Tidal tails and bridges in galactic encounters

Candidate number: 6899V

ABSTRACT: We develop a restricted n-body simulation to study tidal tail and bridge formation in galactic encounters, providing a quantitative analysis of their development and shape, as well as of the effects of varying the inclination of the encounter and the strength of the perturbation. We further show that a full n-body simulation of rotationally supported *bulge : disk : dark matter halo* galaxies leads to orbital decay and efficient angular momentum transfer away from the luminous components. The resulting merger is well-described as an elliptical galaxy approximated by a *de Vaucouleur* profile. Moreover, we make use of a simulated annealing algorithm to construct a model for the Antennae galaxies without human intervention and show that it matches its morphology and Doppler shift observations from the Hydrogen 21-cm line.

[Word count: 3255]

Contents

1 Introduction

The discovery of multiple *anomalous* pairs of galaxies in the 1950s, with regions protruding into space (tails) and thin structures joining the pairs (bridges), lead to the suggestion that tidal encounters could have played a role in their formation. After all, the Antennae, the Mice, and many other pairs of such objects (figure 1) were often described so as to be "in obvious interaction" [1]. While these processes are nowadays understood to drive star formation [2, 3] and play a role in galactic evolution [4, 5], they were originally met with skepticism, as it was thought that encounters were tremendously unlikely and that tides could not produce such thin and elongated structures. In 1972, the pivotal work of Toomre & Toomre [6] established, through restricted n-body numerical simulation, the feasibility of tidal bridge and tail formation in close galactic encounters.

16 years later, Barnes advanced the field by considering the first self-consistent model including a dark matter halo and displaying dynamical friction [7]. Further models included star formation [8] and merged tree based codes [9] with Soft Particle Hydrodynamics (SPH) [10, 11], rivalling the also common Adaptive Mesh Refinement (AMR) method.

Following [12], the current approaches can be divided into two. First, efforts at exploring the large parameter space to provide statistical interpretations, where the publicly available GalMer dataset is the prime example [13]. Second, attempts at matching particular galaxies and observations [14, 15], such as star formation profiles [16] and cluster evolution [17], with high complexity and resolution.

In this work, however, we first deliberately develop a restricted n-body simulation. Hence we aim to show that bridges and tails are kinematic phenomena, that do not, for instance, necessitate self-gravity to explain the thinness of their features. This is presented in section 2. More advanced topics on dynamical friction in dark matter halos and reproducing astronomical observations are presented later in sections 3 and 4 respectively.

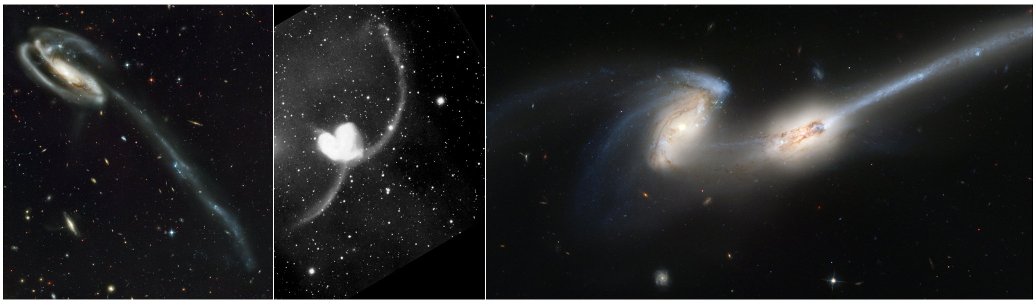


Figure 1. A selection of interacting galaxies as captured by the Hubble telescope. From left to right, The Tadpole (Arp 188), The Antennae (NGC 4038/4039) and The Mice (NGC 4676). We refer to *tails* as the elongated regions protruding from galaxies into space (all cases) and to *bridges* as the thin structures joining a pair of galaxies (right). Tails are commonly referred to as *counterarms* when they are clearly bound to their progenitor; we do not make such blurred distinction here.

2 Parabolic encounters

2.1 Analysis & implementation

Qualitative understanding of the tidal structures can be obtained by considering a simplified model in which two heavy point masses, one of which is surrounded by massless test particles constituting a galactic disk, interact gravitationally in a parabolic encounter of pericenter distance $r_{\text{min}} = 1$.[1] The inclusion of test particles surrounding the main mass only is deliberate, as for massless particles a second ring can be directly superposed from a complementary simulation where the main and companion masses have been interchanged. In fact, neglecting self-gravity, an *a priori* radial density distribution for the ring is unnecessary, as its effect could be reproduced by reweighting the particles based on their initial positioning once the simulation is complete. We thus place the test particles on discrete rings that can be trivially followed independently. The choice of parabolic trajectories is driven by two factors: (i) we expect encounters to be rare and not initially bound ($e \geq 1$). (ii) when no dynamical friction is included, lasting features are more likely to occur for soft, low eccentricity interactions.

[1] We choose units such that $G = 1$. One can choose to rescale all results to a typically-sized encounter, based on the Antennae, by letting 10.4 kpc, $5 \times 10^{11} \mathrm{M}_{\odot}$ and 21.3 Gyr correspond to 1 unit of length, mass and time respectively.

The code is presented in appendix A and documented thoroughly, but it is worth highlighting that we make use of an Object Oriented approach and configuration files for module reusing, save the progress of the simulation dynamically to avoid data loss, and vectorize the operations for clarity and performance (section 3.1). This leads to trivial generalization to more than two galaxies, different distributions and galactic objects, or SPH.

Numerically, we employ a 4^{th} order symplectic Verlet integrator and include "Plummer" gravitational softening [18] with characteristic length scale $\epsilon = 0.1$ to prevent numerical instabilities and realistically account for the extended bulge. Employing massless rings further reduces statistical fluctuations (relaxation effects).[2] Since each test mass does not represent a single star, the use of adaptative timesteps to model close interactions is unnecessary.

We perform four tests to validate the correctness and probe the numerical behaviour of our approach:

i We verify that galaxies evolve unperturbed in isolation, with absolute variations in the test particles' orbital radii of $< 10^{-3}$.

ii We confirm that pairs of masses follow the expected analytical orbits for no softening to within $< 10^{-3}$ for various eccentricities and mass ratios.

iii We study the deviation of test particles from their correct trajectories in a simple encounter and select a conservative timestep $dt = 10^{-3}$ for which the discrepancy is 4×10^{-3} length units on average within the timescale of interest (figure 2, left).

iv We ensure the energy of the system is conserved, to within 1% in the same encounter (figure 2, right).

2.2 Prograde and retrograde encounters

We first present a prograde encounter (figure 3), where the spin of the disk and the orbital angular momentum are aligned. A violent interaction is observed, leading to both a tail and a bridge. For this equal mass encounter, only particles that are initially placed at a radius of at least $0.4r_{\text{min}}$ contribute to the tidal structures, with more loosely bound rings resulting in a larger tail. Equivalently, close encounters are necessary for significant tails and bridges to form.

This prograde encounter suggests that tidal tails are the result of a broad resonance between orbiting particles and the companion mass. This is more easily observed in figure 4 where the particles are coloured according to their final fate. As the companion mass approaches the galaxy, the circularly symmetric disk elongates. At this point, the test masses closer to the companion will form a bridge; those on the opposite side will result in a tidal tail that can become several times larger than the original disk. It should be noted

[2] The mass of a galaxy is certainly not concentrated in its central bulge, but massless rings allow for an efficient $O(n)$ implementation in the number of test particles. Additionally, we stress once more that it will allow us to show that tidal tails are a merely kinematic phenomenon.

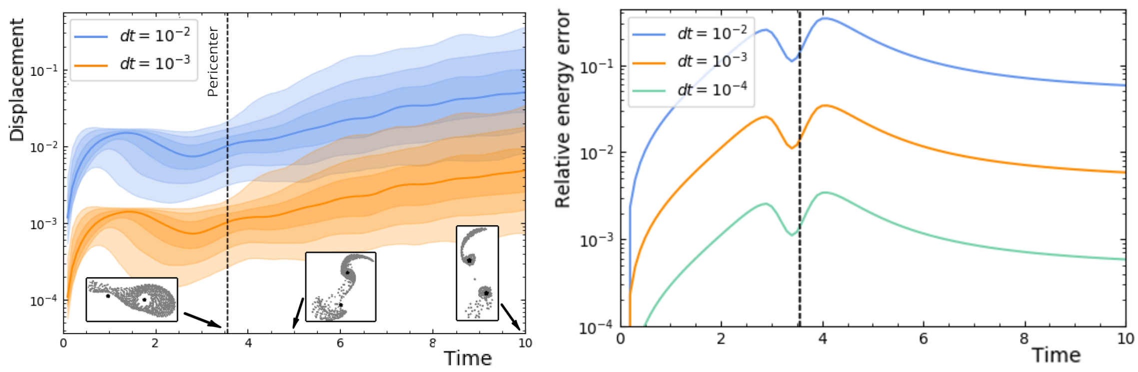


Figure 2. Errors in the displacement of the test masses (left) and in the energy of the system (right) for a prograde equal mass encounter (figure 3). The smoothed shaded regions contain 38%, 68% and 88% of the particles. For the chosen timestep $dt = 10^{-3}$ test particles deviate from their correct trajectory (taken as $dt = 10^{-4}$) by 4×10^{-3} length units on average within the timescale of interest, exceeding our plotting resolution. The energy is conserved to within $\sim 1\%$.

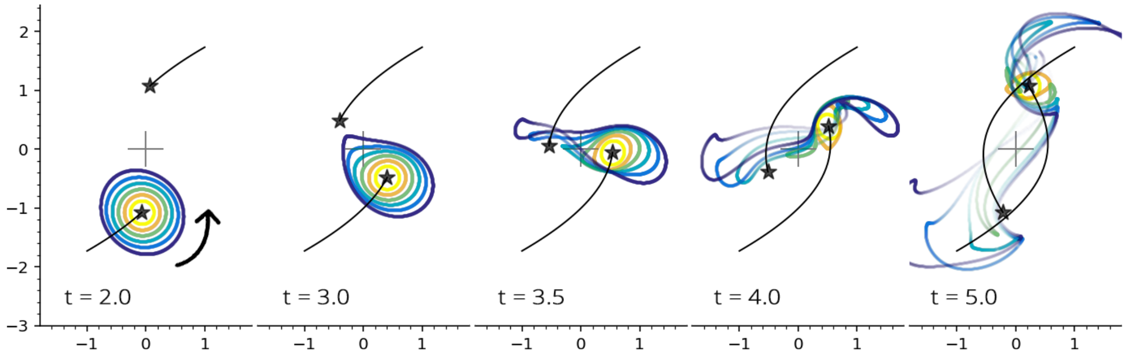


Figure 3. A prograde flat parabolic encounter with a companion of equal mass. Coloured rings originally placed at a radius .2, .3, .4, .5, .6, .7 and .8 from the main mass are shown at different times, with opacity reflecting the local density of particles. The central cross indicates the position of the center of mass and the stars that of the main and companion central masses. Only rings placed at a radius of .4 (green) or higher lead to bridges and tails, with further rings resulting in larger features.

that, whereas the tail is permanent in this encounter, the bridge is transient, with most particles eventually falling back to the main mass or being trapped by the companion, and few escaping from both masses. Of those particles that follow the perturbing mass, 94% will remain bound to it (figure 5), albeit in highly elliptical orbits, in sharp contrast to the largely unbound particles in the tail, which can span more than a 100 kpc in real galaxies.

To further support the resonance proposition, we show a similar interaction but where the encounter is retrograde (figure 6). The rings are now mostly undisturbed, even at large radii where for a real encounter the galactic rings themselves would have overlapped. Although it is tempting to argue these features on the basis of the Lagrangian points of the

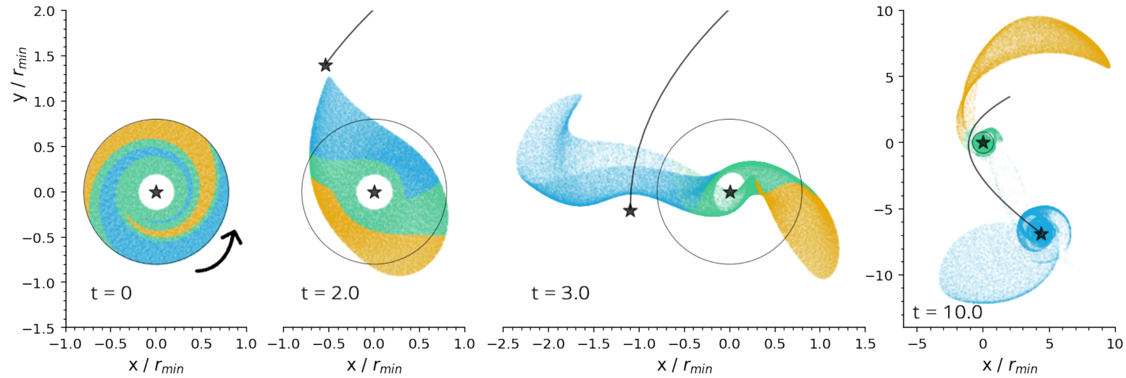


Figure 4. A prograde flat parabolic encounter with a companion of double the mass. A uniform disk of size $0.6r_{\rm min}$ surrounds the main mass and is coloured according to the eventual fate of each particle: tail (orange), orbiting the main mass (green) or stolen by the companion (blue). Whereas the bridge is transient and has negligible density at large times, the tail formed is permanent and progressively grows to become several times larger than the original disk.

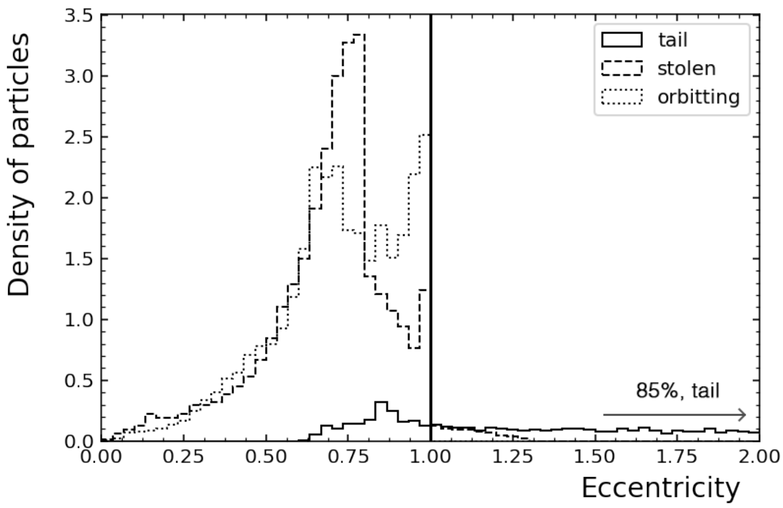


Figure 5. Eccentricity at large times for the encounter shown in figure 4 (prograde, parabolic, $m_{\rm companion} = 2 \times m_{\rm main}$). All of the particles that remain close to the main mass (green in figure 4) and 98% of those that follow the companion (blue) are bound, although in highly eccentric orbits. For the tail (orange), the opposite holds, with 94% of the particles having eccentricity $e > 1$ (85% of those particles have $e > 2$ and are not shown).

system, we must note that tails also form in inclined encounters (see later in section 2.4), and as such the resonance idea must be interpreted broadly.

2.3 A quantitative analysis

To perform a more quantitative analysis, it is necessary to design an accurate, automated measurement scheme. We develop a sequential segmentation algorithm for this task, illus-

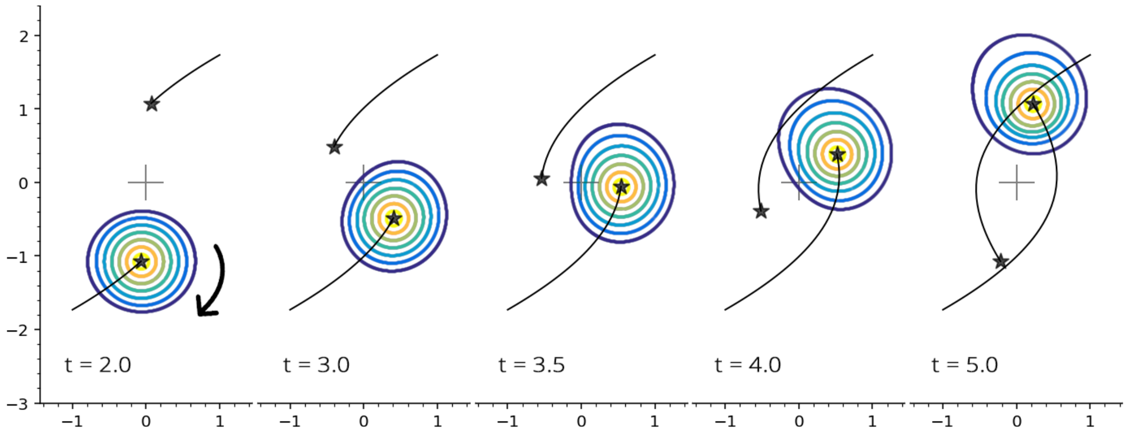


Figure 6. A retrograde flat parabolic encounter with a companion of equal mass. Coloured rings originally placed at a radius .2, .3, .4, .5, .6, .7 and .8 from the main mass are shown at different times. The rings remain mostly undisturbed, with no bridge or tail formation.

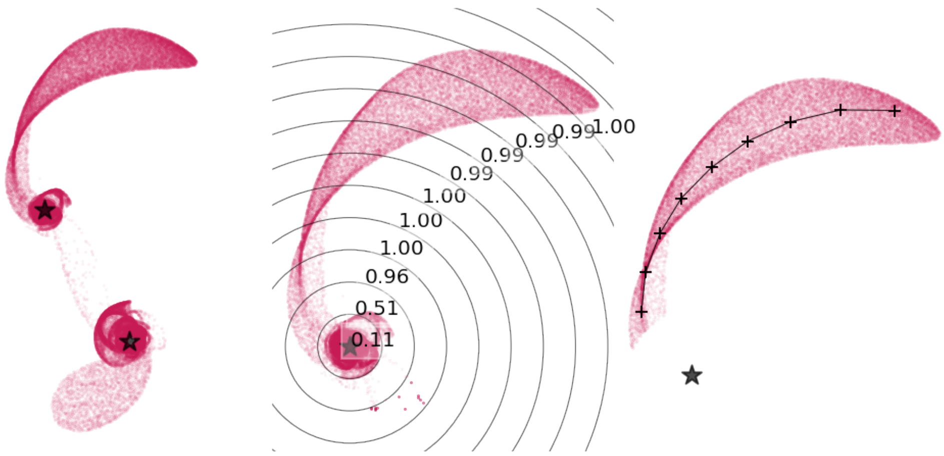


Figure 7. Depiction of the algorithm described in section 2.3. A limited number of concentric spheres (shown as rings) are used here for illustration purposes only, anotated with their f value. The stray particles from the bridge are emphasized in the middle picture and completely removed by the algorithm.

trated in figure 7 and described here:

1. We divide the space with planes normal to the segment joining the massive centers, at distances 30% and 70% along its length. The middle region is defined as the bridge to within a width of 3 units and removed.

2. The remaining particles are classified as belonging to the main or companion mass based on their relative distance to each. Those belonging to the main mass are divided

into 100 concentric spheres centered on the main mass, and the quantity $f = \frac{\sum_i \vec{r_i}}{\sum_i |r_i|}$, where r_i denotes the radial vector of the i^{th} particle is computed for each ring.

3. The start of the tail is associated with a sharp increase in f. We choose a cutoff at $f_0 = 0.80$, select the first ring with a value of f larger than f_0 and remove all closer rings. We also calculate the vector $\vec{q} = \sum_i \vec{r_i}$ for the now innermost ring and remove all particles in a direction opposite to $\vec{q}$ placed in the now innermost 5 rings, as these are commonly stray particles from the bridge. The barycenters of the spherical shells can now be joined to determine the shape and length of the tail.

The algorithm is satisfactorily checked to match human expectations for all the cases presented in this report.[3] We employ it to study the time evolution of prograde encounters, based on the qualitative example in the preceding section, but where the companion mass varies from that of the main mass (figure 8). It confirms that tidal bridges are transient features whose life is however extended when the perturbing mass is small. Tidal tails, on the contrary, are only significant when the companion is at least of similar mass, but can then span vast lengths. It follows that when two tails are observed, as is the case for The Antennae (figure 1, middle), both galaxies should have similar masses.

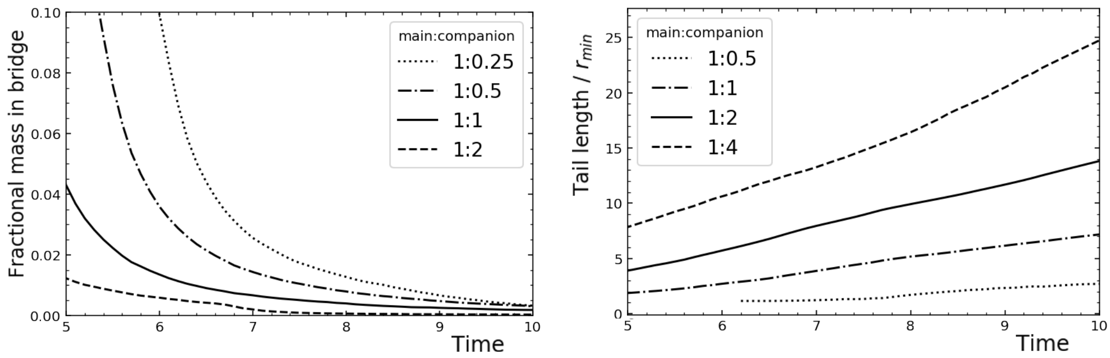


Figure 8. Evolution of the mass in the bridge (left) and the tail length (right) for a set of flat, prograde, elliptic encounters of different *main mass:companion mass* ratios. A uniform disk of size $0.6r_{\text{min}}$ initially surrounded the main mass.

At large times, when the bridge has become negligible, we may assign each particle to one of three groups: still orbiting the main galaxy, part of the tail, or stolen by the companion. Figure 9 shows that the fractional mass belonging to each group is a strong function of the perturbing mass to main mass ratio. When the companion is multiple times more massive, the encounter is highly disruptive and a large fraction of the disk ends up either stolen ($> 30\%$) or forming the tail ($> 40\%$).

Making use of the algorithm introduced, we can additionally analyse the shape of the tail. To the surprise of the author, flat encounters display a universal tail shape, independent of the mass of the companion, the time since pericenter and mostly the closeness of

[3] We have, in fact, already made use of it to colour the particles based on their final classification in figure 4.

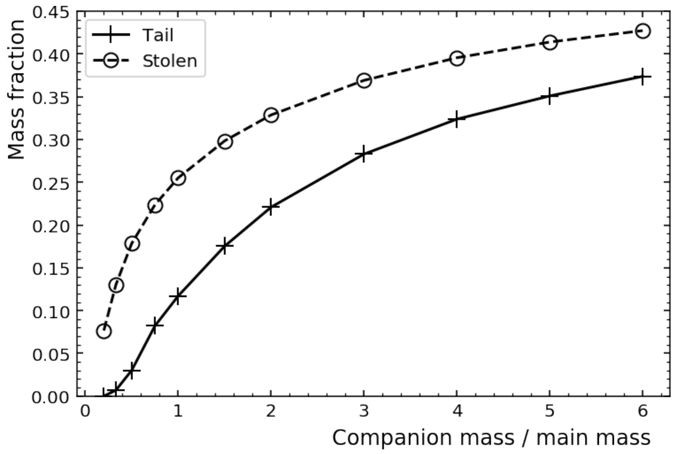


Figure 9. Fractional mass of the disk that, at large times, belongs to the tidal tail (solid) or has been stolen by the companion (dashed) as a function of the companion mass. The remaining fractional mass mostly continues closely orbiting the main mass. The main mass is initially surrounded by a uniform disk of radius $0.7r_{\mathrm{min}}$.

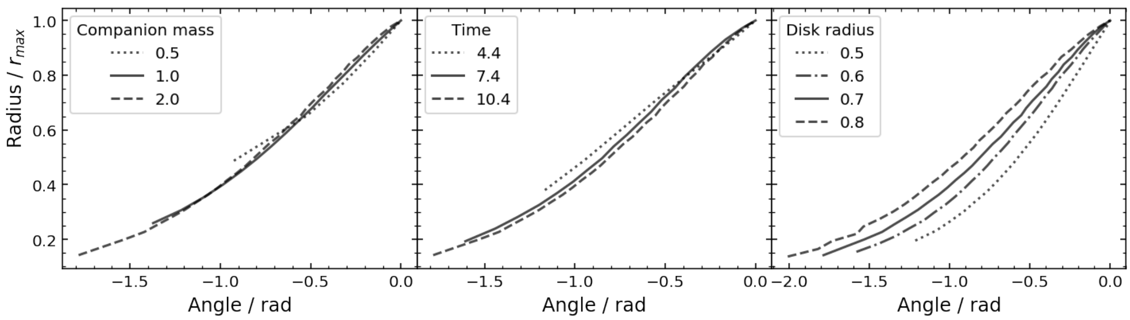


Figure 10. Shape of the tail for flat prograde encounters with pericenter distance $r_{\mathrm{min}} = 1$ differing on the mass of the companion (left), the time since pericenter (center) and the radius of the ring (right). The radial distance is normalized by dividing by the radial extension of the tail. The fact that the curves start at different radii is simply due to the difficulty of meaningfully defining the tail close to the main mass.

the encounter (figure 10). Although this will not hold true for more general interacting geometries, it makes it easier to guess appropriate viewing angles or initial conditions, since tails such as those of the Mice galaxies are now known not to be flat but rather curved away from the viewing plane.

2.4 Extending the geometry

It is clear that for a flat galactic encounter as presented so far both tails would be curved in the same direction. Observations of for instance the Antennae, where this is not the case, urge us to generalize the geometry of the event. This can be accomplished through the

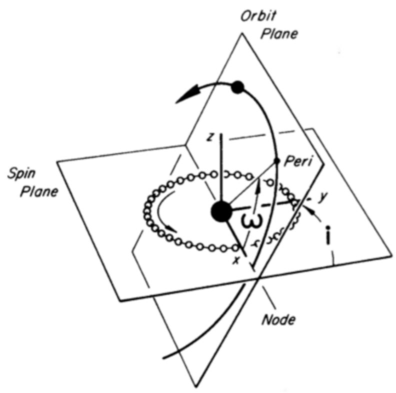


Figure 11. The interacting geometry, reproduced from [6]. i denotes the angle between the galactic and orbital planes; ω denotes the angle between the galactic plane and the pericenter, as measured from the main mass.

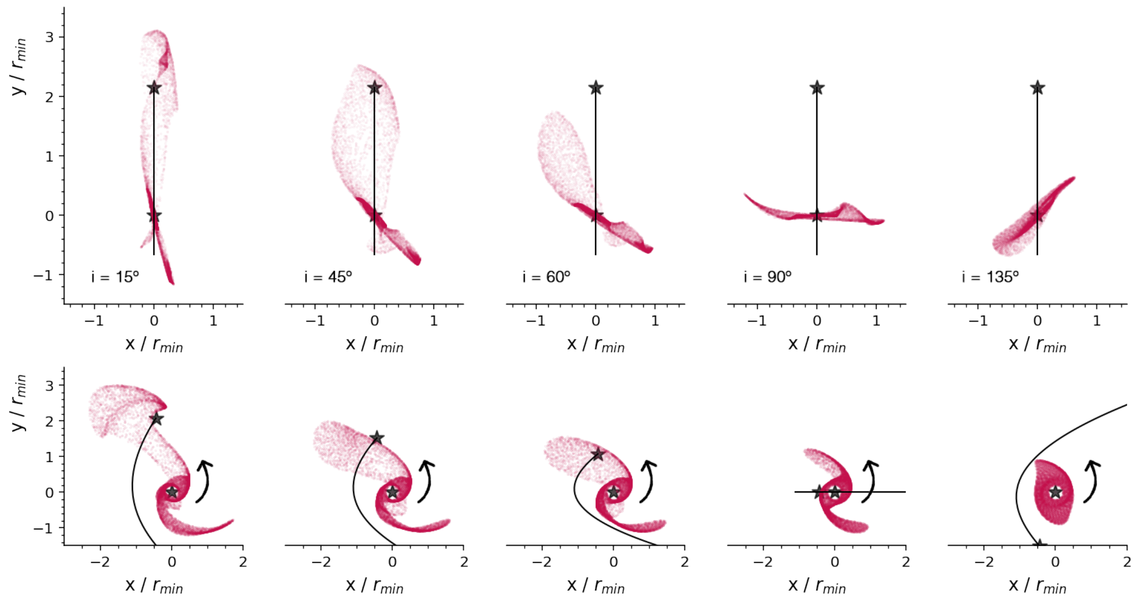


Figure 12. Tail and bridge formation in parabolic encounters of equal mass with $\omega = 0°$ for varying inclination angles i. Each event is plotted at $t = 5$ for a viewing direction along the intersection of the orbital and galactic planes (top row) and normal to the galactic plane (bottom). The 3$^{\mathrm{rd}}$ event from the left is an example of a *faux*-bridge, as it can be seen to not be connected to the companion in the top row but may appear to be in the bottom one.

introduction of two angles, the inclination i[4] and the pericenter argument ω, as illustrated in figure 11. A survey of the inclination, restricted to parabolic encounters of equal mass with $\omega = 0°$, results in a curious phenomenon. Figure 12 shows that bridges only form for low inclinations ($i < 60°$). Higher values lead to *faux*-bridges, as no mass is stolen and they

[4] $i = 0°$ corresponds to a flat prograde encounter; $i = 180°$ to a flat retrograde encounter.

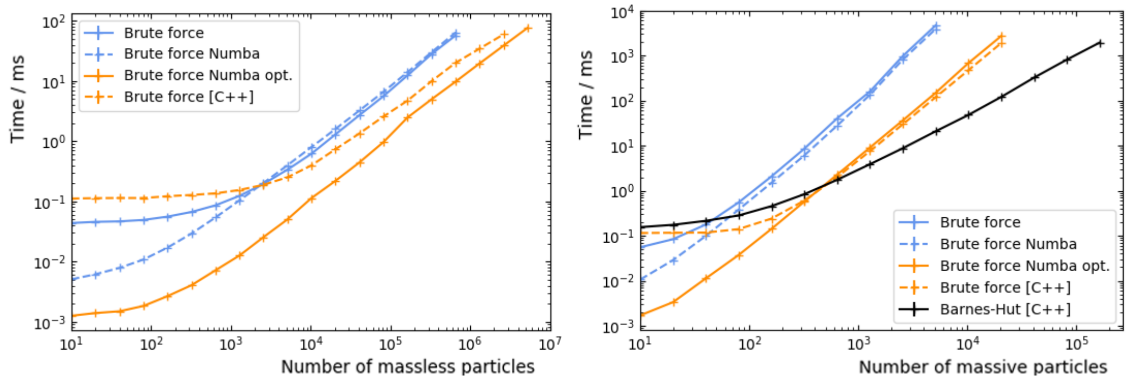


Figure 13. Performance comparison of the methods used to compute the gravitational forces in double precision. A variable number of particles are placed randomly in a cubic box. We show results for a single massive particle and massless test particles (left) and all massive particles (right). "Brute-force Numba" denotes the same code used for the "Brute-Force" algorithm but having compiled it using Numba whereas "Brute-force Numba opt." denotes an optimized version devised for low-level performance, similar in style to "Brute Force `C++`". The Barnes-Hut approximation agrees to within 1% with the exact methods for the shown $\theta_{\max} = 0.7$. All measurements were made in a single thread of a `2.7 GHz Intel Core i5` and have statistical uncertainties comparable to the marker size.

are not connected to the companion but may appear to be from certain viewing angles. Tails can on the other hand be formed for high inclination, with the main difference being that they appear less curved when the galaxy is viewed normal to the spin plane.[5]

3 Dark matter halos

3.1 Computational performance

Although a dark matter halo can be introduced by considering extended matter distributions, as opposed to central point masses, to consider halo-halo interactions and include orbital decay through dynamical friction a full n-body simulation is required. One may consider a brute-force pairwise summation algorithm, but it is clear that its $\mathcal{O}(n^2)$ complexity on the number of massive particles makes it a suboptimal choice. To that end, we implement a Barnes-Hut tree that provides an approximation to the gravitational forces with complexity $\mathcal{O}(n \log n)$. Due to the vast overhead of object creation in Python, we implement the tree in `C++`, compile it to a shared library, and use the standard library module `ctypes` to call the functions from Python. For a fair comparison, we also provide a version of the pairwise summation algorithm written in `C++` and one written in Python but compiled to low-level instructions using *Numba*.[6] Figure 13 shows the expected results,

[5] One could at this point include a survey of ω. We do not find it worthwhile as the resulting tails all look far too similar —the interested reader is referred to [6]— and we would perhaps only highlight that tidal structure formation is inhibited as ω moves away from $0°$ for high inclinations ($i > 60°$). At this point this is more of a blessing, as one can then conceptually superpose two encounters to match astronomical observations by modifying the value of ω, without worrying too much about drastically affecting the tails.

[6] *Numba* is a Python library which *"translates Python functions to optimized machine code at runtime"*.

	mass ratios	eccentricity	no. of particles
	bulge : disk : halo	e	*bulge : disk : halo*
A	1 : 1 : 0	0.5	500 : 500 : 0
B	1 : 3 : 16	0.5	500 : 500 : 0
C	1 : 1 : 0	1.0	500 : 500 : 2000
D	1 : 3 : 16	1.0	500 : 500 : 2000

Table 1. Parameters for the 4 encounters considered in section 3. The total mass of each galaxy and distance of closest approach are kept constant at $r_{\min} = 1$ and $M = 1$. The interaction geometry is loosely inspired by the Antennae, following [7], with $i_1 = i_2 = 60°$ and $\omega_1 = \omega_2 = 30°$.

with the Barnes-Hut algorithm outperforming all other approaches when more than 400 massive particles are used. It is clear that interfacing with `C++` results in an overhead but we expect this to be $\mathcal{O}(n)$, as this is the size of the exchanged arrays. For massless test particles, the use of code optimized for *Numba* ($\mathcal{O}(n)$) leads to the fastest performance due to no interfacing costs.

For a sensible 10 ms computing time step, 10^4 massive or 10^6 massless bodies may be used.[7] The implementation is optimized for memory reusing, loop unrolling and cache hit minimization but further improvements could straightforwardly be achieved through concurrency and Single Instruction Multiple Data (SIMD) techniques or more laboriously by exploiting the GPU.[8] We note that we have implemented basic unit testing for these routines and are thus confident that they are in agreement.

3.2 Orbital decay

Following [7], we use prototype *bulge:disk:dark halo* cold (rotationally supported) galaxies.[9] The bulge is constructed using a Plummer distribution of characteristic length 0.04, the disk using an exponential distribution of decay length 0.2, and the dark matter halo using a NFW profile with $R_s = 1$ and cut-off at $R_0 = 5$. We simulate 4 different encounters, summarized in table 1, elliptical ($e = 0.5$) and parabolic, both with halo and no halo.

Figure 14 shows the trajectory followed by the bulge in all four encounters. A full n-body simulation results in orbital decay in all cases. In fact, dynamical friction is so efficient that even hyperbolic trajectories can decay in a few crossings. It must be noted as such that, whereas a dark halo is not necessary to observe dynamical friction, when included orbital decay may occur even at large separations, greatly enhancing the probability of interactions in the Universe and the importance of galactic mergers.

[7] Scaling to the $\sim 10^{11}$ stars in the Milky Way is unreasonable, as a more accurate SPH simulation would be preferred over mindless scaling. In any case, we note that locally adaptative timestep algorithms would then allow to model close star interactions without significant performance drawbacks.

[8] This, or alternatively the use of BLAS, would also allow one to compute only the upper-half of the skew-hermitian matrices involved. Numpy does not provide specific routines for hermitian and skew-hermitian matrices leading to either wasteful computation or cumbersome code.

[9] We do not evolve the galaxies in isolation and construct them so as to be formally in equilibrium, as opposed to adiabatically. Softening of the gravitational potential is thus needed to minimize two-body relaxation effects.

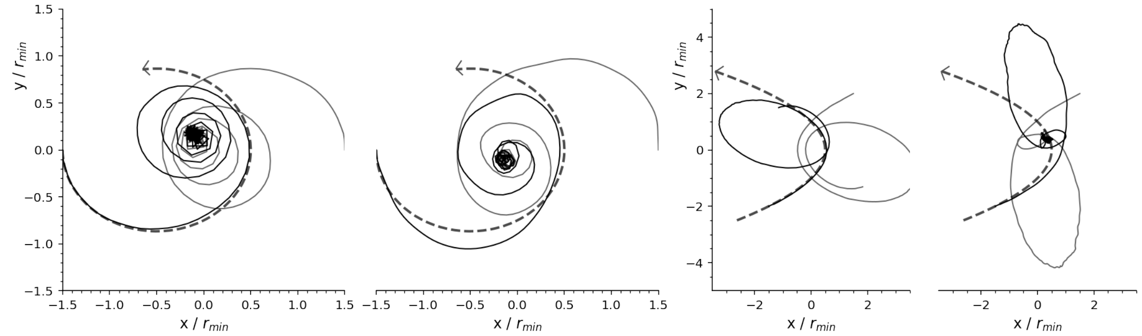


Figure 14. Trajectories of the central bulge for the full n-body simulation. Encounters from left to right: elliptical with no halo (A), elliptical with halo (B), parabolic with no halo (C), parabolic with halo (D). Efficient dynamical friction is observed in all examples. Solid lines represent the bulge trajectories and dashed lines the Keplerian trajectories that would be observed for point galaxies. The view is normal to the orbital plane.

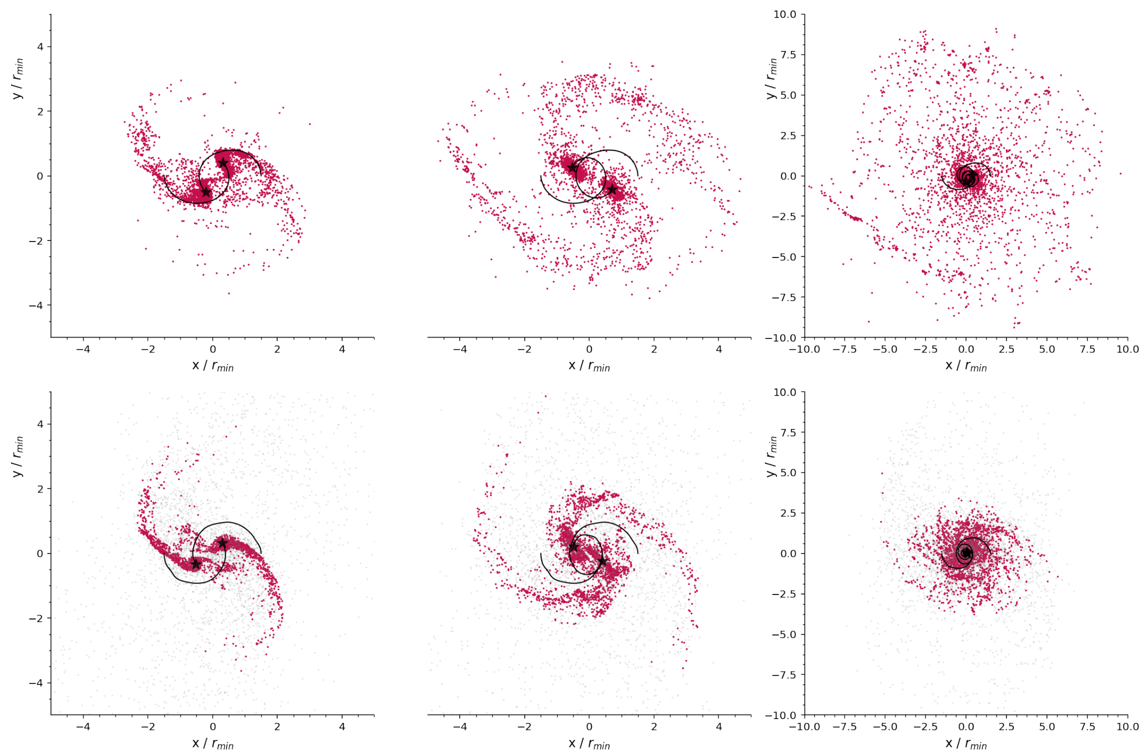


Figure 15. Elliptical encounters with *bulge:disk:halo* mass ratios 1:1:0 (top, A) and 1:3:16 (bottom, B). We show snapshots of the luminous components slightly after the first pericenter (left), slightly before the second pericenter (center) and at large times when the galaxies have merged (right). A full n-body simulation results in the expected orbital decay. A dark halo leads to thinner tails and a more compact merger remnant. The view is normal to the orbital plane.

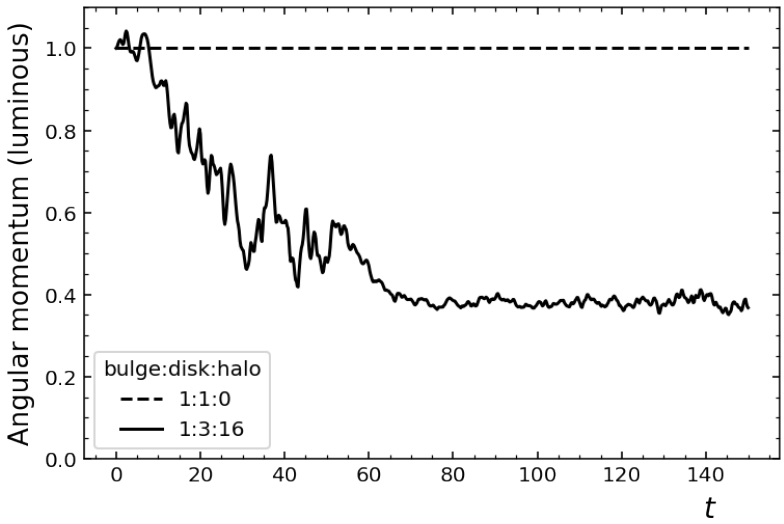


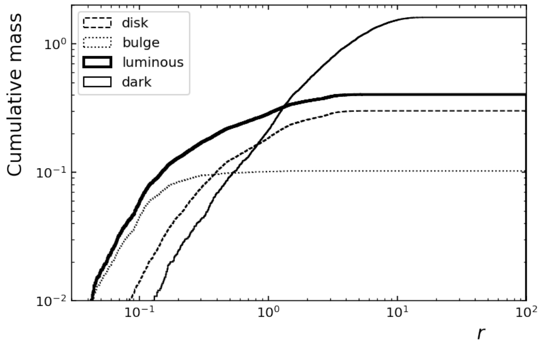
Figure 16. Evolution of the angular momentum of the luminous component for hyperbolic encounters with no dark matter halo (dashed, C) and with a dark matter halo (solid, D). The angular momentum in encounter C is conserved to within relative numerical error of 10^{-10} (providing a further test of the numerical implementation), whereas in D 60% of the angular momentum of the luminous components is transferred away. The exact pairwise summation method is employed for accuracy.

When all luminous components are plotted (figure 15), the halo results in thinner tails, consistent with many observed systems, and a more compact merger remnant. The explanation is straightforward: it provides a mechanism for angular momentum in the disk and bulge to be transferred away. This is confirmed by encounter D (figure 16), where the merger only possesses 40% of the initial angular momentum in the luminous components.

3.3 Structure of a merger remnant

The idea that that two spiral galaxies can merge to create an elliptical galaxy was established as reasonable by Toomre in subsequent work (see the "Toomre sequence" [4]) and first studied closely in the context of dark halos by Barnes [7] (see [12] for a review). Numerically, the study is simple, as one only needs to allow our previous encounter D, the most physically plausible of them all, to evolve until dynamical equilibrium is reached.

The merger is consistent with the properties of a typical elliptical galaxy with principal axes in the ratio $7 : 10 : 13$. One observes in figure 17 that the resulting galaxy has a size only slightly larger than its progenitors, as has been noted before. More interestingly, the resulting merger is well described by *de Vaucouleurs' law*, a commonly used model parameterising the surface brightness I as a function of the distance to the center of the galaxy R as $\log(I) = \log(I_0) - kr^{1/4}$, where I_0 and k are constants. This provides evidence for the now accepted theory that elliptical galaxies may originate from the merging of other galaxies. Based on this idea, CDM models propose the hierarchical scenario, where many large scale structures in the Universe can be explained through successive gravitational



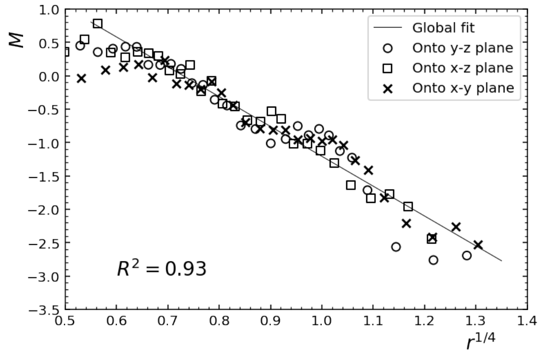


Figure 17. Cumulative mass distribution of the resultant merger for encounter D (left) and magnitude $M = \log(\text{surface density})$ (in arbitrary units) as a function of $r^{1/4}$ (right). The surface density is well-described by *de Vaucouleurs' law*, as has been reported multiple times ([7] and references therein). The 5 closest and 2 furthest points for each series are omitted in the fit, as they are highly dependent on the original bulge distribution and suffer from high uncertainty in the determination of the radii.

collapse due to instabilities and galactic mergers.

4 The Antennae galaxies

4.1 Analysis & implementation: Automated matching

The Antennae galaxies are one of the best studied mergers due to their proximity [19, 20], allowing for the result of the simulations to be compared to data in great detail. Owing to this, and further motivated by their beauty, we aim in this section to explain how the pair could have arisen. Our target is to find a set of parameters consistent with the observations through an algorithm that scans the parameter space without any human intervention. This is an area of active research (Identikit [21, 22], AGC [23]) where simplified simulations are used and humans still commonly carry out part of the process manually [24]. In the case of the Antennae, the state of the art simulations are to this day largely based on the parameters proposed by Toomre & Toomre [14].

We experiment with Bayesian optimization and genetic algorithms, but find them to suffer from boundary issues[10] and unnecessary hyperparameter complexity for the problem. We settle for Simulated Annealing due to its ability to handle a relatively large number of parameters (table 2) and to converge without too many evaluations.

In detail, we select a sensible range for each allowed parameter and draw the first sample at random.[11] We choose an exponential cooling scheme for the *temperature* and in every

[10] The algorithm repeatedly evaluates (less likely optimal) points near the boundaries, particularly when Upper Confidence Boundary acquisition functions are employed. For a large number of parameters this is computationally wasteful and whereas proposed solutions exist [25] this would take us too far into the field of black-box optimization.

[11] To lower the number of parameters and ensure performance, we make use of the simplified model from section 2, with massless test particles in a uniform ring, deemed appropriate for obtaining a plausible set of geometrical parameters.

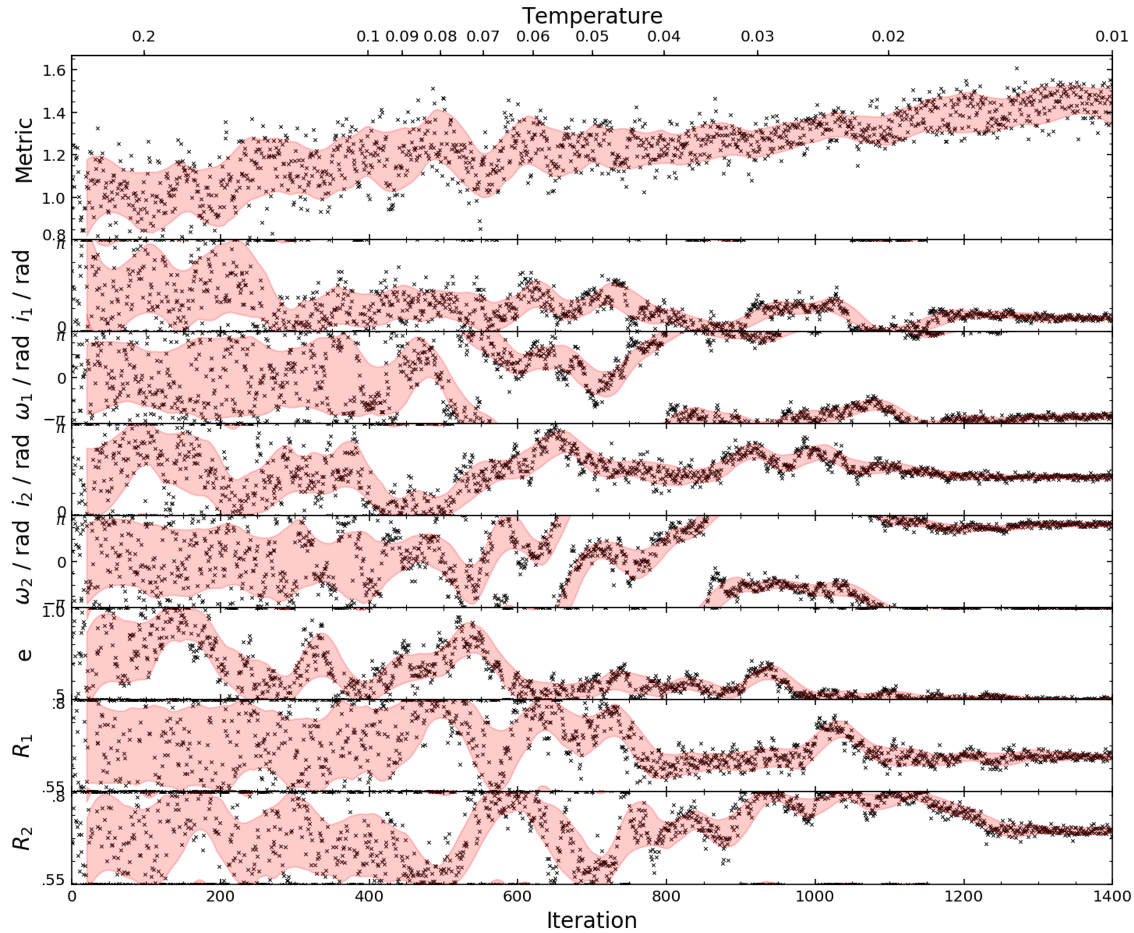


Figure 18. Evolution of the simulated annealing algorithm. Each dot shows a single evaluation, with the performance metric shown in the first row (higher is better) and the initializing parameters in all the following. The shaded regions indicate the 10%−90% confidence bands. Note that there is a significant error ($\sim \pm 0.04$) in each metric evaluation, since it involves probing multiple displaying parameters stochastically. The algorithm is seen to converge to a set of parameters, with a generally upwards trend for the metric, providing evidence for its correctness.

iteration perturb each parameter of the current best result by adding numbers drawn from a Gaussian distribution with standard deviation linearly proportional to the temperature. The score for each evaluation is the F_1 score between two low resolution binarized images: the ground truth derived from astronomical observations and a 2D projection of the simulation.[12] The choice is driven by the necessity for an extremely fast performance evaluation, as each simulation must be compared to the ground truth at multiple time instants (t), viewing directions (θ, ϕ), possible rotations along the line of sight (Ω), scalings (s) and

[12] The F_1 score is the harmonic average of the sensitivity and recall and matched human expectations of what constituted a good match. We segment the ground truth into two galaxies, match them separately and add the two scores.

Parameter	Allowed range	Final value
Mass ratio^a	1.0	1.0
Eccentricity	0.5 - 1.0	0.5
1st galaxy:		
Inclination (i_1)	0 - π	26.3°
Pericenter arg. (ω_1)	$-\pi$ - π	$-153.0°$
Disk radius (R_1)	0.55 - 0.8	0.65
2nd galaxy:		
Inclination (i_2)	0 - π	76.8°
Pericenter arg. (ω_2)	$-\pi$ - π	147.3°
Disk radius (R_2)	0.55 - 0.8	0.70 (0.64)
Viewing:		
θ	0 - π	(1.86)
ϕ	0 - 2π	(4.10)
Ω	0 - 2π	(5.10)
Scaling^b	-	(12.0 kpc)
$(x, y)^c$ offset	-	-
HI spectrum:		
Velocity offset	-	(43 km/s)
Velocity scaling^c	-	(150 km/s)

Table 2. Parameters matched by the simulated annealing algorithm. The velocity offset and velocity scaling are not matched by the algorithm but introduced here for later use. We show in parenthesis those parameters which we modify or select manually. ^{a}We set the ratio of their masses to 1 as astronomical observations and previous work suggest this is reasonable [14]. ^{b}One dimensionless unit in the simulation corresponds in this case to the distance and velocity given in the table for the assumed 22 Mpc distance to the Antennae with total mass for each galaxy $5 \times 10^{11} M_\odot$, following [14]. ^{c}The offset is physically meaningless here as it depends on the framing of the ground truth image, but must nevertheless be matched.

translations (x, y) to obtain the best match. In fact, despite each single image comparison being optimized to take $20\mu s$, the simulation itself only amounts to $\sim 10\%$ of the computing cost.[13] The correctness of the algorithm is examined by globally optimizing several single- and multi-dimensional analytical functions with additive Gaussian distributed noise.

4.2 Observations and the Hydrogen 21-cm line

Figure 18 shows the results of running the simulated annealing algorithm for 1400 samples ($\sim$ 2 days of computing time). The final parameters are those of the best run and are tabulated in table 2. The low eccentricity may be particularly worrying as a low period elliptical orbit could hardly have been the case. In reality, having neglected orbital decay, it

[13] Among the other explored metrics, we highlight the Wasserstein distance, obtained by solving an optimal transport problem, which matched human expectations better and can be made translationally invariant but is too computationally expensive to evaluate.

was expected that a closer despite unphysical match would result from elliptical encounters. It is due to this same omission of a dark matter halo that many previous approaches ran simulations with $e = 0.5$ [6]. The final model, which qualitatively matches the Antennae except for a small number of stray particles next to one of the disks is shown in figure 19. The parameters found agree only partially with those provided originally by Toomre and Toomre [6]. When taken to follow their convention,[14] our parameters read $i_1 = 26.3°$, $i_2 = 76.8°$, $\omega_1 = 27.0°$, $\omega_2 = -32.7°$ compared to theirs $i_1 = i_2 = 60°$, $\omega_1 = \omega_2 = -30°$. We believe this is due to the problem being under-constrained with only one observation viewpoint.

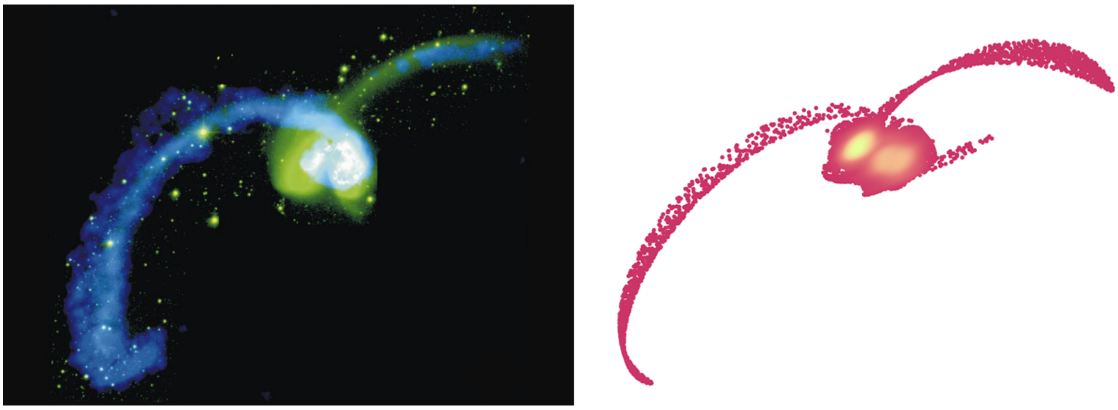


Figure 19. Comparison between observations of the Antennae galaxies obtained from [20] (left), from which the low resolution binarised ground truth is obtained, and the result of our best simulation at time t=13.5 (right). 20,000 test masses are included per galaxy for plotting purposes with brightness indicating the density of each region. The observations combine HI data (green) with optical images (white and blue).

Moreover, we compare our model to HI observations and find a surprisingly good agreement (figure 20), including the "twists" at the tail ends.[15] We emphasize here that our simulated annealing algorithm did not attempt to match the line of sight velocity observations in any way, and as such this independent test gives us confidence in our model. Even more surprisingly, the match is similar to that of recent SPH models (including radiative cooling, star formation and feedback from Type II supernovae) [14]. It would be unfair to put these two models at the same level, as the main reason for the "extra machinery" is to probe bursts of star formation in the overlapping region,[16] but it is certainly now reasonable to say that Toomre & Toomre [6] couldn't have been more right when they wrote that "[these structures are] in essence kinematic".

[14] Toomre and Toomre allow i to take negative values but restrict ω to be $|\omega| < 90°$.

[15] We expect that a closer reproduction of the features of the tails would require a more realistic model of the galaxies (bulge + disk + halo, see section 3), but the number of parameters would become intractable.

[16] The Antennae are currently undergoing a starburst phase in the overlap region, a feature that has proven hard to replicate, as many simulations predict enhanced star formation at the galactic centers instead [26]. It has been suggested that collisions of Giant Molecular Clouds could account for this feature [26, 27].

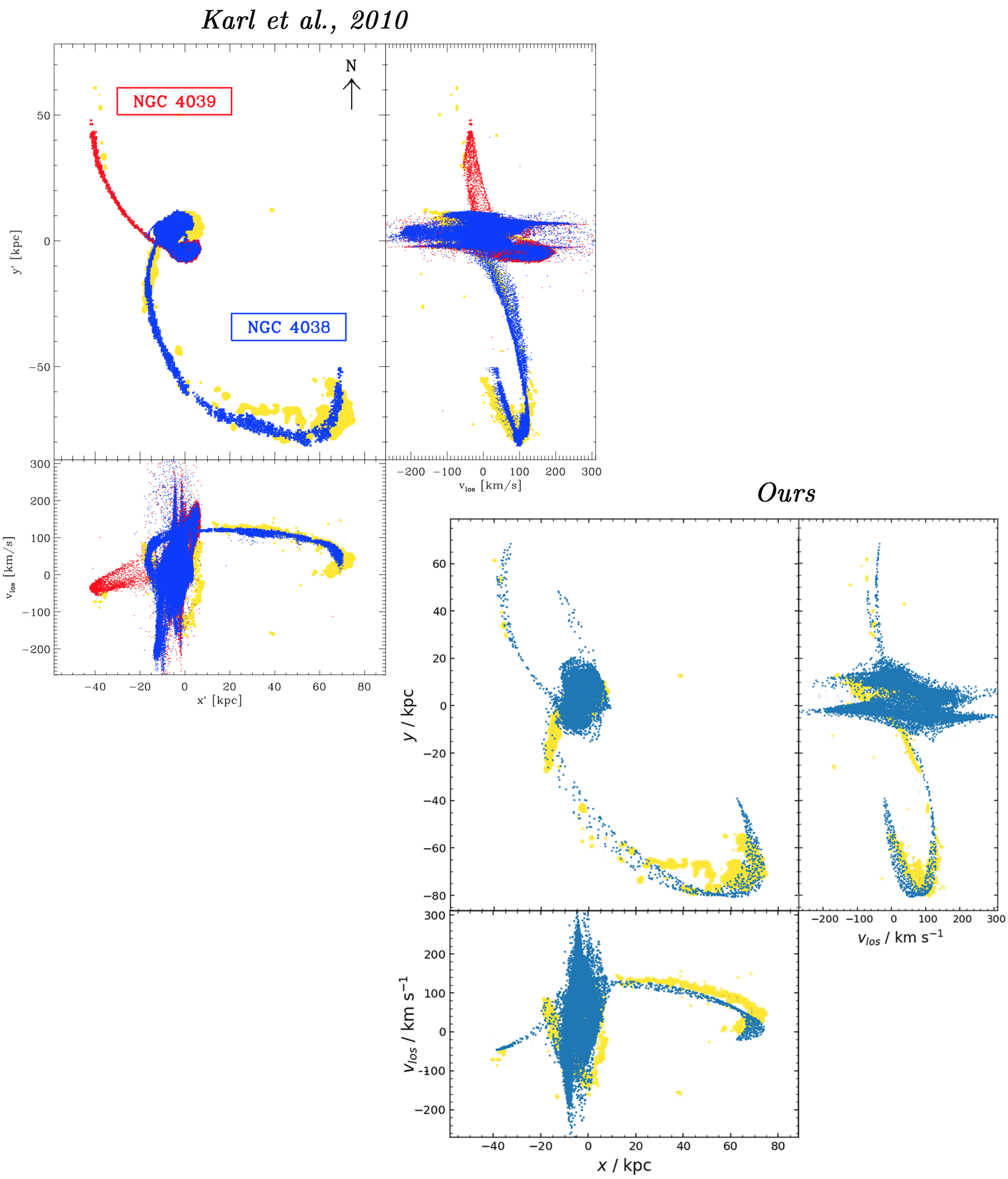


Figure 20. Comparison to HI kinematic data from [20]. Yellow points represent the observational data, blue and red the model. We show the results from *Karl et al., 2010* [14] (top left, Soft Particle Hydrodynamics + Star formation + Radiative cooling + Type II Supernovae feedback) and of our best model (bottom right). The velocity offset and scaling were matched manually. The initial radius of the second galaxy was modified slightly (table 2).

[2993 words to this mark]

5 Conclusions

We have shown that a simplified model where two galaxies, surrounded by massless rings of particles, interact is sufficient to reproduce many of the phenomena associated with these encounters, emphasizing that bridges, tails and counterarms are at heart a kinematic phenomenon. Bridges are found to be transient and a feature of soft perturbations whereas tails require the perturbing galaxy to be of similar size. Both are however dependent on *spin-orbit coupling*, i.e. the alignment of the spin of the galaxy with the angular momentum of the orbit, as bridges, and to a lesser extent tails, are inhibited by high inclination encounters.

We also consider self-gravitating rotationally supported models and find them to exhibit dynamical friction. A dark matter halo provides an efficient mechanism for orbital decay and for angular momentum to be transferred away from the luminous components, leading to elliptical merger remnants that follow *de Vaucouleur's* law. Finally, we provide a simple model for the Antennae galaxies obtained by sampling the high dimensional space without human intervention, comparable to current semi-supervised and unsupervised approaches ([21–23]). The obtained model matches observational data of the HI spectroscopic line to great accuracy. The main open area remains a study of star formation, which would require a SPH simulation.

In any case, the idea that galactic interactions account for the morphology of some peculiar galaxies is well established. More interesting open topics include whether cosmological simulations based on Cold Dark Matter models (ΛCDM) can lead to a structured formation of the universe through galactic mergers, and the study of dwarf galaxy formation in the tidal tails [28].

A Source code listing

Legible documentation, automatically generated using `pdoc` from the docstrings, can be found in the `docs/` folder. We structure the code in an Object Oriented manner, to allow for our modules to be reused, and follow appropriate style conventions. The simplest way to run an encounter is through the command line:

```
> python run_simulation.py config.yml --output_folder --verbose
```

We make use of configuration files in YAML (`.yml`) format. These are extremely simple and concise, as one only needs to specify those parameters that differ from the default (`config/default.yml`). For example, the encounter proposed for the Antennae in [6] can be specified as:

.

```yaml
name: toomre1972
galaxy1:
  orientation: [240, -30]
  disk:
    l: .7
galaxy2:
  orientation: [120, -30]
  disk:
    particles: 2000
    l: .7
```

A .yml can contain multiple encounters separated with three dashes (as per usual YAML standard). In this case the extra encounters need only specify those parameters that change with respect to the first encounter, making parameter surveys simple. .yml files for the encounters we consider can be found in the `config/` folder. The core of the simulation (`Simulation, Galaxy`) can be found in the file `simulation.py`, and makes use of the routines defined in `acceleration.py`. The gravitational computation routines are general —allowing some particles to be set as massless— and have been optimized to take advantage of compiler loop unrolling and vectorization as well as to minimize cache hits and memory allocations for performance.

The C++ library can be found in the `cpp/` folder and the simulated annealing algorithm is contained in `run_simulated_annealing.py`. Moreover, the submission includes an interactive widget (`analysis/interactive.ipynb`) that can be used to examine the encounters. Its interface is shown in figure 21.

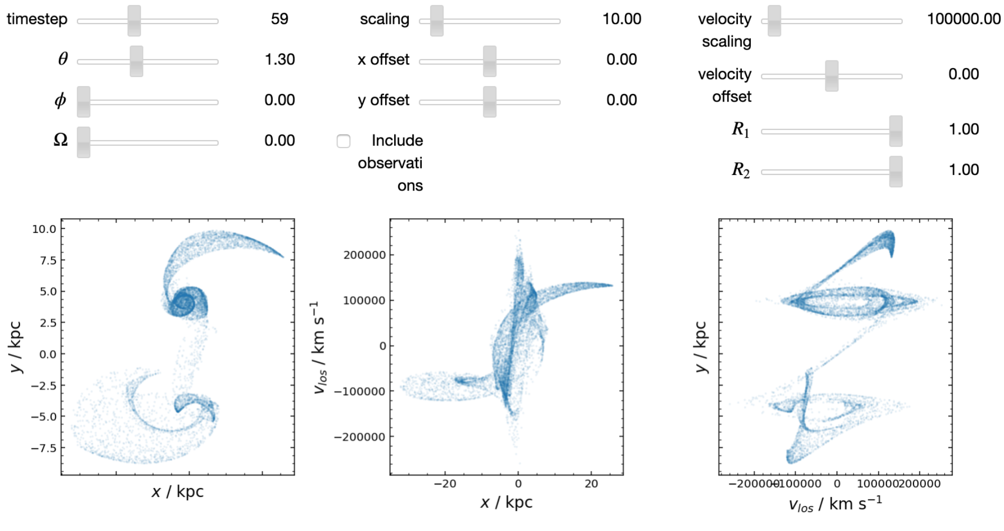


Figure 21. Interface of the interactive widget `analysis/interactive.ipynb` that can be used to analyse the encounters. It allows the time and viewing direction to be varied. For massless particles in rings, the radii of the galaxies can also be modified without rerunning the simulation. Observational data of the Antenna Hydrogen 21-cm line can be matched using this tool.

Due to their obvious length we do not reproduce here helper functions (`utils.py`), statistical distributions (`distributions.py`) and analysis and plotting code (`analysis/`), except for the segmentation algorithm (`segmentation.py`) described in section 2.3. These can be found in the online submission and should be ran from the main module, that is for instance:

```
> python -m analysis.tailShapePlot
```

Configuration and running encounters

config/default.yml


```yaml
---
# When calling > python simulation.py filename.yml --output_folder
# the simulation will use the default parameters here unless specified
name: default

simulation:
  dt: 0.001 #timestep of the simulation
  tmax: 15 #total runtime of the simulation
  soft: 0.1 #plummer softening characteristic length
  saveEvery: 100 #the state of the simulation is saved every saveEvery steps
  method: bruteForce #method for computing gravitational forces
  # One of 'bruteForce', 'bruteForceNumba',
  # bruteForceNumbaOptimized', 'bruteForceCPP', 'barnesHutCPP'.

orbit:
  e: 1 #eccentricity
  rmin: 1 #separation at pericenter
  R0: 4 #separation at t=0

galaxy1:
  orientation: [0, 0] #[theta, phi] in degrees
  # These are related to i, ω through theta = i + 180 and ω = phi
  centralMass: 1 #mass of the central point object
  bulge:
    model: plummer #alternatively: hernquist
    totalMass: 0
    particles: 0 #number of particles
    l: .04 #characteristic length scale both for plummer and Hernquist models
  disk:
    model: uniform #alternatively: rings, exp
    totalMass: 0
    particles: 2000 #number of particles
    l: 0.8 #for uniform: single number for maximum radius
    #l: [0., .7, 100] #for rings: [closest ring, furthest ring, number of rings]
    #l: .2 #for exp: characteristic decay length
  halo:
    model: NFW #Navarro-Frenk-White profile only
    totalMass: 0
    particles: 0 #number of particles
    rs: 1 #characteristic length scale of NFW. Cutoff is 5*rs

# The same options are available for the second galaxy
galaxy2:
  orientation: [0, 0]
  centralMass: 1
  bulge:
    model: plummer
    totalMass: 0
    particles: 0
    l: .04
  disk:
    model: uniform
    totalMass: 0
    particles: 0 #the second galaxy does not possess a ring by default
    l: 0.7
  halo:
    model: NFW
    totalMass: 0
    particles: 0
    rs: 1
```


run_simulation.py

```python
"""Command line tool to run a YAML simulation configuration file."""

import yaml
import argparse

from utils import update_config
from simulation import Simulation, Galaxy


# Parse command line arguments:
# > python simulation.py config_file.yml output_folder
parser = argparse.ArgumentParser(description='''Run a galactic
        collision simulation.''')
parser.add_argument('config_file', type=argparse.FileType('r'),
        help='''Path to configuration file for the simulation, in YAML format.
        See the config folder for examples.''')
parser.add_argument('--output_folder', default=None,
        help='''Name of the output folder in data/ where the results will be
        saved. The directory will be created if necessary. If none is provided,
        the name attribute in the configuration file will be used instead.''')
parser.add_argument('--verbose', action='store_true', default=False,
        help='''In verbose mode the simulation will print its progress.''')

args = parser.parse_args()

# Load the configuration for this simulation
CONFIG = yaml.load(open("config/default.yml", "r")) # default configuration
updates = list(yaml.load_all(args.config_file))

for update in updates:
```

```python
    # For multiple configurations in one file,
    # the updates are with respect to the first one.
    update_config(CONFIG, updates[0])
    update_config(CONFIG, update)
    # If no output folder is provided, the name in CONFIG is used instead
    outputFolder = (CONFIG['name'] if args.output_folder is None
        else args.output_folder)

    # Run the simulation
    sim = Simulation(**CONFIG['simulation'], verbose=args.verbose,
        CONFIG=CONFIG)
    galaxy1 = Galaxy(**CONFIG['galaxy1'], sim=sim) # create the galaxies
    galaxy2 = Galaxy(**CONFIG['galaxy2'], sim=sim)
    sim.setOrbit(galaxy1, galaxy2, **CONFIG['orbit']) # define the orbit
    sim.run(**CONFIG['simulation'], outputFolder=outputFolder)
```

Numerical components

acceleration.py

```python
"""Defines the possible routines for computing the gravitational forces in the
simulation.

All the methods in this file require a position (n, 3) vector, a mass (n, )
vector and an optional softening scale float."""

import ctypes
import numpy.ctypeslib as ctl
import numpy as np
from numba import jit


def bruteForce(r_vec, mass, soft=0.):
    """Calculates the acceleration generated by a set of masses on themselves.
      Complexity O(n*m) where n is the total number of masses and m is the
      number of massive particles.

    Parameters:
        r_vec (array): list of particles positions.
            Shape (n, 3) where n is the number of particles
        mass (array): list of particles masses.
            Shape (n,)
        soft (float): characteristic plummer softening length scale
    Returns:
        forces (array): list of forces acting on each particle.
            Shape (n, 3)
    """
    # Only calculate forces from massive particles
    mask = mass!=0
    massMassive = mass[mask]
    rMassive_vec = r_vec[mask]
    #  x m x 1 matrix (m = number of massive particles) for broadcasting
    mass_mat = massMassive.reshape(1, -1, 1)
    # Calculate displacements
    # r_ten is the direction of the pairwise displacements. Shape (n, m, 3)
    # r_mat is the absolute distance of the pairwise displacements. (n, m, 1)
    r_ten = rMassive_vec.reshape(1, -1, 3) - r_vec.reshape(-1, 1, 3)
    r_mat = np.linalg.norm(r_ten, axis=-1, keepdims=True)
    # Avoid division by zeros
    # $a = M/(r+\epsilon)^2$, where $\epsilon$ is the softening scale
    # r_ten/r_mat gives the direction unit vector
    accel = np.divide(r_ten * mass_mat/(r_mat+soft)**2, r_mat,
        where=r_ten.astype(bool), out=r_ten) # Reuse memory from r_ten
    return accel.sum(axis=1) # Add all forces on each particle

@jit(nopython=True) # Numba annotation
def bruteForceNumba(r_vec, mass, soft=0.):
    """Calculates the acceleration generated by a set of masses on themselves.
    It is done in the same way as in bruteForce, but this
    method is ran through Numba"""
    mask = mass!=0
    massMassive = mass[mask]
    rMassive_vec = r_vec[mask]
    mass_mat = massMassive.reshape(1, -1, 1)
    r_ten = rMassive_vec.reshape(1, -1, 3) - r_vec.reshape(-1, 1, 3)
    # Avoid np.linalg.norm to allow Numba optimizations
    r_mat = np.sqrt(r_ten[:,:,0:1]**2 + r_ten[:,:,1:2]**2 + r_ten[:,:,2:3]**2)
    r_mat = np.where(r_mat == 0, np.ones_like(r_mat), r_mat)
    accel = r_ten/r_mat * mass_mat/(r_mat+soft)**2
    return accel.sum(axis=1) # Add all forces in each particle

@jit(nopython=True) # Numba annotation
def bruteForceNumbaOptimized(r_vec, mass, soft=0.):
    """Calculates the acceleration generated by a set of masses on themselves.
    This is optimized for high performance with Numba. All massive particles
    must appear first."""
    accel = np.zeros_like(r_vec)
    # Use superposition to add all the contributions
    n = r_vec.shape[0] # Number of particles
    delta = np.zeros((3,)) # Only allocate this once
    for i in range(n):
        # Only consider pairs with at least one massive particle i
        if mass[i] == 0: break
        for j in range(i+1, n):
            # Explicitely separate components for high performance
            # i.e. do not do delta = r_vec[j] - r_vec[i]
```

```
            # (The effect of this is VERY relevant (x10) and has to do with
            # memory reallocation) Numba will vectorize the loops.
            for k in range(3): delta[k] = r_vec[j,k] - r_vec[i,k]
            r = np.sqrt(delta[0]*delta[0] + delta[1]*delta[1] + delta[2]*delta[2])
            tripler = (r+soft)**2 * r

            # Compute acceleration on first particle
            mr3inv = mass[i]/(tripler)
            # Again, do NOT do accel[j] -= mr3inv * delta
            for k in range(3): accel[j,k] -= mr3inv * delta[k]

            # Compute acceleration on second particle
            # For pairs with one massless particle, no reaction force
            if mass[j] == 0: break
            # Otherwise, opposite direction (+)
            mr3inv = mass[j]/(tripler)
            for k in range(3): accel[i,k] += mr3inv * delta[k]
    return accel

# C++ interface, load library
ACCLIB = None
def loadCPPLib():
    """Loads the C++ shared library to the global variable ACCLIB. Must be
    called before using the library."""
    global ACCLIB
    ACCLIB = ctypes.CDLL('cpp/acclib.so')
    # Define appropiate types for library functions
    doublepp = np.ctypeslib.ndpointer(dtype=np.uintp) # double**
    doublep = ctl.ndpointer(np.float64, flags='aligned, c_contiguous')#double*
    # Check cpp/acclib.cpp for function signatures
    ACCLIB.bruteForceCPP.argtypes = [doublepp, doublep,
        ctypes.c_int, ctypes.c_double]
    ACCLIB.barnesHutCPP.argtypes = [doublepp, doublep,
        ctypes.c_int, ctypes.c_double, ctypes.c_double,
        ctypes.c_double, ctypes.c_double, ctypes.c_double]

def bruteForceCPP(r_vec, m_vec, soft=0.):
    """Calculates the acceleration generated by a set of masses on themselves.
    This is ran in a shared C++ library through Brute Force (pairwise sums)
    Massive particles must appear first."""
    # Convert array to data required by C++ library
    if ACCLIB is None: loadCPPLib() # Singleton pattern
    # Change type to be appropiate for calling library
    r_vec_c = (r_vec.ctypes.data + np.arange(r_vec.shape[0])
        * r_vec.strides[0]).astype(np.uintp)
    # Set return type as double*
    ACCLIB.bruteForceCPP.restype = np.ctypeslib.ndpointer(dtype=np.float64,
        shape=(r_vec.shape[0]*3,))
    # Call the C++ function: double* bruteForceCPP
    accel =  ACCLIB.bruteForceCPP(r_vec_c, m_vec, r_vec.shape[0], soft)
    # Change shape to get the expected Numpy array (n, 3)
    accel.shape = (-1, 3)
    return accel

def barnesHutCPP(r_vec, m_vec, soft=0.):
    """Calculates the acceleration generated by a set of masses on themselves.
    This is ran in a shared C++ library using a BarnesHut tree"""
    # Convert array to data required by C++ library
    if ACCLIB is None: loadCPPLib() # Singleton pattern
    # Change type to be appropiate for calling library
    r_vec_c = (r_vec.ctypes.data + np.arange(r_vec.shape[0])
        * r_vec.strides[0]).astype(np.uintp)
    # Set return type as double*
    ACCLIB.barnesHutCPP.restype = np.ctypeslib.ndpointer(dtype=np.float64,
        shape=(r_vec.shape[0]*3,))
    # Explicitely pass the corner and size of the box for the top node
    px, py, pz = np.min(r_vec, axis=0)
    size = np.max(np.max(r_vec, axis=0) - np.min(r_vec, axis=0))
    # Call the C++ function: double* barnesHutCPP
    accel =  ACCLIB.barnesHutCPP(r_vec_c, m_vec, r_vec.shape[0],
        size, px, py, pz, soft)
    # Change shape to get the expected Numpy array (n, 3)
    accel.shape = (-1, 3)
    return accel
```

simulation.py

```
"""Definition of the Simulation class and the Galaxy constructor."""

import os
import pickle
import numpy as np
import matplotlib.pyplot as plt

from utils import random_unit_vectors, cascade_round
from distributions import PLUMMER, HERNQUIST, UNIFORM, EXP, NFW
import acceleration


###########################################################################
###########################################################################
class Simulation:
    """"Main class for the gravitational simulation.

    Attributes:
        r_vec (array): position of the particles in the current timestep.
            Shape: (number of particles, 3)
```

```python
        rprev_vec (array): position of the particles in the previous timestep.
            Shape: (number of particles, 3)
        v_vec (array): velocity in the current timestep.
            Shape: (number of particles, 3)
        a_vec (array): acceleration in the current timestep.
            Shape: (number of particles, 3)
        mass (array): mass of each particle in the simulation.
            Shape: (number of particles,)
        type (array): non-unique identifier for each particle.
            Shape: (number of particles,)
        tracks (array): list of positions through the simulation for central
            masses. Shape: (tracked particles, n+1, 3).
        CONFIG (array): configuration used to create the simulation.
            It will be saved along the state of the simulation.

        dt (float): timestep of the simulation
        n (int): current timestep. Initialized as n=0.
        soft (float): softening length used by the simulation.
        verbose (boolean): When True progress statements will be printed.
    """

    def __init__(self, dt, soft, verbose, CONFIG, method, **kwargs):
        """Constructor for the Simulation class.

        Arguments:
            dt (float): timestep of the simulation
            n (int): current timestep. Initialized as n=0.
            soft (float): softening length used by the simulation.
            verbose (bool): When True progress statements will be printed.
            CONFIG (dict): configuration file used to create the simulation.
            method (string): Optional. Algorithm to use when computing the
                gravitational forces. One of 'bruteForce', 'bruteForce_numba',
                'bruteForce_numbaopt', 'bruteForce_CPP', 'barnesHut_CPP'.
        """
        self.n = 0
        self.t = 0
        self.dt = dt
        self.soft = soft
        self.verbose = verbose
        self.CONFIG = CONFIG
        # Initialize empty arrays for all necessary properties
        self.r_vec = np.empty((0, 3))
        self.v_vec = np.empty((0, 3))
        self.a_vec = np.empty((0, 3))
        self.mass = np.empty((0,))
        self.type = np.empty((0, 2))
        algorithms = {
            'bruteForce': acceleration.bruteForce,
            'bruteForceNumba': acceleration.bruteForceNumba,
            'bruteForceNumbaOptimized': acceleration.bruteForceNumbaOptimized,
            'bruteForceCPP': acceleration.bruteForceCPP,
            'barnesHutCPP': acceleration.barnesHutCPP
        }
        try:
            self.acceleration = algorithms[method]
        except: raise Exception("Method '{}' unknown".format(method))

    def add(self, body):
        """Add a body to the simulation. It must expose the public attributes
           body.r_vec, body.v_vec, body.a_vec, body.type, body.mass.

        Arguments:
            body: Object to be added to the simulation (e.g. a Galaxy object)
        """
        # Extend all relevant attributes by concatenating the body
        for name in ['r_vec', 'v_vec', 'a_vec', 'type', 'mass']:
            simattr, bodyattr = getattr(self, name), getattr(body, name)
            setattr(self, name, np.concatenate([simattr, bodyattr], axis=0))
        # Order based on mass
        order = np.argsort(-self.mass)
        for name in ['r_vec', 'v_vec', 'a_vec', 'type', 'mass']:
            setattr(self, name, getattr(self, name)[order])

        # Update the list of objects to keep track of
        self.tracks = np.empty((np.sum(self.type[:,0]=='center'), 0, 3))

    def step(self):
        """Perform a single step of the simulation.
           Makes use of a 4th order Verlet integrator.
        """
        # Calculate the acceleration
        self.a_vec = self.acceleration(self.r_vec, self.mass, soft=self.soft)
        # Update the state using the Verlet algorithm
        # (A custom algorithm is written mainly for learning purposes)
        self.r_vec, self.rprev_vec = (2*self.r_vec - self.rprev_vec
            + self.a_vec * self.dt**2, self.r_vec)
        self.n += 1
        # Update tracks
        self.tracks = np.concatenate([self.tracks,
            self.r_vec[self.type[:,0]=='center'][:,np.newaxis]], axis=1)

    def run(self, tmax, saveEvery, outputFolder, **kwargs):
        """Run the galactic simulation.

        Attributes:
            tmax (float): Time to which the simulation will run to.
```

```python
                This is measured here since the start of the simulation,
                not since pericenter.
            saveEvery (int): The state is saved every saveEvery steps.
            outputFolder (string): It will be saved to /data/outputFolder/
        """
        # When the simulation starts, intialize self.rprev_vec
        self.rprev_vec = self.r_vec - self.v_vec * self.dt
        if self.verbose: print('Simulation starting. Bon voyage!')
        while(self.t < tmax):
            self.step()
            if(self.n % saveEvery == 0):
                self.save('data/{}'.format(outputFolder))

        print('Simulation complete.')

    def save(self, outputFolder):
        """Save the state of the simulation to the outputFolder.
           Two files are saved:
                sim{self.n}.pickle: serializing the state.
                sim{self.n}.png: a simplified 2D plot of x, y.
        """
        # Create the output folder if it doesn't exist
        if not os.path.exists(outputFolder): os.makedirs(outputFolder)

        # Compute some useful quantities
        # v_vec is not required by the integrator, but useful
        self.v_vec = (self.r_vec - self.rprev_vec)/self.dt
        self.t = self.n * self.dt # prevents numerical rounding errors

        # Serialize state
        file = open(outputFolder+'/data{}.pickle'.format(self.n), "wb")
        pickle.dump({'r_vec': self.r_vec, 'v_vec': self.v_vec,
                     'type': self.type, 'mass': self.mass,
                     'CONFIG': self.CONFIG, 't': self.t,
                     'tracks': self.tracks}, file)

        # Save simplified plot of the current state.
        # Its main use is to detect ill-behaved situations early on.
        fig = plt.figure()
        plt.xlim(-5, 5); plt.ylim(-5, 5); plt.axis('equal')
        # Dark halo is plotted in red, disk in blue, bulge in green
        PLTCON = [('dark', 'r', 0.3), ('disk', 'b', 1.0), ('bulge', 'g', 0.5)]
        for type_, c, a in PLTCON:
            plt.scatter(self.r_vec[self.type[:,0]==type_][:,0],
                self.r_vec[self.type[:,0]==type_][:,1], s=0.1, c=c, alpha=a)
        # Central mass as a magenta star
        plt.scatter(self.r_vec[self.type[:,0]=='center'][:,0],
            self.r_vec[self.type[:,0]=='center'][:,1], s=100, marker="*", c='m')
        # Save to png file
        fig.savefig(outputFolder+'/sim{}.png'.format(self.n), dpi=150)
        plt.close(fig)

    def project(self, theta, phi, view=0):
        """Projects the 3D simulation onto a plane as viewed from the
           direction described by the (theta, phi, view). Angles in radians.
           (This is used by the simulated annealing algorithm)

        Parameters:
            theta (float): polar angle.
            phi (float): azimuthal angle.
            view (float): rotation along line of sight.
        """
        M1 = np.array([[np.cos(phi), np.sin(phi), 0],
                       [-np.sin(phi), np.cos(phi), 0],
                       [0, 0, 1]])
        M2 = np.array([[1, 0, 0],
                       [0, np.cos(theta), np.sin(theta)],
                       [0, -np.sin(theta), np.cos(theta)]])
        M3 = np.array([[np.cos(view), np.sin(view), 0],
                       [-np.sin(view), np.cos(view), 0],
                       [0, 0, 1]])

        M = np.matmul(M1, np.matmul(M2, M3)) # combine rotations
        r = np.tensordot(self.r_vec, M, axes=[1, 0])

        return r[:,0:2]

    def setOrbit(self, galaxy1, galaxy2, e, rmin, R0):
        """Sets the two galaxies galaxy1, galaxy2 in an orbit.

        Parameters:
            galaxy1 (Galaxy): 1st galaxy (main)
            galaxy2 (Galaxy): 2nd galaxy (companion)
            e: eccentricity of the orbit
            rmin: distance of closest approach
            R0: initial separation
        """
        m1, m2 = np.sum(galaxy1.mass), np.sum(galaxy2.mass)
        # Relevant formulae:
        # $r_0 = r(1+e)\cos(\phi)$, where $r_0$ ($\neq R_0$) is the semi-latus rectum
        # $r_0 = r_{\min}(1+e)$
        # $v^2 = GM(2/r - 1/a)$, where a is the semimajor axis

        # Solve the reduced two-body problem with reduced mass $\mu$ (mu)
        M = m1 + m2
        r0 = rmin * (1 + e)
```

```
        try:
            phi = np.arccos((r0/R0 - 1) / e) # inverting the orbit equation
            phi = -np.abs(phi) # Choose negative (incoming) angle
            ainv = (1 - e) / rmin # ainv = 1/a, as a may be infinite
            v = np.sqrt(M * (2/R0 - ainv))
            # Finally, calculate the angle of motion. angle = tan(dy/dx)
            # dy/dx = ((dr/dφ)sin(φ) + r cos(φ))/((dr/dφ)cos(φ) − r sin(φ))
            dy = R0/r0 * e * np.sin(phi)**2 + np.cos(phi)
            dx = R0/r0 * e * np.sin(phi) * np.cos(phi) - np.sin(phi)
            vangle = np.arctan2(dy, dx)
        except: raise Exception('''The orbital parameters cannot be satisfied.
            For elliptical orbits check that R0 is posible (<rmax).''')

        # We now need the actual motion of each body
        R_vec = np.array([[R0*np.cos(phi), R0*np.sin(phi), 0.]])
        V_vec = np.array([[v*np.cos(vangle), v*np.sin(vangle), 0.]])

        galaxy1.r_vec += m2/M * R_vec
        galaxy1.v_vec += m2/M * V_vec
        galaxy2.r_vec += -m1/M * R_vec
        galaxy2.v_vec += -m1/M * V_vec

        # Explicitely add the galaxies to the simulation
        self.add(galaxy1)
        self.add(galaxy2)

        if self.verbose: print('Galaxies set in orbit.')


##############################################################################
##############################################################################
class Galaxy():
    """"Helper class for creating galaxies.

    Attributes:
        r_vec (array): position of the particles in the current timestep.
            Shape: (number of particles, 3)
        v_vec (array): velocity in the current timestep.
            Shape: (number of particles, 3)
        a_vec (array): acceleration in the current timestep.
            Shape: (number of particles, 3)
        mass (array): mass of each particle in the simulation.
            Shape: (number of particles,)
        type (array): non-unique identifier for each particle.
            Shape: (number of particles,)    """
    def __init__(self, orientation, centralMass, bulge, disk, halo, sim):
        """Constructor for the Galaxy class.

           Parameters:
                orientation (tupple): (inclination i, argument of pericenter w)
                centralMass (float): mass at the center of the galaxy
                bulge (dict): passed to the addBulge method.
                disk (dict): passed to the addDisk method.
                halo (dict): passed to the addHalo method.
                sim (Simulation): simulation object
        """
        if sim.verbose: print('Initializing galaxy')
        # Build the central mass
        self.r_vec = np.zeros((1, 3))
        self.v_vec = np.zeros((1, 3))
        self.a_vec = np.zeros((1, 3))
        self.mass = np.array([centralMass])
        self.type = np.array([['center', 0]])
        # Build the other components
        self.addBulge(**bulge)
        if sim.verbose: print('Bulge created.')
        self.addDisk(**disk)
        if sim.verbose: print('Disk created.')
        self.addHalo(**halo)
        if sim.verbose: print('Halo created.')
        # Correct particles velocities to generate circular orbits
        # a_centripetal = v^2/r
        a_vec = sim.acceleration(self.r_vec, self.mass, soft=sim.soft)
        a = np.linalg.norm(a_vec, axis=-1, keepdims=True)
        r = np.linalg.norm(self.r_vec, axis=-1, keepdims=True)
        v = np.linalg.norm(self.v_vec[1:], axis=-1, keepdims=True)
        direction_unit = self.v_vec[1:]/v
        # Set orbital velocities (except for central mass)
        self.v_vec[1:] = np.sqrt(a[1:] * r[1:]) * direction_unit
        self.a_vec = np.zeros_like(self.r_vec)
        # Rotate the galaxy into its correct orientation
        self.rotate(*(np.array(orientation)/360 * 2*np.pi))
        if sim.verbose: print('Galaxy set in rotation and oriented.')

    def addBulge(self, model, totalMass, particles, l):
        """Adds a bulge to the galaxy.

            Parameters:
                model (string): parametrization of the bulge.
                    'plummer' and 'hernquist' are supported.
                totalMass (float): total mass of the bulge
                particles (int): number of particles in the bulge
                l (float): characteristic length scale of the model.
        """
        if particles == 0: return None
        # Divide the mass equally among all particles
```

```
        mass = np.ones(particles) * totalMass/particles
        self.mass = np.concatenate([self.mass, mass], axis=0)
        # Create particles according to the radial distribution from model
        if model == 'plummer':
            r = PLUMMER.ppf(np.random.rand(particles), scale=l)
        elif model == 'hernquist':
            r = HERNQUIST.ppf(np.random.rand(particles), scale=l)
        else: raise Exception("""Bulge distribution not allowed.
                    'plummer' and 'hernquist' are the supported values""")
        r_vec = r[:,np.newaxis] * random_unit_vectors(size=particles)
        self.r_vec = np.concatenate([self.r_vec, r_vec], axis=0)
        # Set them orbitting along random directions normal to r_vec
        v_vec = np.cross(r_vec, random_unit_vectors(size=particles))
        self.v_vec = np.concatenate([self.v_vec, v_vec], axis=0)
        # Label the particles
        type_ = [['bulge', 0]]*particles
        self.type = np.concatenate([self.type, type_], axis=0)

    def addDisk(self, model, totalMass, particles, l):
        """Adds a disk to the galaxy.

            Parameters:
                model (string): parametrization of the disk.
                    'rings', 'uniform' and 'exp' are supported.
                totalMass (float): total mass of the bulge
                particles (int): number of particles in the bulge
                l: fot 'exp' and 'uniform' characteristic length of the
                    model. For 'rings' tupple of the form (inner radius,
                    outer radius, number of rings)
        """
        if particles == 0: return None
        # Create particles according to the radial distribution from model
        if model == 'uniform':
            r = UNIFORM.ppf(np.random.rand(particles), scale=l)
            type_ = [['disk', 0]]*particles
            r_vec = r[:,np.newaxis] * random_unit_vectors(particles, '2D')
            self.type = np.concatenate([self.type, type_], axis=0)
        elif model == 'rings':
            # l = [inner radius, outter radius, number of rings]
            distances = np.linspace(*l)
            # Aim for roughly constant areal density
            # Cascade rounding preserves the total number of particles
            perRing = cascade_round(particles * distances / np.sum(distances))
            particles = int(np.sum(perRing)) # prevents numerical errors
            r_vec = np.empty((0, 3))
            for d, n, i in zip(distances, perRing, range(l[2])):
                type_ = [['disk', i+1]]*int(n)
                self.type = np.concatenate([self.type, type_], axis=0)
                phi = np.linspace(0, 2 * np.pi, n, endpoint=False)
                ringr = d * np.array([[np.cos(i), np.sin(i), 0] for i in phi])
                r_vec = np.concatenate([r_vec, ringr], axis=0)
        elif model == 'exp':
            r = EXP.ppf(np.random.rand(particles), scale=l)
            r_vec = r[:,np.newaxis] * random_unit_vectors(particles, '2D')
            type_ = [['disk', 0]]*particles
            self.type = np.concatenate([self.type, type_], axis=0)
        else:
            raise Exception("""Disk distribution not allowed.
                    'uniform', 'rings' and 'exp' are the supported values""")
        self.r_vec = np.concatenate([self.r_vec, r_vec], axis=0)
        # Divide the mass equally among all particles
        mass = np.ones(particles) * totalMass/particles
        self.mass = np.concatenate([self.mass, mass], axis=0)
        # Set them orbitting along the spin plane
        v_vec = np.cross(r_vec, [0, 0, 1])
        self.v_vec = np.concatenate([self.v_vec, v_vec], axis=0)

    def addHalo(self, model, totalMass, particles, rs):
        """Adds a halo to the galaxy.

            Parameters:
                model (string): parametrization of the halo.
                    Only 'NFW' is supported.
                totalMass (float): total mass of the halo
                particles (int): number of particles in the halo
                rs (float): characteristic length scale of the NFW profile.
        """
        if particles == 0: return None
        # Divide the mass equally among all particles
        mass = np.ones(particles)*totalMass/particles
        self.mass = np.concatenate([self.mass, mass], axis=0)
        # Create particles according to the radial distribution from model
        if model == 'NFW':
            r = NFW.ppf(np.random.rand(particles), scale=rs)
        else: raise Exception("""Bulge distribution not allowed.
                    'plummer' and 'hernquist' are the supported values""")
        r_vec = r[:,np.newaxis] * random_unit_vectors(size=particles)
        self.r_vec = np.concatenate([self.r_vec, r_vec], axis=0)
        # Orbit along random directions normal to the radial vector
        v_vec = np.cross(r_vec, random_unit_vectors(size=particles))
        self.v_vec = np.concatenate([self.v_vec, v_vec], axis=0)
        # Label the particles
        type_ = [['dark'], 0]*particles
        self.type = np.concatenate([self.type, type_], axis=0)

    def rotate(self, theta, phi):
```

```python
        """Rotates the galaxy so that its spin is along the (theta, phi)
           direction.

        Parameters:
            theta (float): polar angle.
            phi (float): azimuthal angle.
        """
        M1 = np.array([[1, 0, 0],
                       [0, np.cos(theta), np.sin(theta)],
                       [0, -np.sin(theta), np.cos(theta)]])
        M2 = np.array([[np.cos(phi), np.sin(phi), 0],
                       [-np.sin(phi), np.cos(phi), 0],
                       [0, 0, 1]])
        M = np.matmul(M1, M2) # combine rotations
        self.r_vec = np.tensordot(self.r_vec, M, axes=[1, 0])
        self.v_vec = np.tensordot(self.v_vec, M, axes=[1, 0])
```

Shared C++ *library*

acclib.cpp

```cpp
#include <vector>
#include <iostream>
#include <math.h>
#include <chrono>

using namespace std;
using namespace std::chrono;
#include "Node.h"


/*
Calculates the self gravitational acceleration for a set of particles located
at r_vec (n, 3) with masses m_vec (n,) using a Barnes-Hut tree with theta = 0.7

Parameters:
    r_vec: the position of the particles.
    m_vec: the masses of the particles.
    n: the number of particles. This cannot be infered by C++ and must be
        passed directly.
    size: size of the top node in the octtree.
    px, py, pz: coordinates of the lowest corner of the top node.
    size: characteristic softening scale.

Returns:
    The accelerations computed for each mass (n, 3).
*/
extern "C" double* barnesHutCPP(double** r_vec, double* m_vec, int n,
    double size, double px, double py, double pz, double soft){
    // Create nodes
    std::vector<Node*> nodes;
    for (int i = 0; i < n; i++){
        nodes.push_back(new Node(r_vec[i], m_vec[i]));
    }

    // Create the tree
    Node* tree = new Node(nodes, size, px, py, pz);

    // Calculate the accelerations for each node. We want to return the
    // result as an array and use a 1D array for simplicity since this will
    // be allocated continously in the heap and can be reshaped in Numpy.
    double* accel = new double[3*n];
    for (int i = 0; i < nodes.size(); i++){
        nodes[i]->treeWalk(*tree, 0.7, soft); // thetamax = 0.7
        accel[3*i+0] = nodes[i]->g[0];
        accel[3*i+1] = nodes[i]->g[1];
        accel[3*i+2] = nodes[i]->g[2];
    }

    // Return as an (n,) array
    return accel;
}

/*
Calculates the self gravitational acceleration for a set of particles located
at r_vec (n, 3) with masses m_vec (n,) using Brute Force pairwise summation.
Massive particles must appear first.

Parameters:
    r_vec: the position of the particles.
    m_vec: the masses of the particles.
    n: the number of particles. This cannot be infered by C++ and must be
        passed directly.
    size: characteristic softening scale.

Returns:
    The accelerations computed for each mass (n, 3).
*/
extern "C" double* bruteForceCPP(double** r_vec, double* m_vec,
    int n, double soft){

    // Initialize result and fill with 0s
    // Use a 1D array so as not to have to convert back in Numpy
    double* accel = new double[3*n];
    for (int i=0; i<3*n; i++){
        accel[i] = 0;
```

```
    }

    // Compute the acceleration
    for (int i=0; i<n; i++){
        // Only consider pairs with at least one massive particle i
        if (m_vec[i] == 0.) break;
        for (int j=i+1; j<n; j++){
            // Distance between particles
            double delta[3];
            for (int k = 0; k < 3; k++) delta[k] = r_vec[j][k] - r_vec[i][k];
            double r = sqrt(delta[0]*delta[0]
                + delta[1]*delta[1]
                + delta[2]*delta[2]);
            double tripler = (r+soft) * (r+soft) * r;

            // Compute acceleration on first particle
            double mr3inv = m_vec[i]/tripler;
            for (int k = 0; k < 3; k++) accel[3*j+k] -= mr3inv * delta[k];

            // Compute acceleration on second particle
            // For pairs with one massless particle, no reaction force
            if (m_vec[j] == 0.) break;
            // Otherwise, opposite direction (+)
            mr3inv = m_vec[j]/tripler;
            for (int k = 0; k < 3; k++) accel[3*i+k] += mr3inv * delta[k];
        }
    }

    // Return as an (n,) array
    return accel;
}
```

Node.h

```
#include <vector>

/*
Node class for the Barnes-Hut tree. The choice of pointers as opposed
to references is driven by the necessity to interact with Numpy arrays
using ctypes.
*/
class Node{
private:
    double COM[3]; // Center of mass
    double m; // Mass of the node
    double size; // Size of box, equal for all dimensions
    std::vector<Node*> children;

public:
    double g[3]; // Gravitational acceleration on the node

    // Constructors
    Node(const std::vector<Node*> &pBodies, const double pSize,
        const double px, const double py, const double pz);
    Node(const double* pr_vec, const double pm);

    // Methods
    void treeWalk(const Node &node, const double thetamax, const double soft);
};
```

Node.cpp

```
#include "Node.h"
#include <vector>
#include <iostream>
#include <math.h>
using namespace std;

/*
Node constructor. Used recursively to build the Barnes-Hut tree. px, py, pz
denote the corner (lowest value for each dimension) of the box of size pSize.
pBodies is a vector containing all the nodes that must be placed in this box.
*/
Node::Node(const vector<Node*> &pBodies, const double pSize,
    const double px, const double py, const double pz){
    size = pSize; // Required later for treeWalk

    // Divide into subnodes (octants)
    vector<Node*> subBodies[2][2][2];

    for (int i = 0; i < pBodies.size(); i++){
        int xIndex, yIndex, zIndex;

        if (pBodies[i]->COM[0] < (px + (size / 2))) xIndex = 0;
        else xIndex = 1;

        if (pBodies[i]->COM[1] < (py + (size / 2))) yIndex = 0;
        else yIndex = 1;

        if (pBodies[i]->COM[2] < (pz + (size / 2))) zIndex = 0;
        else zIndex = 1;

        subBodies[xIndex][yIndex][zIndex].push_back(pBodies[i]);
    }


    // Recursively place the nodes
    // g++ will unroll these loops
```

```
    for (int i = 0; i < 2; i++){
        for (int j = 0; j < 2; j++){
            for (int k = 0; k < 2; k++){
                switch(subBodies[i][j][k].size()){
                    case 0: continue;
                    case 1:
                        subBodies[i][j][k][0]->size = size/2;
                        children.push_back(subBodies[i][j][k][0]);
                        break;
                    default:
                        children.push_back(new Node(subBodies[i][j][k], size/2,
                            px + size/2*i, py + size/2*j, pz + size/2*k));
                }
            }
        }
    }

    // Recursively calculate the COM
    memset(COM, 0, sizeof(COM)); // Set COM to 0s
    m = 0.; // mass
    for (int i = 0; i < children.size(); i++){
        m += children[i]->m;
        for (int j = 0; j < 3; j++)
            COM[j] += children[i]->m * children[i]->COM[j];
    }
    // COM only relevant if there is mass in the octant
    if (m > 0) for (int i = 0; i < 3; i++) COM[i] /= m;

}

/*
Node constructor. Used to build the leaf nodes directly from the data passed
from Python using ctypes.
*/
Node::Node(const double* pr_vec, const double pm){
    // Initialize a node (leaf)
    for (int i = 0; i < 3; i++) COM[i] = pr_vec[i];
    memset(g, 0, sizeof(g)); // Set g to 0s
    m = pm; // mass
}

/*
Calculate the acceleration at the this node. Used recursively calling
treeWalk(topNode, thetamax). This is O(log n) and will be called for
each node in the tree: O(n log n). Soft defines the characteristic
plummer softening scale.
*/
void Node::treeWalk(const Node &node,
    const double thetamax, const double soft){
    // Calculate distance to node
    double delta[3];
    for (int i = 0; i < 3; i++) delta[i] = node.COM[i] - COM[i];
    double r = sqrt(delta[0]*delta[0] + delta[1]*delta[1] + delta[2]*delta[2]);

    if (r==0) return; // Do not interact with self

    // If it satisfies the size/r < thetamax criterion, add the g contribution
    if (node.children.size() == 0 || node.size/r < thetamax){
        double tripler = (r+soft) * (r+soft) * r;
        for(int i = 0; i < 3; i++) g[i] += node.m * delta[i] / tripler;
    }
    else{ // Otherwise recurse into its children
        for (int i = 0; i < node.children.size(); i++){
            treeWalk((*node.children[i]), thetamax, soft);
        }
    }
}
```

Other

analysis/segmentation.py

```python
"""Segmentation algorithm used to identify the different structures
that are formed in the encounter. This file can be called from the
command line to make an illustrative plot of the algorithm.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

import utils


def segmentEncounter(data, plot=False, mode='all'):
    """Segment the encounter into tail, bridge, orbitting and
    stolen particles as described in the report.

    Parameters:
        data: A data instance as saved by the simulation to a pickle file
        plot: If true the segmentation will be plotted and shown. Highly
            useful for debugging.
        mode (string): If mode is 'all' all parts of the encounter will be
            identified. If mode is 'bridge' only the bridge will be
            identified. This is useful when there may be no tail.

```

```python
    Returns:
        masks (tupple): tupple of array corresponding to the masks of the
            (bridge, stolen, orbitting, tail) particles. One can then use
            e.g. data['r_vec'][bridgeMask].
        shape (tupple): tupple of (distances, angles) as measured from the
            center of mass and with respect to the x axis. They define the
            shape of the tail
        length (float): total length of the tail.
    """
    nRings = 100 # number of rings to use when segmenting the data

    # Localize the central masses
    r_vec = data['r_vec']
    centers = r_vec[data['type'][:,0]=='center']
    rCenters_vec = centers[1] - centers[0]
    rCenters = np.linalg.norm(rCenters_vec)
    rCenters_unit = rCenters_vec/np.linalg.norm(rCenters_vec)
    # Take particles to be on the tail a priori and
    # remove them as they are found in other structures
    particlesLeft = np.arange(0, len(r_vec))


    if plot:
        colour = '#c40f4c'
        f, axs = plt.subplots(1, 3,  figsize=(9, 4), sharey=False)
        f.subplots_adjust(hspace=0, wspace=0)
        axs[0].scatter(r_vec[:,0], r_vec[:,1], c=colour, alpha=0.1, s=0.1)
        axs[0].axis('equal')
        utils.plotCenterMasses(axs[0], data)
        axs[0].axis('off')

    # Step 1: project points to see if they are part of the bridge
    parallelProjection = np.dot(r_vec - centers[0], rCenters_unit)
    perpendicularProjection = np.linalg.norm(r_vec - centers[0][np.newaxis]
        - parallelProjection[:,np.newaxis] * rCenters_unit[np.newaxis], axis=-1)
    bridgeMask = np.logical_and(np.logical_and(0.3*rCenters < parallelProjection,
        parallelProjection < .7*rCenters), perpendicularProjection < 2)

    # Remove the bridge
    notInBridge = np.logical_not(bridgeMask)
    r_vec = r_vec[notInBridge]
    particlesLeft = particlesLeft[notInBridge]

    if mode == 'bridge':
        return (bridgeMask, None, None, None), None, None

    # Step 2: select stolen particles by checking distance to centers
    stolenMask = (np.linalg.norm(r_vec - centers[0][np.newaxis], axis=-1)
        > np.linalg.norm(r_vec - centers[1][np.newaxis], axis=-1))
    # Remove the stolen part
    notStolen = np.logical_not(stolenMask)
    r_vec = r_vec[notStolen]
    particlesLeft, stolenMask = particlesLeft[notStolen], particlesLeft[stolenMask]

    # Step 3: segment data into concentric rings (spherical shells really)
    r_vec = r_vec - centers[0]
    r = np.linalg.norm(r_vec, axis=-1)
    edges = np.linspace(0, 30, nRings) # nRings concentric spheres
    indices = np.digitize(r, edges) # Classify particles into shells

    if plot:
        axs[1].scatter(r_vec[:,0], r_vec[:,1], c=colour, alpha=.1, s=.1)
        axs[1].axis('equal')
        axs[1].scatter(0, 0, s=100, marker="*", c='black', alpha=.7)
        axs[1].axis('off')

    # Step 4: find start of tail
    start = None
    for i in range(1, nRings+1):
        rMean = np.mean(r[indices==i])
        rMean_vec = np.mean(r_vec[indices==i], axis=0)
        parameter = np.linalg.norm(rMean_vec)/rMean

        if plot:
            circ = patches.Circle((0,0), edges[i-1], linewidth=0.5,edgecolor='black',facecol
            axs[1].add_patch(circ)
            txtxy = edges[i-1] * np.array([np.sin(i/13), np.cos(i/13)])
            axs[1].annotate("{:.2f}".format(parameter), xy=txtxy, backgroundcolor='#ffffff55

        if start is None and parameter>.8 :
            start = i #Here starts the tail
            startDirection = rMean_vec/np.linalg.norm(rMean_vec)
            if not plot: break;

    if start is None: #abort if nothing found
        raise Exception('Could not identify tail')

    # Step 5: remove all circles before start
    inInnerRings = indices < start
    # Remove particles on the opposite direction to startDirection.
    # in the now innermost 5 rings. Likely traces of the bridge.
    oppositeDirection = np.dot(r_vec, startDirection) < 0
    in5InnermostRings = indices <= start+5
    orbitting = np.logical_or(inInnerRings,
        np.logical_and(oppositeDirection, in5InnermostRings))
    orbittingMask = particlesLeft[orbitting]
```

```python
    r_vec = r_vec[np.logical_not(orbitting)]
    tailMask = particlesLeft[np.logical_not(orbitting)]

    if plot:
        axs[2].scatter(r_vec[:,0], r_vec[:,1], c=colour, alpha=0.1, s=0.1)
        axs[2].axis('equal')
        axs[2].scatter(0, 0, s=100, marker="*", c='black', alpha=.7)
        axs[2].axis('off')

    # Step 6: measure tail length and shape
    r = np.linalg.norm(r_vec, axis=-1)
    indices = np.digitize(r, edges)
    # Make list of barycenters
    points = [list(np.mean(r_vec[indices==i], axis=0))
        for i in range(1, nRings) if len(r_vec[indices==i])!=0]
    points = np.array(points)
    # Calculate total length
    lengths = np.sqrt(np.sum(np.diff(points, axis=0)**2, axis=1))
    length = np.sum(lengths)
    # Shape (for 2D only)
    angles = np.arctan2(points[:,1], points[:,0])
    distances = np.linalg.norm(points, axis=-1)
    shape = (distances, angles)

    if plot:
        axs[2].plot(points[:,0], points[:,1], c='black', linewidth=0.5, marker='+')

    if plot:
        plt.show()

    return (bridgeMask, stolenMask, orbittingMask, tailMask), shape, length


if __name__ == "__main__":
    data = utils.loadData('200mass', 10400)
    segmentEncounter(data, plot=True)
```

run_simulated_annealing.py

```python
"""Simulated annealing algorithm used to match the simulation of the
Antennae to the observations by comparing binarized images."""

import numpy as np
import scipy
import pickle
from scipy import ndimage
from fast_histogram import histogram2d
from scipy.signal import convolve2d
from numba import jit
from matplotlib.image import imread
import datetime

from simulation import Simulation, Galaxy

T = .25 # Initial tempreature
STEPS = 1500
DECAY = .998 # Exponential decay factor

def simToBitmap(sim, theta, phi, view, scale, x, y, galaxy):
    """Obtain a bitmap of one galaxy as viewed from a given direction.
       The binning has been chosen so that the scale and the offset (x,y)
       are expected to be approximately 1 and (0, 0) respectively.

        Parameters:
            sim (Simulation): Simulation to project.
            theta (float): polar angle of viewing direction.
            phi (float): azimuthal angle of viewing direction.
            view (float): rotation angle along viewing direction.
            scale (float): scaling factor.
            x (float): x offset
            y (float): y offset
            galaxy (int): galaxy to plot. Either 0, or 1. They are assumed
                to have the same number of particles.
    """
    # Obtain components in new x',y' plane
    r_vec = (sim.project(theta, phi, view) - [[x+.12,y+1.3]]) * scale
    # Select a single galaxy. We match them separately in the algorithm.
    if galaxy==0: r_vec = r_vec[2:len(r_vec)//2-1] #omit central masses
    if galaxy==1: r_vec = r_vec[len(r_vec)//2-1:]
    # Use a fast histogram, much faster than numpy ()
    H = histogram2d(r_vec[:,0], r_vec[:,1],
        range=[[-5,5], [-5,5]], bins=(50, 50))
    im = np.zeros((50, 50))
    H = convolve2d(H, np.ones((2,2)), mode='same') # Smooth the image
    im[H>=1] = 1 # Binary the image

    return im

@jit(nopython=True) # Numba annotation
def bitmapScoreAlgo(im1, im2):
    """Computes the f1 score betwen two binarized images

    Parameters
        im1 (arr): nxm ground truth image
        im2 (arr): nxm candidate image

    Returns
```

```python
        f1 score
    """
    TP = np.sum((im1==1.0) & (im2==1.0))
    TN = np.sum((im1==0.0) & (im2==0.0))
    FP = np.sum((im1==0.0) & (im2==1.0))
    FN = np.sum((im1==1.0) & (im2==0.0))
    if TP==0: return 0
    precision = TP/(TP+FP)
    recall = TP/(TP+FN)
    return 2*precision*recall / (precision + recall)

# The matching algorithm attempts to improve the match by shifting the
# image by one pixel in each direction. If none improve the score the
# f1-score of said local maximum is retuned. To make this highly efficient
# (as this is run millions of time) we explicitely write functions to
# shift an image by 1 pixel in each direction, as these can then be compiled
# using Numba (jit annotation) to low-level code.
# Performance is crucial here and must sadly be prioritized over conciseness
@jit(nopython=True)
def shiftBottom(im, im2):
    """Shifts an image by one pixel downwards.

    Parameters:
        im (arr): the nxm image to shift by one pixel.
        im2 (arr): an nxm array where the new image will be placed.

    Returns:
        A reference to im2
    """
    im2[1:] = im[:-1]
    im2[0] = 0
    return im2

@jit(nopython=True)
def shiftTop(im, im2):
    """Shifts an image by one pixel upwards."""
    im2[:-1] = im[1:]
    im2[-1] = 0
    return im2

@jit(nopython=True)
def shiftLeft(im, im2):
    """Shifts an image by one pixel to the left."""
    im2[:,1:] = im[:,:-1]
    im2[:,0] = 0
    return im2

@jit(nopython=True)
def shiftRight(im, im2):
    """Shifts an image by one pixel to the right."""
    im2[:,:-1] = im[:,1:]
    im2[:,-1] = 0
    return im2

@jit
def bitmapScore(im1, im2, _prev=None, _bestscore=None, _zeros=None):
    """Computes the bitmap score between two images. This is the f1-score
       but we allow the algorithm to attempt to improve the score by
       shifting the images. The algorithm is implemented recursively.

    Parameters:
        im1 (array): The ground truth nxm image.
        im2 (array): The candidate nxm imgae.
        _prev: Used internally for recursion.
        _bestscore: Used internally for recursion.
        _zeros: Used internally for recursion.

    Returns:
        f1-score for the two images.
    """
    # When the function is called externally, initialize an array of zeros
    # and compute the score for no shifting. The zeros array is used for
    # performance to only create a new array once.
    if _bestscore is None:
        _bestscore = bitmapScoreAlgo(im1, im2)
        _zeros = np.zeros_like(im2)
    # Attempt to improve the score by shifting the image in a direction
    # Keeping track of _prev allows to not 'go back' and undo and attempt
    # to undo a shift needlessly.
    if _prev is not 0: # try left
        shifted = shiftLeft(im2, _zeros)
        score = bitmapScoreAlgo(im1, shifted)
        if score > _bestscore: return bitmapScore(im1, shifted,
            _prev=1, _bestscore=score, _zeros=_zeros)
    if _prev is not 1: # try right
        shifted = shiftRight(im2, _zeros)
        score = bitmapScoreAlgo(im1, shifted)
        if score > _bestscore: return bitmapScore(im1, shifted,
            _prev=0, _bestscore=score, _zeros=_zeros)
    if _prev is not 2: # try top
        shifted = shiftTop(im2, _zeros)
        score = bitmapScoreAlgo(im1, shifted)
        if score > _bestscore: return bitmapScore(im1, shifted,
            _prev=3, _bestscore=score, _zeros=_zeros)
    if _prev is not 3: # try bottom
        shifted = shiftBottom(im2, _zeros)
```

```python
        score = bitmapScoreAlgo(im1, shifted)
        if score > _bestscore: return bitmapScore(im1, shifted,
            _prev=2, _bestscore=score, _zeros=_zeros)
    # Return _bestscore if shifting did not improve (local maximum).
    return _bestscore


def attemptSimulation(theta1, phi1, theta2, phi2, rmin=1, e=.5, R=2,
        disk1=.75, disk2=.65, mu=1, plot=False, steps=2000):
    """Runs a simulation with the given parameters and compares it
    to observations of the antennae to return a score out of 2.

    Parameters:
        theta1 (float): polar angle for the spin of the first galaxy.
        phi1 (float): azimuthal angle for the spin of the first galaxy.
        theta2 (float): polar angle for the spin of the second galaxy.
        phi2 (float): azimuthal angle for the spin of the second galaxy.
        rmin (float): closest distance of approach of orbit.
        e (float): eccentricity of the orbit.
        R (float): initial separation
        disk1 (float): radius of the uniform disk of the first galaxy
        disk2 (float): radius of the uniform disk of the first galaxy
        mu (float): ratio of masses of the two galaxies
        plot (float): if true the simulation will be saved to data/annealing/
        steps (float): number of times the f1 score will be computed, along
            random viewing directions per 100 timesteps.

    Returns:
        f1-score: score obtained by the simulation.
    """

    # Initialize the simulation
    sim = Simulation(dt=1E-3, soft=0.1, verbose=False, CONFIG=None, method='bruteFc
    galaxy1 = Galaxy(orientation = (theta1, phi1), centralMass=2/(1+mu),
        sim=sim, bulge={'model':'plummer', 'particles':0, 'totalMass':0, 'l':0},
        disk={'model':'uniform', 'particles':2000, 'l':disk1, 'totalMass':0},
        halo={'model':'NFW', 'particles':0, 'rs':1, 'totalMass':0})
    galaxy2 = Galaxy(orientation = (theta2, phi2), centralMass=2*mu/(1+mu),
        sim=sim, bulge={'model':'plummer', 'particles':0, 'totalMass':0, 'l':0},
        disk={'model':'uniform', 'particles':2000, 'l':disk2, 'totalMass':0},
        halo={'model':'NFW', 'particles':0, 'rs':1, 'totalMass':0})
    sim.setOrbit(galaxy1, galaxy2, e=e, R0=R, rmin=rmin) # define the orbit

    # Run the simulation manually
    # Initialize the parameters consistently with the velocities
    sim.rprev_vec = sim.r_vec - sim.v_vec * sim.dt
    # Keep track of the best score
    bestScore = 0
    # and its corresponding binarized image and parameters
    bestImage, bestParams = 0, 0
    hasReachedPericenter = False

    # Run until tmax = 25
    for i in range(25001):
        sim.step()
        if i%100==0: # Every Δt = 0.1
            # Check if we are close to pericenter
            centers = sim.r_vec[sim.type[:,0] == 'center']
            if np.linalg.norm(centers[0] - centers[1]) < 1.3*rmin:
                hasReachedPericenter = True
            # Do not evaluate the f1-score until pericenter.
            if not hasReachedPericenter: continue

            # Check multiple (steps) viewing directions at random
            localBestScore = 0
            localBestImage, localBestParams = 0, 0
            for j in range(steps):
                # The viewing directions are isotropically distributed
                theta = np.arccos(np.random.uniform(-1, 1))
                phi = np.random.uniform(0, 2*np.pi)
                # Rotation along line of sight
                view = np.random.uniform(0, 2*np.pi)
                scale = np.random.uniform(0.6, 1.0)
                x = np.random.uniform(-1.0, 1.0) # Offsets
                y = np.random.uniform(-1.0, 1.0)
                # Get images for each galaxy and compute their score separately
                im1 = simToBitmap(sim, theta, phi, view, scale, x, y, galaxy=0)
                im2 = simToBitmap(sim, theta, phi, view, scale, x, y, galaxy=1)
                score = bitmapScore(GT1, im1) + bitmapScore(GT2, im2)
                if score > localBestScore:
                    localBestScore = score
                    localBestImage = [im1,im2]
                    localBestParams = [i, theta, phi, view, scale, x, y]

            if bestScore < localBestScore:
                bestScore = localBestScore
                bestImage = localBestImage
                bestParams = localBestParams
            if plot:
                sim.save('annealing', type='2D')

    print('Best score for this attempt', bestScore)
    print('using viewing parameters', bestParams)

    return bestScore
```

```python

################################################################################
################################################################################

# Generate a (50, 50) bitmap for each galaxy
# They are stored globally in GT1 and GT2 (Ground Truth)
im = imread('literature/figs/g1c.tif')
im = np.mean(im, axis=-1)
im = scipy.misc.imresize(im, (50,50))
GT1 = np.zeros((50, 50))
GT1[im > 50] = 1

im = imread('literature/figs/g2c.tif')
im = np.mean(im, axis=-1)
im = scipy.misc.imresize(im, (50,50))
GT2 = np.zeros((50, 50))

# Define the limits and relate scale of the variations in each parameter
# In the same order as attemptSimulation
# phi1, theta1, phi2, theta2, rmin (fixed), e, R (fixed), disk1, disk2
LIMITS = np.array([[np.pi, 2 * np.pi], [-np.pi, np.pi],
                   [0, np.pi], [-np.pi, np.pi],
                   [1,1], [.5,1.0], [2.2,2.2], [.5,.8], [.5,.8]])
VARIATIONS = np.array([.08, .15, .08, .15, 0, .01, 0, .01, .01])

# Choose a random starting point and evaluate it
bestparams = [np.random.uniform(*l) for l in LIMITS]
log = []
bestscore = attemptSimulation(*bestparams, steps=500)
print('Starting with score', bestscore, 'with parameters', bestparams)
log.append([bestscore, bestparams, True])

for i in range(STEPS):
    T = T * DECAY #exponential decay
    # Perturb the parameters
    params = bestparams + np.random.normal(scale=VARIATIONS) * 2 * T / 0.04
    # Allow the angles from −π to π to wrap around
    for j in [1,3]:
        params[j] = np.mod(params[j] - LIMITS[j][0], LIMITS[j][1] - LIMITS[j][0])
        params[j] += LIMITS[j][0]
    # Clip parameters outside their allowed range
    params = np.clip(params, LIMITS[:, 0], LIMITS[:, 1])
    # Evaluate the score for this attempt, use more steps as i progresses
    # so as to reduce the noise in the evaluation of the score
    score = attemptSimulation(*params, steps=500 + i)
    # Perform simulated annealing with a typical exponential rule
    if score > bestscore or np.exp(-(bestscore-score)/T) > np.random.rand():
        # Flip to this new point
        print('NEW BEST ____', i, T, score, params)
        bestscore = score
        bestparams = params
        log.append([score, params, True])
    else: # Remain in the old point
        log.append([score, params, False])
    # Save the progress for future plotting
    pickle.dump(log, open('data/logs/logb.pickle', "wb" ) )
```

References

[1] F. Zwicky, *Luminous and dark formations of intergalactic matter*, *Physics Today* **6** (1953) 7–11.

[2] J. C. Mihos and L. Hernquist, *Gasdynamics and Starbursts in Major Mergers*, **464** (June, 1996) 641, [astro-ph/9512099].

[3] K. M. Dasyra et al., *Dynamical properties of ultraluminous infrared galaxies. I. mass ratio conditions for ulirg activity in interacting pairs*, *Astrophys. J.* **638** (2006) 745–758, [astro-ph/0510670].

[4] A. Toomre et al., *Evolution of galaxies and stellar populations*, in *Proceedings of a Conference at Yale University, May*, pp. 19–21, 1977.

[5] T. Naab et al., *Minor Mergers and the Size Evolution of Elliptical Galaxies*, **699** (July, 2009) L178–L182, [arXiv:0903.1636].

[6] A. Toomre and J. Toomre, *Galactic bridges and tails*, *The Astrophysical Journal* **178** (1972) 623–666.

[7] J. E. Barnes, *Encounters of disk/halo galaxies*, **331** (Aug., 1988) 699–717.

[8] J. C. Mihos et al., *Modeling the Spatial Distribution of Star Formation in Interacting Disk Galaxies*, **418** (Nov., 1993) 82.

[9] J. Barnes and P. Hut, *A hierarchical o (n log n) force-calculation algorithm*, *nature* **324** (1986), no. 6096 446.

[10] L. B. Lucy, *A numerical approach to the testing of the fission hypothesis*, *The astronomical journal* **82** (1977) 1013–1024.

[11] R. A. Gingold and J. J. Monaghan, *Smoothed particle hydrodynamics: theory and application to non-spherical stars*, *Monthly notices of the royal astronomical society* **181** (1977), no. 3 375–389.

[12] P.-A. Duc and F. Renaud, *Tides in colliding galaxies*, in *Tides in astronomy and astrophysics*, pp. 327–369. Springer, 2013. astro-ph/1112.1922.

[13] I. Chilingarian et al., *The GalMer database: galaxy mergers in the virtual observatory*, *Astronomy & Astrophysics* **518** (2010) A61, [astro-ph/1003.3243].

[14] S. J. Karl et al, *One moment in time - modeling star formation in the antennae*, *The Astrophysical Journal Letters* **715** (2010), no. 2 L88, [astro-ph/1003.0685].

[15] R. Teyssier et al., *The driving mechanism of starbursts in galaxy mergers*, *The Astrophysical Journal Letters* **720** (2010), no. 2 L149, [astro-ph/1006.4757].

[16] M. Maji et al., *The formation and evolution of star clusters in interacting galaxies*, *The Astrophysical Journal* **844** (2017), no. 2 108, [astro-ph/1606.07091].

[17] P. A. M. Belles, *Formation of stars and star clusters in colliding galaxies*. PhD thesis, 2013.

[18] S. Aarseth and F. Hoyle, *Dynamical evolution of clusters of galaxies, i*, *Monthly Notices of the Royal Astronomical Society* **126** (1963), no. 3 223–255.

[19] C. D. Wilson et al., *High-resolution imaging of molecular gas and dust in the antennae (ngc 4038/39): super giant molecular complexes*, *The Astrophysical Journal* **542** (2000), no. 1 120, [astro-ph/0005208].

[20] J. Hibbard et al., *High-Resolution H I Mapping of NGC 4038/39 ("The Antennae") and Its Tidal Dwarf Galaxy Candidates*, [astro-ph/0110581].

[21] J. E. Barnes et al., *Identikit 1: A modeling tool for interacting disk galaxies*, *The Astronomical Journal* **137** (2009), no. 2 3071, [astro-ph/0811.3039].

[22] J. E. Barnes, *Identikit 2: an algorithm for reconstructing galactic collisions*, *Monthly Notices of the Royal Astronomical Society* **413** (2011), no. 4 2860–2872, [astro-ph/1101.5671].

[23] J. B. Smith et al., *The automatic galaxy collision software*, *arXiv preprint arXiv:0908.3478* (2009) [astro-ph/0908.3478].

[24] S. Karl, *The Antennae Galaxies-a key to galactic evolution*. PhD thesis, 2011.

[25] C. Oh, E. Gavves, and M. Welling, *BOCK : Bayesian Optimization with Cylindrical Kernels*, *arXiv e-prints* (Jun, 2018) arXiv:1806.01619, [arXiv:1806.01619].

[26] S. J. K. et al.", *Towards an accurate model for the Antennae Galaxies*, *Astron. Nachr.* **329** (2008) 1042, [arXiv:0809.5020].

[27] M. Noguchi, *Triggering of repetitive starbursts in merging galaxies*, *Monthly Notices of the Royal Astronomical Society* **251** (1991), no. 2 360–368.

[28] M. Wetzstein, T. Naab, and A. Burkert, *Do dwarf galaxies form in tidal tails?*, *Mon. Not. Roy. Astron. Soc.* **375** (2007) 805–820, [astro-ph/0510821].