axes. The code for this can be found in listing 16. When this is done, our GUI looks ready to start presenting information (see figure 6).

```
#graphicsView is the graph in which the galaxies are displayed
        self.graphicsView.setXRange(-20,20)
        self.graphicsView.setYRange(-20,20)
        self.graphicsView.hideAxis('bottom')
        self.graphicsView.hideAxis('left')

        #velplot is a plot of the relative velocities over time
        self.velplot.setXRange(0,2000)
        self.velplot.setYRange(0,1000)
        self.velplot.setLabel('bottom', 'Time [Myr]')
        self.velplot.setLabel('left', 'Relative Velocity [km/s]')

        #Create different colors which are used for the axes
        #mkPen outlines the item in the proposed color
        #mkBrush fills the item in the proposed color

        #Yellow
        self.pen1 = pg.mkPen(color = (173,171,2))
        self.brush1 = pg.mkBrush('y')

        #Purple/Pink
        self.pen2 = pg.mkPen(color = (155,41,163))

        #White
        self.pen3 = pg.mkPen(color = (255,255,255))

        #Used for green text
        self.pen4 = pg.mkPen(color = (85,170,0))
        #Used for red text
        self.pen5 = pg.mkPen(color = (184,6,0))

        #light blue
        self.pen6 = pg.mkPen(color =(28,176,217))
        self.brush6 = pg.mkBrush(color = (3,252,252))

        #Changing axis colors
        self.velplot.plotItem.getAxis('left').setPen(self.pen2)
        self.velplot.plotItem.getAxis('bottom').setPen(self.pen3)


        #distplot is a plot of the galaxy separation over time
        self.distplot.setXRange(0,2000)
        self.distplot.setYRange(0,200)
        #Plotted above the velocity plot so we hide the x-axis as this is common
    between the two
        self.distplot.hideAxis('bottom')
        self.distplot.setLabel('left','Galaxy Separation [kpc]')
        self.distplot.plotItem.getAxis('left').setPen(self.pen4)

        #Add labels to our graphicsView plot which give the Time, Galaxy
    Separation, and Relative Velocity
        self.distlabel.setStyleSheet("color: rgb(85,170,0)")
        self.distlabel.setText("Galaxy Separation:")
        self.vellabel.setStyleSheet("color: rgb(155,41,163)")
        self.vellabel.setText("Relative Velocity:")
        self.timelabel.setStyleSheet("color: white")
        self.timelabel.setText("Time:")
```

Listing 16: Setting up the graphs and defining text labels



Peri [kpc]: 10.5
Yellow Galaxy Mass (w.r.t. Blue Galaxy mass): 1.00
Number of Stars: 250

Time: 0 Myr
Galaxy Separation: 50.46 kpc
Relative Velocity: 288.49 km/s

**Start** - Starts the simulation. In order to start a new simulation, first **Reset.**

**Pause** - Pauses the simulation in time. Can be **resumed** by pressing **Start.**

**Reset** - Clears the graph and new input parameters can be filled in

**Friction** - Turns on dynamical friction, calculated using a Chandrasekhar-like analytic implementation. This option will allow galaxies to merge.

**Big Halos** - When enabled, the dark matter halo is four times bigger and more massive

**Blue Centered** - The Blue (main) galaxy will remain at the center of the graph at all time. Note that if this is selected, the graph can not be interacted with unless the **Pause** button is pressed.

**Controlling the Graph** - The axes of the graph can be moved around by holding the left mouse button and moving left or right. Holding the right mouse button and moving left or right changes the orientation of the x and y-axis. This is very sensitive to slight inputs so be careful when changing the orientation of the graph.

**Zoom** - To zoom in or out, use the mouse wheel while the mouse is over the graph

The following parameters can be changed:

**Peri** - The distance of closest approach (in kiloparsecs)

**Yellow Galaxy Mass** - The mass of the yellow (companion) galaxy in terms of the blue (main) galaxy's mass (i.e. 1 = equal mass)

**Yellow/Blue Theta** - The inclination of the galaxy in the orbital plane [degrees]

**Yellow/Blue Phi** - The slew of the primary galaxy in the orbital plane [degrees]

**Number of Stars** - The total number of stars in the simulation, divided evenly between galaxies.

* **Orientation of galaxy** may look slightly different when looking at the galaxies as we are dealing with circular shapes as opposed to elliptic shapes.

Figure 6: An image of the GUI after defining axis parameters and labels

Now that we have set-up the GUI to look the way we want it, we need to examine the instance methods that are called upon when certain buttons are pressed, to see exactly how these call upon our main code to simulate the galaxy interactions. The first button that is pressed when the user wishes to run the code after filling in all their desired parameters is the Initialize button. Pressing this button calls upon two methods, the first of which is called "MakeGalaxy". The purpose of MakeGalaxy, as the name suggests is two create two galaxies as objects part of our StarGalaxy class. The code for this can be found in listing 17.

```
    def MakeGalaxy(self):
        """The function MakeGalaxy creates two galaxies based on the parameters
    provided in the UI"""
        #gal represents the main galaxy while comp represents the companion galaxy
        #First some general parameters taken from Chris Mihos' version
        galmass = 4.8
        ahalo = 0.1
        vhalo = 1.0
        rthalo = 5.0
        galpos = np.full((3,1),0)
        galvel = np.full((3,1),0)
        diskSize = 2.5

        #The following values are given by the inputs the user has provided
        galtheta = int(self.greenThetabox.value())
        galphi = int(self.greenPhibox.value())
        comptheta = int(self.redThetabox.value())
        compphi = int(self.redphibox.value())
        galn = int(0.5 * int(self.starsnum.value()))
        compn = int(0.5 * int(self.starsnum.value()))
        self.gal = StarGalaxy(galmass,ahalo,vhalo,rthalo,galpos,galvel,diskSize,
    galtheta,galphi,galn)
        self.comp = StarGalaxy(galmass,ahalo,vhalo,rthalo,galpos,galvel,diskSize,
    comptheta,compphi,compn)

        #If big halos is enabled, then our parameters are slightly different
        if self.checkbighalo.isChecked():
            self.gal.rthalo = 20.0
            self.gal.galmass = (self.gal.vhalo**2 * self.gal.rthalo**3)/((self.gal
    .ahalo+self.gal.rthalo)**2)
        else:
            self.gal.galmass = 4.8
            self.gal.rthalo = 5.0

        #Mass ratio is provided by the user
        massrat = float(self.massratio.value())

        #Using the mass ratio, we can scale the parameters of the companion galaxy
     according to the function scaleMass
        self.comp.scaleMass(massrat)

        #If big halos is enabled, then our parameters are slightly different
        if self.checkbighalo.isChecked():
            self.comp.rthalo = self.comp.rthalo*4.0
            self.comp.galmass = (self.comp.vhalo**2 * self.comp.rthalo**3)/((self.
    comp.ahalo+self.comp.rthalo)**2)
```

Listing 17: Instance Method "MakeGalaxy" which creates the two galaxies used in the simulation based on the parameters given by the user

As can be seen from listing 17, the first thing we do is to provide some default parameters. Note that we also set the initial positions and velocities of the galaxies to 0. The actual positions and velocities of the galaxies are calculated in our "MakeOrbit" method. To create the two galaxies, we take the users input for the inclination and slew of both galaxies, as well as the number of stars in each galaxy. From here, we create two galaxies with identical parameters, the only difference being the angles provided.

Next, we check if the user has opted for big halos. If the big halos check box is ticked, then the dark matter halos are four times bigger and more massive. The next step is then to scale the mass of the companion galaxy, depending on what the user gave as a mass ratio. This is done through our scaleMass method which we defined in our stargalaxy class. After we have scaled the mass of our galaxy, we will then also apply the necessary changes to the dark matter halos if the user has chosen for this. From here, we now have two galaxies with all our desired initial parameters, except for their initial positions and velocities. This is where our next method, "MakeOrbit", comes into play (see listing 18).

```
def MakeOrbit(self):
        """Make orbit creates the parabolic orbit for both galaxies. It then used
    this orbit to determine the initial position and velocity of the galaxy
    centers. Finally, it initialize the stars inside both galaxies and then plots
    the galaxies along with their orbiting stars"""
```

```
        energy = 0
        eccentricity = 1
        rperi = 3.0


        #Parameter provided by user (provides the distance of closest approach)
        tperi = float(self.peribox.value())

        #Returns the initial position and velocities of our galaxy centers
        self.crashOrbit = Orbit(energy,rperi,tperi,eccentricity,self.gal.galmass,
    self.comp.galmass,self.gal.galpos,self.comp.galpos,self.gal.galvel,self.comp.
    galvel)

        #Sets the initial position and velocity of the galaxy centers
        self.gal.setPosvel(self.crashOrbit.bod1pos,self.crashOrbit.bod1vel)
        self.comp.setPosvel(self.crashOrbit.bod2pos,self.crashOrbit.bod2vel)


        #If blue centered is checked, then we alter the X and Y range of the graph
     so that the blue(main) galaxy
        # is always in the center of our plot
        if self.bluecenterbox.isChecked():
            self.graphicsView.setXRange(self.gal.galpos[0,0]-20,self.gal.galpos
    [0,0]+20)
            self.graphicsView.setYRange(self.gal.galpos[1,0]-20,self.gal.galpos
    [1,0]+20)
        #The same principle holds for yellow centered
        if self.yellowcenterbox.isChecked():
            self.graphicsView.setXRange(self.comp.galpos[0,0]-20,self.comp.galpos
    [0,0]+20)
            self.graphicsView.setYRange(self.comp.galpos[1,0]-20,self.comp.galpos
    [1,0]+20)
        #Plots the position of the galactic center for both galaxies
        #We give the galaxies slightly different colors to that of the stars to
    distinguish more easily between the two
        Gal = pg.ScatterPlotItem(brush = self.brush6, size=6)
        Gal.setData([self.gal.galpos[0,0]],[self.gal.galpos[1,0]])

        Comp = pg.ScatterPlotItem(brush=self.brush1, size=6)
        Comp.setData([self.comp.galpos[0,0]],[self.comp.galpos[1,0]])

        #Plots the points onto our plot
        self.graphicsView.addItem(Gal)
        self.graphicsView.addItem(Comp)




        #Initializes the parameters for the stars in both galaxies
        self.gal.InitStars()
        self.comp.InitStars()



        #Plots the surrounding stars for both galaxies
        self.graphicsView.plot((self.gal.starpos[0,:]),(self.gal.starpos[1,:]),pen
     = None, symbol = "t1", symbolPen = self.pen6,symbolSize = 3)
        self.graphicsView.plot(self.comp.starpos[0,:],self.comp.starpos[1,:],pen =
     None, symbol = "t1", symbolPen = self.pen1, symbolSize = 3, fill=True)
```

```
        #dist is the galaxy separation
        dist = 3.5 * np.linalg.norm((self.gal.galpos-self.comp.galpos))
        #We then print the galaxy separation on our graph
        self.distlabel.setText("Galaxy Separation: {:.2f} kpc".format(dist))

        #vel is the relative velocity of the two galaxies
        vel = 250. * np.linalg.norm((self.gal.galvel-self.comp.galvel))
        #We then print the relative velocity on our graph
        self.vellabel.setText("Relative Velocity: {:.2f} km/s".format(vel))

        #Print inital time of t=0
        self.timelabel.setText("Time: 0 Myr")
```

Listing 18: Instance Method "MakeOrbit" which sets the initial positions and velocities of the galaxies as well as creating stars around the galaxies

Similarly to the MakeGalaxy method, MakeOrbit starts with a few fixed default parameters, along with the fact that for a parabolic orbit, the eccentricity is 1 while the energy is 0. From here, the user provides the distance of closest approach and using this value, we create an object in the class orbit which immediately initializes our orbit providing initial positions and velocities for our galaxies which we then set to our galaxies using the setPosvel method.

From there, MakeOrbit deals with the plotting of our galaxy positions. The first thing that is done is a check to see if the user has indicated that they wish the simulation to be centered on either of the galaxies. If this is the case, then the axis limits are redefined such that the position of the specified galaxy is always at the center of our simulation. Then, both galaxy positions are plotted on our graph, and we initialize the stars for each galaxy using the InitStars method. We then plot the positions of our stars for each galaxy in the corresponding color. In this way, one can better understand how the evolved galaxy after the collision is comprised of a mixture of stars from both initial galaxies. Then, the final thing we do in MakeOrbit is to calculate the galaxy separation as well as the relative velocity of the galaxies, and then print these out in our text labels.

Having now initialized our system, we obtain a graph which shows the two galaxies along with all their stars, as well as giving us the separation between the galaxies and their relative velocities. An image of our graph after our galaxies and stars have been initialized can be found in figure 7.

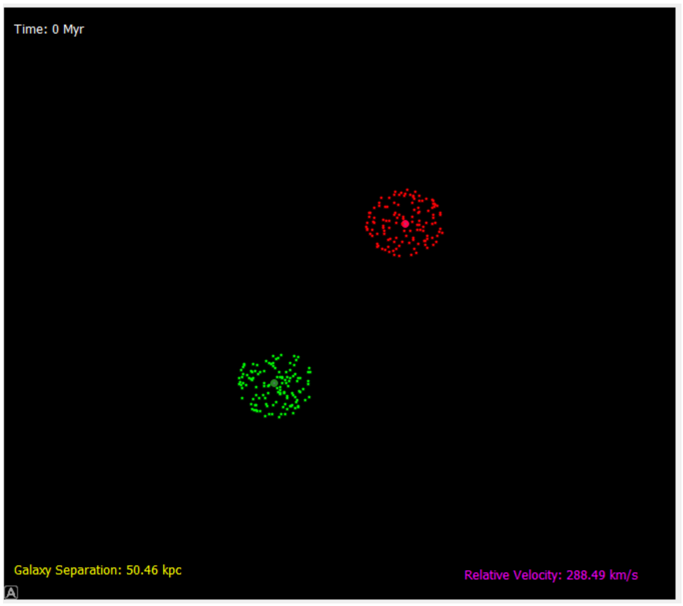


Figure 7: Graph after initialize button is pressed, showing the galaxies and their stars

### 3.4.2 Evolving the galaxies and stars over time

The next step, is to then evolve our system over time. When the Start button is pressed, our method updateplot will be carried out, which will evolve our system over time, allowing us to see how the galaxies interact. The updateplot method can be found in listing 19

```
def update_plot(self):
        """The update plot function runs the simulation. It calculates the acceleration of the galactic centers for both galaxies as well as the stars within the galaxies. These are then used to move the galaxy centers as well as the stars. The plot is then updated to show this evolution."""

        #We set the restartbutton and pausebutton to True here, while these are active, the plot is constantly updated
        #If the pause button is pressed, the update plot function stops running but we can still see our plots
        #If the reset button is pressed, then we clear all our graphs and reset the time back to 0
        self.restartbutton.clicked = True
        self.pausebutton.clicked = True

        #dtime taken from Chris Mihos' version, used to evolve the system over time
        dtime = 0.04
        if self.restartbutton.clicked:
            while self.pausebutton.clicked:
                #Clears the graphs so that the new positions can be plotted without overlapping the previous plot
                self.graphicsView.clear()
                self.distplot.clear()
                self.velplot.clear()

                #Calculating the acceleration using our acceleration function
                self.gal.galacc = self.comp.Acceleration(self.gal.galpos)
                self.comp.galacc = self.gal.Acceleration(self.comp.galpos)
                #dist is the relative distance between the two galaxies
                dist = 3.5 * np.linalg.norm((self.gal.galpos-self.comp.galpos))


                #If Dynamic Friction is present (given by user), then acceleration is calculated slightly differently
                if self.checkfriction.isChecked():
                    self.gal.galacc = self.gal.galacc + self.comp.DynFric(self.gal.InteriorMass(dist/3.5), self.gal.galpos,self.gal.galvel)
                    self.comp.galacc = self.comp.galacc + self.gal.DynFric(self.comp.InteriorMass(dist/3.5), self.comp.galpos,self.comp.galvel)

                comacc = ((self.gal.galmass*self.gal.galacc) + (self.comp.galmass*self.comp.galacc))/(self.gal.galmass + self.comp.galmass)
                self.gal.galacc = self.gal.galacc - comacc
                self.comp.galacc = self.comp.galacc - comacc

                self.gal.staracc = self.gal.Acceleration(self.gal.starpos) + self.comp.Acceleration(self.gal.starpos)
                self.comp.staracc = self.comp.Acceleration(self.comp.starpos) + self.gal.Acceleration(self.comp.starpos)

                #After the acceleration is calculated, we then evolve the system using the functions MoveGalaxy and MoveStars


                self.gal.MoveGalaxy(dtime)
                self.gal.MoveStars(dtime)

                self.comp.MoveGalaxy(dtime)
                self.comp.MoveStars(dtime)

                #dist and vel are the relative position and velocity between the two galaxies
                dist = 3.5 * np.linalg.norm((self.gal.galpos-self.comp.galpos))
                vel = 250. * np.linalg.norm((self.gal.galvel-self.comp.galvel))


                #Again, if blue centered is checked, then we want the graph to constantly have the blue (main) galaxy at the focus, so we constantly update the X and Y axis limits so that the galaxy appears at the center
                if self.bluecenterbox.isChecked():
                    self.graphicsView.setXRange(self.gal.galpos[0,0]-20,self.gal.galpos[0,0]+20)
                    self.graphicsView.setYRange(self.gal.galpos[1,0]-20,self.gal.galpos[1,0]+20)
                #Similarly for the yellow galaxy
                if self.yellowcenterbox.isChecked():
                    self.graphicsView.setXRange(self.comp.galpos[0,0]-20,self.comp.galpos[0,0]+20)
                    self.graphicsView.setYRange(self.comp.galpos[1,0]-20,self.comp.galpos[1,0]+20)

                #Plotting the positions of the galaxies exactly the same as before
                Gal = pg.ScatterPlotItem(brush=self.brush6, symbol='o', size=6)
                Gal.setData([self.gal.galpos[0,0]],[self.gal.galpos[1,0]])
                Comp = pg.ScatterPlotItem(brush=self.brush1,symbol='o', size=6)
                Comp.setData([self.comp.galpos[0,0]],[self.comp.galpos[1,0]])
                self.graphicsView.addItem(Gal)
                self.graphicsView.addItem(Comp)
```

```

                #Likewise, we also plot the positions of the stars exactly the same as before
                self.graphicsView.plot((self.gal.starpos[0,:]),(self.gal.starpos[1,:]),pen = None, symbol = "t1", symbolPen = self.pen6, symbolSize = 3)
                self.graphicsView.plot((self.comp.starpos[0,:]),(self.comp.starpos[1,:]),pen = None, symbol = "t1", symbolPen = self.pen1, symbolSize = 3)


                #Here we append our galaxy separation and relative velocity to the arrays defined at the start
                #These are then used to plot the galaxy separation and relative velocity as a function of time
                self.Dist.append(dist)
                self.Vel.append(vel)
                #Result is multiplied by 12 to be in Myr (taken from Chris Mihos' version)
                self.Time.append(self.time*12)

                #Updates the labels with the new values of each parameter
                self.distlabel.setText("Galaxy Separation: {:.2f} kpc".format(dist))
                self.vellabel.setText("Relative Velocity: {:.2f} km/s".format(vel))
                self.timelabel.setText("Time: {:.2f} Myr".format(self.time*12))

                #Plots the relative velocity and galaxy separation
                self.velplot.plot(self.Time,self.Vel,pen=self.pen2, width = 2, name = "Relative Velocity")
                self.distplot.plot(self.Time,self.Dist,pen=self.pen4, width = 2, name = "Galaxy Separation")


                #processEvents updates the plots
                QApplication.processEvents()

                #Advance the time
                self.time += dtime
        else:
            None
```

Listing 19: Instance Method "updateplot" which evolves the system in time

The first thing we do in updateplot is to set the status of our restart button and pause button to True. While both of these booleans are true, our application will continuously evolve the system over time. If either the pause button, or the reset button is pressed, then the updateplot method will cease to run. The specifics of what the restart button and the pause button do, will be discussed in the next section.

When the updateplot method is running, the first thing it does is it clears all our graphs. If this is not done, then as the system evolves, the new positions of our galaxies and stars will be plotted on an already exisiting plot of their old positions. By clearing the graphs each time, we can essentially have the galaxy and stars "move" instead. The next thing our updateplot function then does is to calculate the acceleration of both galaxies. This is done using our Acceleration method from our Galaxy class which we defined earlier. If the user enables friction, then the effect of dynamic friction will also be calculated using the method DynFric. The effect of dynamic friction will then be added to our acceleration to find a new acceleration for our galaxies. From here, we then find comacc, which represents the acceleration of the center of mass of the two galaxies. This is then subtracted from each galaxies acceleration to obtain their final acceleration.

Then, we calculate the acceleration of our stars by summing the acceleration due to both galaxies separately. After having calculated the accelerations of both our galaxies as well as our stars, we can then evolve the positons and velocities of these using the methods MoveGalaxy as well as MoveStars. To do this, we use a fixed default parameter dtime to advance the system through time. Now that our stars and galaxies have new positions and velocities, we calculate the galaxy separation and relative velocity of the galaxies again.

We then plot the new positions of the galaxies and the stars, again, checking to see if the user opted for the simulation to be centered around either galaxy and plotting accordingly. The galaxy and star positions are plotted in exactly the same way as in our initialization stage. However, here we also create two new plots, as we plot the galaxy separation over time, as well as the relative velocity of the two galaxies over time. Then, we simply update the plots and advance the time by dtime. This process is then repeated until the application is interrupted by the user.

### 3.4.3 Interrupting the simulation

Now we have a working simulation that plots the evolution of the galaxies and their stars over time, giving the user information on the galaxy separation as well as the relative velocity of the two galaxies. The simulation will run indefinitely unless the program is closed or interrupted by the user. Two buttons have been added for the user to interrupt the program. The first button is the reset button. When the reset button is pressed, as mentioned earlier, our boolean will change from True to False, and our program will no longer continue to run the method updateplot. Instead, another method is called upon, namely the reset method (see listing 20).

```
    def reset(self):
        """Reset clears all our visual plots and informationand resets the time to
    0, so that new parameters can be given and the simulation can be run again"""

        self.restartbutton.clicked = False
        self.pausebutton.clicked = False
        self.graphicsView.clear()
        self.graphicsView.setXRange(-20,20)
        self.graphicsView.setYRange(-20,20)
        self.velplot.clear()
        self.distplot.clear()
        self.Dist = []
        self.Vel = []
        self.Time = []
        self.time = 0.00
        self.distlabel.setText("Galaxy Seperation:")
        self.vellabel.setText("Relative Velocity:")
        self.timelabel.setText("Time:")
```

Listing 20: Instance Method "reset" used to reset the simulation

Pressing the reset button thus interrupts our updateplot method, and it clears all our graphs, resetting the time to 0. From here, the user can now change parameters and then initialize the system again, and new galaxies with different properties can now be simulated. This allows the user to easily study the effect of changing different parameters without having to restart the application every time.

The other button present is the pause button. The pause button is similar to the reset button in that pressing it interrupts the updateplot method. Pressing the pause button also calls upon the pause method (listing 21)

```
    def pause(self):
        """We create a seperate function for when pause is clicked that simply
    sets the pausebutton.clicked boolean to False This then means that our plot
    will no longer be updated until start is pressed once again"""
        self.pausebutton.clicked = False
        #When start is pressed again, we re-run our update_plot function in which
    we immediately set the pausebutton boolean to True again, so that our
    simulation will continue to run from where it left off.
```

Listing 21: Instance Method "pause" used to stop the simulation temporarily

The difference between the reset and the pause button is that pressing the pause button does not clear any of the previous information, so the graphs still remain intact allowing the user to examine the evolution of the system at a specific time more easily. Pressing start again will then simply resume the simulation and will continue to evolve the system.

### 3.4.4 Running the Application

Now that we have all the functionality desired in our program, the only thing left is to add some code to start up the GUI when the python file is run. This is done in listing 22.

```
# here we check if there already exists an QApplication. If not we create a new
    one.
if not QApplication.instance():
    app = QApplication(sys.argv)
else:
    app = QApplication.instance()

window = GUI()



# here the application is started


app.exec_() #for use an in interactive shell
```

Listing 22: Code to launch the GUI

# 4 Evaluating the educational value of the Application

In order to evaluate the quality of the application as an educational tool, the galcrash Python tool was tested by a tutorial group consisting of 5 students and 1 teaching assistant. The aim of the tutorial was to "report on the simulations of galaxy interactions and mergers, and why more and more powerful computers are required to conduct larger and more accurate simulations". In this section, the positive and negative feedback received will be discussed, and will be used to come up with further improvements to be made to the application in order to enrich its educational quality. The full evaluation report can be found in Appendix B.

## 4.1 Discussing the feedback from tutorial session

In this session, students were able to observe different types of mergers by playing around with the parameters. This allowed students to discover the characteristic results of major and minor mergers, noting that at the end of a galaxy interaction, you are left with either a single merged galaxy, two galaxies or two galaxies that are kind-of joined together. Students also discovered that dynamical friction was vital to the galaxies merging, but were unaware of what exactly dynamical friction was. Furthermore, they learned that the distribution of stars in galaxies are impacted from a distance, even when the galaxies are not "touching" each other. Students also observed that the simulation has limitations, and has very different parameters than real-life galaxies, such as the simulated galaxies only having 250 stars, when in reality, the number of stars present in a galaxy is on the order of billions. This is important to realize as the simulation makes many assumptions and simplifications that are not necessarily correct, and therefore the simulation should be used to help explain observations and gain a basic understanding of galaxy interactions.

The galcrash application proved to be very fruitful in helping students understand the basics of galaxy interactions. A large part of this was due to the fact that the code was written in Python. As the students were somewhat familiar with Python, it allowed them to look into the code to search for answers to questions that arose. Properties like how the initial velocities are calculated can easily be found in the code, allowing the students to better understand the calculations and mathematics going on within the code to produce the simulations. Looking at the code of such an application also provided students with an example of how to run code with a user interface in Python, which could be helpful for students looking to make similar simulations in the future.

Some criticisms of the application were made, one of which was that when the masses of the two galaxies were chosen to be very different, things were "not obeying the laws of physics". This is largely due to the problem of time-steps used in simulations. The properties of the stars and the galaxies are constantly changing but in a simulation, we only change these properties at certain time-steps, keeping them constant in between these time-steps. This leads to inaccuracies in simulations especially at larger mass differences, in which the changes in the properties of the stars and galaxies are much more volatile. Using a smaller time-step would help in improving the accuracy of the simulation but would require many more calculations, slowing down the speed of the application. While this could be done, this is not necessarily ideal as the purpose of the application is to allow students to simulate galaxy interactions, and should be easily accessible, not requiring supercomputers to be able to run such an application. The application is already quite taxing, with students having had trouble running the simulations when the number of stars was very high due to the speed or the overheating of computers.

Having said this, this "loss of physics" could also be used as a learning opportunity to teach students about using time-steps within simulations and thus give students more insight into the drawbacks of creating simulations and where the limitations of such simulations lie.

## 4.2 Changes made to the application

Along with feedback mentioned in the evaluation report in Appendix B, feedback was also given on certain technical issues, as well as requesting extra features that would enhance the experience of the students. Based on that feedback, several changes and additions were made to the application to improve the overall quality of the application.

Previously, an initialize button was used to plot the galaxies based on the parameters provided by the user. This was done is such a way that every time a user would change a parameter, they would have to press Initialize again to see how changing that parameter reflects the initial state of the simulation. This could make it quite difficult to see how certain parameters like inclination and slew angles affected the orientation of the galaxies. Thus, the initialize button was removed, and the application was reworked in such a way that the application automatically plots both galaxies along with their stars based on the default parameters when launched. Then, when the user provides any change in a parameter, this is also immediately reflected in the image of the two galaxies, so the user has a more dynamic view of the impact of changing parameters on the system. Furthermore, other small additions were also added to improve the quality of the application. The application contains an option to have the graph centered around the main galaxy. However, there was no option for the graph to be centered around the companion galaxy. This is something students also wanted to do, and so this feature was added.

Another change made to the application was in the colour scheme. In the original design of the application, the galaxies simulated were coloured red and green. However, upon receiving feedback that such a scheme could prove quite difficult to follow for visually impaired or colour blind people, the colour scheme was changed to a yellow-blue scheme instead. In this way, the contrast was increased to improve overall visibility and make it more widely accessible to a variety of students. There were also some minor technical issues with the GUI that were brought up that are being reviewed.

## 4.3 Summary

Overall, the application proved to be a very useful tool for students to gain a basic understanding of galaxy interactions and how changing certain parameters impacted the outcome of the interaction. By having the application written in Python, it allowed students to be able to look into the code to find more information about the underlying mathematics in such a manner that it was understandable and could be followed.

Some concepts like major and minor mergers were described, as well as tidal tails. However, students were unaware that they were describing these scenarios. Therefore, it may be useful when using this application to introduce such concepts and vocabulary to the students, so that they may better understand what is happening. Another important aspect that is not well described is dynamical friction. As students discovered, dynamical friction is the driving force behind the merging of galaxies. However, students were unaware of what dynamical friction actually means. By having the concept of dynamical friction explained, students would be more aware of why the galaxies are merging. Thus, the application would work best when accompanied with a tutorial, in which the concepts can be introduced and the students can explore these concepts through help of the simulation.

# 5 Conclusion

The aim of this project was to produce an updated version of the GalCrash application first created by Chris Mihos to run using Java. The updated version was written in Python, with its purpose being to educate students about galaxy interactions. The application runs simulations of galaxy interactions between two spiral galaxies, taking parameter inputs from the user. The new application written in Python has the same functionality as the old java application, but with more explanations within the code, allowing the user to better understand how the underlying code works, and what processes are taking place. The fact that the application is written in Python also allowed students to look into the code to further explore the simulation while still understanding what is being done.

According to the evaluation report, the application was overall very successful in introducing students to the concepts of galaxy mergers. It allowed the students to explore parameters and their effects on galaxy interactions. The application gave a very basic introduction to students on the concepts of dynamical friction and allowed students to explore major and minor mergers. However, the application lacked explanation as to what these concepts were. Students were able to identify the impact of dynamical friction but were unaware of what it actually was. Students discovered the properties of major and minor merger remnants without knowing what major and minor mergers were. Thus while the application allowed students to explore galaxy interactions, it did not offer a lot of information with regards to extra vocabulary and what the processes taking place actually are. By accompanying the application with a tutorial, in which the concepts are explained, the application would be much more successful in helping students understand these concepts and then exploring the properties of these concepts.

## 5.1 Further Improvements

The new application does not add many new features that were missing in the java application. This is something that could be improved upon further in future. Possible new features may include things like presets, wherein one can easily choose two real galaxies to simulate their merger. By selecting a preset, the user would not have to input any information about the mass ratio or the number of stars, as this would already be known, but it would allow the user to simulate the collision of two known galaxies easily. An example of this could be the Milky Way-Andromeda galaxy collision. By having such presets, it allows the user to very easily select their required collision parameters without having to do prior research on their properties.

Possible other features could see the inclusion of more diversity in the simulation. Being able to choose different types of galaxies other than spirals, or different types of orbits for the galaxies would allow the simulation much more flexibility in terms of its capabilities.

Another possible additional feature that could be added is to allow the user to specify a time for which they want to see the evolution. As an example, if the user wishes to see how different parameters affect the galaxy interaction at a time of 100Myr, then providing the user with such a tool that they can specify the time for which they are interested in seeing the results. This could make it easier for the user to compare the results of galaxy interactions for varying parameters.

# A Derivation of Rotation Matrix

The GalCrash application allows for the user to input the inclination and slew of the two galaxies. The inclination and slew of the galaxies affect the way in which the positions and velocities of the stars within a galaxy are distributed. This is done via a rotation matrix. To better illustrate our inclination and slew angles, we use figure 8.

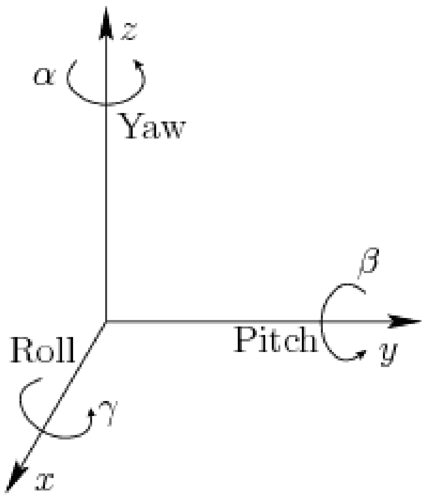


Figure 8: Yaw, Pitch and Roll Rotation Axes (LaValle, 2006)

The yaw, pitch and roll axes all have well defined rotation matrices. By comparing our inclination and slew axes to these three, we can determine the rotation matrices of our system. To do this, we look at figure 9 again, which can also be found below.

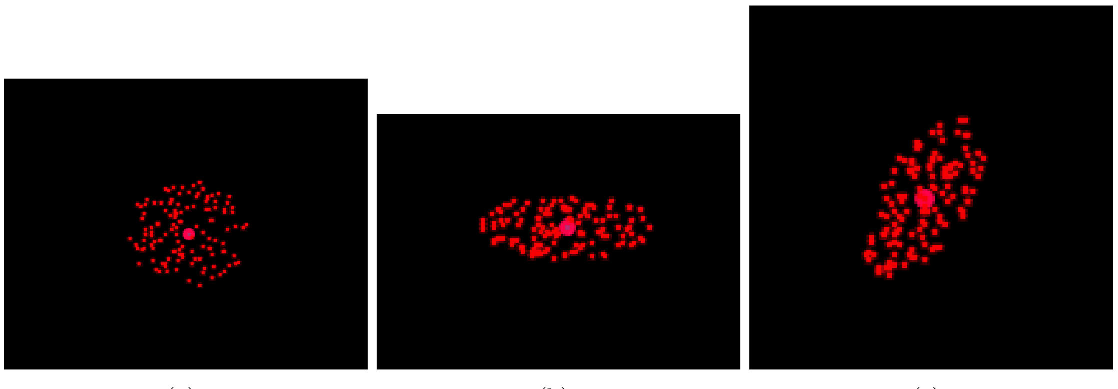


Figure 9: (a) Inclination and Slew of red galaxy are both 0°, (b) Inclination of red galaxy changed to 90°, (c) Inclination and Slew of red galaxy are both 90°

Comparing these figures, we see that an increase in our inclination angle is a rotation around the x-axis (roll) while an increase in our slew angle equates to a rotation around the z-axis (yaw). However, yaw and roll occur when there is a counterclockwise rotation about the axes. In our case, we are dealing with clockwise rotations, and so the sin terms in both our roll and yaw matrix should be multiplied by -1. Our inclination angle is given by $\theta$, and so, expressing the roll axis rotation matrix in terms of $\theta$, (and multiplying sin terms by -1), we obtain the following matrix:

$$R_i = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{16}$$

The slew angle is given in terms of $\phi$. Writing the yaw rotation matrix in terms of $\phi$ (multiplying sin terms by -1), we have:

$$R_s = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{17}$$

$$R = R_s \times R_i = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{18}$$

Doing the multiplication, we obtain our rotation matrix:

$$R = \begin{bmatrix} \cos\phi & \cos\theta\sin\phi & \sin\theta\sin\phi \\ -\sin\phi & \cos\theta\cos\phi & \sin\theta\cos\phi \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{19}$$

To obtain the rotation in x,y, and z, we simply multiply by our position vector:

$$\begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} = \begin{bmatrix} \cos\phi & \cos\theta\sin\phi & \sin\theta\sin\phi \\ -\sin\phi & \cos\theta\cos\phi & \sin\theta\cos\phi \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{20}$$

Working this out then yields equations 10 - 12 used in the code (also seen below).

$$x_r = x\cos\phi + y\sin\phi\cos\theta + z\sin\phi\sin\theta \tag{21}$$

$$y_r = -x\sin\phi + y\cos\phi\cos\theta + z\cos\phi\sin\theta \tag{22}$$

$$z_r = -y\sin\theta + z\cos\theta \tag{23}$$

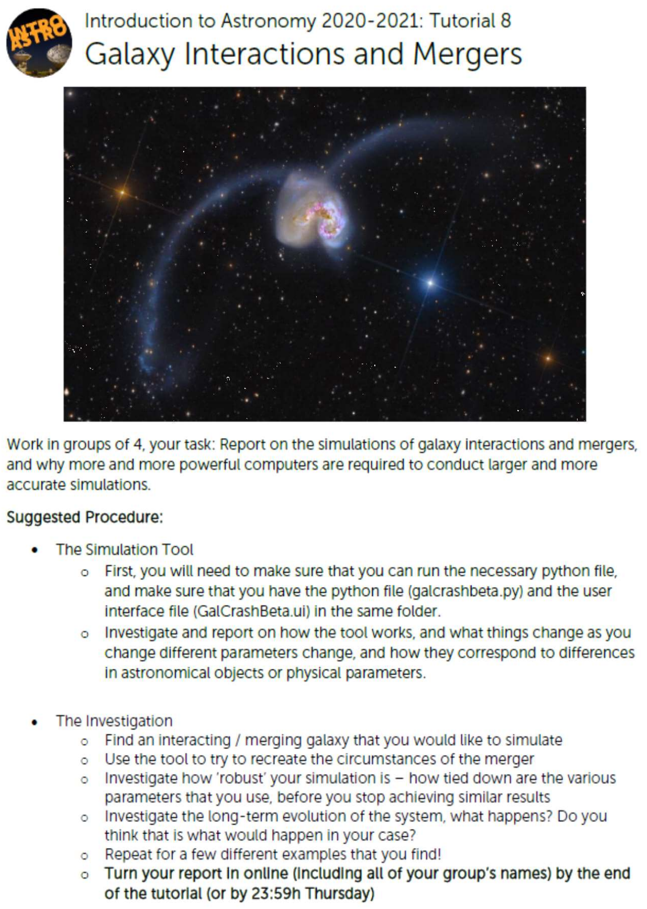
## B Evaluation Report

## Evaluation report – GalCrash Python

**Situation:**

The galcrash python tool was evaluated by a classroom observation and follow-up focus group. The classroom situation was typical for a first-year introductory astronomy tutorial group at the University of Groningen, though due to the COVID-19 restrictions in place at the time of the study, the session was held online in Zoom. The 5 students and 1 Teaching Assistant (TA) involved had all become familiar with using Zoom (or other online learning environments) during the previous weeks of classes. The session was conducted in English, though no participants were native English speakers, however, their regular language of instruction is English, and all exhibit an English language level of at least C1 up to fluent. The students were all at the end of their first year Bachelor studies in Astronomy at the University of Groningen. None of the students, nor the TA, had participated in (or taught) previous versions of the tutorial using pre-existing versions of the software. The session was independently observed by Dr Jake Noel-Storr (reporting here), with the only contact during the session being to respond to technical questions from the Teaching Assistant outside of the learning platform (as is the case in the typical IA tutorial setup). The tutorial ran for 2 hours, and the focus group for 15 minutes at the end.

The tutorial used is shown below:



Introduction to Astronomy 2020-2021: Tutorial 8

Galaxy Interactions and Mergers

Work in groups of 4, your task: Report on the simulations of galaxy interactions and mergers, and why more and more powerful computers are required to conduct larger and more accurate simulations.

**Suggested Procedure:**

- The Simulation Tool
  - First, you will need to make sure that you can run the necessary python file, and make sure that you have the python file (galcrashbeta.py) and the user interface file (GalCrashBeta.ui) in the same folder.
  - Investigate and report on how the tool works, and what things change as you change different parameters change, and how they correspond to differences in astronomical objects or physical parameters.

- The Investigation
  - Find an interacting / merging galaxy that you would like to simulate
  - Use the tool to try to recreate the circumstances of the merger
  - Investigate how 'robust' your simulation is – how tied down are the various parameters that you use, before you stop achieving similar results
  - Investigate the long-term evolution of the system, what happens? Do you think that is what would happen in your case?
  - Repeat for a few different examples that you find!
  - **Turn your report in online (including all of your group's names) by the end of the tutorial (or by 23:59h Thursday)**

**Technical outcomes:**

Technical issues and outcomes unrelated to learning were reported to the developer. None of the extant technical issues had an observable effect on the overall learning experience – and represented only recommendations for improvements to the user interface, or occurrences of unexpected errors.

**Observed outcomes:**

The following learning outcomes were observed to take place during the classroom observation. Paraphrased quotes from the session are included that highlight the learning that took place, including from discussions between the students of things that they screenshotted and shared as "interesting".

| **Learning Outcomes; that…** | **Exemplary Observations (paraphrased quotes)** |
|---|---|
| Simmulations have limitations, but can help us to explain observations | "When the masses are very different, things are definitely not obeying the laws of physics"<br><br>"Wow – we are only using 250 stars, but galaxies have billions" |

2

| | |
|---|---|
| Simmulations with more detail, require greater processing power and technology to complete them | "Its so much slower when you go up to more stars"<br><br>"My laptop is melting now" |
| Astronomical images generally represent a single snapshot in time | "Time is really slow"<br><br>"We have to use the stop button to match an image, so the image is not really the end point for the galaxy interaction" |
| Astronomical images generally represent a single two-dimensional representation of a three-dimensional situation | "Well we can move the simulation around to make it look like the observation, but we cant move the observation around so who knows in 3D if its right" |
| Looking at code can explain how a simulation works and the assumptions that lie within it | "What are the initial velocities… oh wait we can look at the code" |
| Galaxy mergers and interactions can have different outcomes | "We don't know what dynamical friction does, but things won't merge without it"<br><br>"At the end, sometimes they are oscillating (and changing a bit each time), or sometimes a blob, or sometimes are just carrying on forever" |

**Declared learning outcomes:**

During the focus group, students were asked to describe what they thought that they had learned, and these items are listed below:

- That galaxies don't directly crash into each other centre to centre
- How to run code with a user interface in Python
- Mass difference impacts the accuracy / reality *(Note: this was made as a criticism of the code, not the realization that is a problem with time-steps in simmulations)*
- At the end of an interaction or merger you can end up with just one thing, two things, or two things that are kind-of joined together. Distances and velocities can become zero, be oscillating, or keep moving away forever.
- After mergers they can be joined together with spiral arms *(Note: Describing Tidal Tails – indicating that extra vocabulary should also be introduced first)*
- The distribution of stars in galaxies can be impacted even when the galaxies don't "touch" each other *(i.e. Gravity acts at a distance).*
- That images are just 2-D, but each galaxy can be moved around in 3-D around 3 axes, and then the view can also be moved in 3-D, so the physical situation is much more complex than images reveal.

## References

Aceves, H., & Colosimo, M. (2006). Dynamical Friction in Stellar Systems: an introduction. *arXiv e-prints*, arXiv physics/0603066, physics/0603066.

Delgado-Serrano, R., Hammer, F., Yang, Y. B., Puech, M., Flores, H., & Rodrigues, M. (2010). How was the Hubble sequence 6 Gyr ago?, *509*arXiv 0906.2805, A78. https://doi.org/10.1051/0004-6361/200912704

Dynamical friction. (n.d.). Swinburne University of Technology. https://astronomy.swin.edu.au/cms/cpg15x/albums/userpics/dynamicalfriction1.gif

Fraknoi, A., Morrison, D., & Wolff, S. C. (2016). *Galaxy mergers and active galactic nuclei* [Accessed = 12-06-2020]. https://openstax.org/books/astronomy/pages/28-2-galaxy-mergers-and-active-galactic-nuclei

Gallagher, J., & Sparke, L. S. (2007). *Galaxies in the universe: An introduction* (2nd ed.). Cambridge University Press.

Lambas, D. G., Alonso, S., Mesa, V., & O'Mill, A. L. (2012). Galaxy interactions. I. Major and minor mergers., *539*arXiv 1111.2291, A45. https://doi.org/10.1051/0004-6361/201117900

LaValle, S. M. (2006). *Planning algorithms.*

Mihos, C., Caley, D., & Bothun, G. (1999). Galcrash: N-body simulations on the student desktop.