Spectacular Videos: Synthetic Universes, with Star Collision Graph

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.0, November 2022

Abstract

The N-body problem consists of predicting the evolution of celestial bodies bound by gravity. Here I go one step further: up to 1000 stars and star clusters are simulated using various initial conditions, to produce videos that show how these synthetic universes evolve. It tells a lot about the past and future of our current universe, corroborating the theory that it is expanding, albeit more and more slowly. In addition, stars with negative masses and gravity laws other than the standard inverse square, when allowed, lead to the most bizarre systems and spectacular videos. Star collisions are studied in details and lead to interesting graph theory applications. I provide the Python code for these simulations, including the production of animated data visualizations (videos) and graph representations.

Contents

1 Introduction

This project started as an attempt to generate simulations for the three-body problem [Wiki] in astronomy: studying the orbits of three celestial bodies subject to their gravitational interactions. There are many illustrations available online, and after some research, I was intrigued by Philip Mocz's version of the N-body problem: the generalization involving an arbitrary number of celestial bodies. These bodies are referred to as stars in this article. Philip is a computational physicist at Lawrence Livermore National Laboratory, with a Ph.D. in astrophysics from Harvard University. The Python code for his simulations can be found here.

My simulations are based on his code, which I have significantly upgraded. The end result is the three-galaxy problem: small star clusters, each with hundreds of stars, coalescing due to gravitational forces of the individual stars. It simulates the merging of galaxies. In addition, I added a birth process, with new stars constantly generated. I also allow for star collisions, resulting in fewer but bigger stars over time. Finally, my simulations allow for stars with negative masses, as well as unusual gravitation laws, different from the classic inverse square law [Wiki].

These bizarre universes lead to spectacular data animations (MP4 videos), but perhaps most importantly, it may help explain what could cause our universe to expand, including the different stages of compression and expansion over time. Depending on the initial configuration, very different outcomes are possible. Negative masses, with cluster centroids based on the absolute value of the mass while gravitational forces are based on the signed mass, could lead to a different model of the universe. Many well-known phenomena, such as rogue stars escaping their cluster at great velocity, black holes and twin stars formation, star filaments, and star clusters

becoming less energetic over time (decreasing expansion, smaller velocities) are striking features visible in my videos. Star collisions lead to an interesting graph problem.

The implementation uses a discrete approximation to Newton's law of gravity. A previous version based on elliptic orbits but not complying with the laws of our universe, can be found in [1]. Other spectacular orbit visualizations, which have been compared to the 3-body problem but are indeed related to the Riemann Hypothesis in number theory, can be found in [4]. For 3D visualizations, see here. The simulations in this article are 2D and correspond to a projection on one of the 2D planes.

2 Model parameters and simulation results

The evolving star systems featured in the videos start with an initial configuration, consisting of the location, velocity and mass of the stars. These vectors are stored in three numpy arrays in the Python code, named `pos`, `vel` and `mass`, with one entry per star. The position and velocity have three components, corresponding to the X, Y and Z axis. Initial values are randomly generated. A typical video requires about a billion pseudo-random numbers. Thus it is important to use a good pseudo-random number generator (PRNG). How to choose a good seed for the Python PRNG – the Mersenne twister – is discussed in chapter 9 in my book "Intuitive Machine Learning and Explainable AI" [3].

The location, velocity and mass of each star is updated at each iteration, based on the current proximity, speed and velocity of the other stars, according to gravitation laws. Negative masses, star collisions, new star generation and star grouping (star clusters) are allowed. In addition to `pos`, `vel` and `mass`, there is one additional array named `col`, that stores the color of each star, when displayed in the video. Individual colors can change over time, as explained in section 2.1.

2.1 Explanation of color codes

Unless the user selects a model with multiple star clusters via the option `threeClusters=True`, a star with positive mass is blue, and one with negative mass is red. If collisions are allowed, a star will turn and stay orange after absorbing another star. The losing star (the one that gets eaten) has its mass set to zero and won't be visible anymore in the video: the size of a star pictured in the video is proportional to its mass at any given time. When a new star is generated after the initialization step, its color is set to dark violet, regardless of the mass sign. Again, it will turn orange if it collides with another star.

If choosing a system with three star clusters, the star color will be blue, green or magenta depending on which clusters it initially belongs to. Negative masses or new star creation are not yet allowed in these systems, mostly to avoid conflicts with the color scheme. Again, upon collision, the star color turns to orange

2.2 Detailed description of top parameters

Now I describe all the parameters and features available in my implementation, in Table 1. I recommend to read this table, as it shows all the options offered in the Python program. Finding a great set of parameters to illustrate a particular type of system, is not easy. In section 2.3, I describe eight hand-picked set of parameter configurations covering a large variety of situations. With a bit of practice and common sense, it becomes easier to predict the behavior of the system based on the parameter selection.

`starBoost`	Create one massive star in the system if value larger than 1 in absolute value. For instance, `starBoost=-5` means that the star in position 0 in the star tables, has a negative mass which is (in absolute value) 5 times larger than any other star.
`law`	The classic law of gravity uses $r/\|\|r\|\|^3$ in its formula, where $\|\|r\|\|$ is the distance between two interacting bodies. This corresponds to `law=3`, but you can try other values for the exponent.
`speed`	Increase or decrease the initial velocities of the stars. It has a big impact on the evolution of the system, with high speeds potentially above escape velocity or preventing cycling orbits from happening. You can choose `speed=0`, meaning that all stars are initially at rest.
`zoom`	Specifies the visualization window. For instance, `zoom=2` means that the portion of the sky displayed in the video is $[-2, 2] \times [-2, 2]$.
`negativeMass`	When set to `True`, the star masses follow some Gaussian rather than exponential distribution upon creation, and some masses will be negative.

collisions	By default, collisions are not allowed. If set to True, stars getting very close to each other are deemed to have collided: one star adsorbs the other one and its mass is updated accordingly; the other star gets its mass adjusted to zero.
collThresh	Determines how close to each other two stars must be to result in a collision. The lower collThresh, the more collisions. The maximum value is collThresh=1, resulting in no collision. Requires collisions=True.
expand	Some configurations result in the star cluster expanding over time, with fewer and fewer stars in the observation window. A value higher than expand=1 offers a zoom-out, with the window of observations becoming larger over time, giving the impression that your observation point is moving away from the star cluster, with stars seemingly shrinking over time, allowing you to see the full cluster at all times despite its expansion. To the contrary, a negative value corresponds to zoom-in.
origin	There is no "center of the universe", that is, there is no absolute origin. Set origin='Centroid' to focus your vision around the moving centroid of the system (it will be static on the video). If you added a massive star with the parameter starBoost, it makes sense to consider this big star as your origin, by setting origin='Star_0'. Another option is origin='Zero'.
threeClusters	The python code simulates one star cluster. You can expand the possibilities with threeClusters=True. Presently, not implemented with negative masses or new star generation.
p, Nstars, N	Initially start with Nstars active stars, out of a potential of N stars. The active stars are the first ones in the star tables. The inactive stars have their mass set to 0. At each new video frame, it turns an inactive star into an active one with probability p, thus increasing the Nstars counter. Great to start with very few stars and see how the system evolves until it has hundreds of stars, some colliding, and some becoming larger and larger after eating smaller ones. Not implemented with negative masses. The reliance on all inter-distances between stars is the bottleneck in the current implementation.
dt, t, tend	The parameter dt represents time increments, with a large dt resulting in a fast-moving video. Initial and end times are respectively t and tend. The number of frames in the resulting video is (tend-t)/dt.
G	The gravitation parameter. The default value is 1. I tested smaller values as well.
softening	Increase distance between two stars, to avoid division by zero if the distance vanishes.
createVideo	May slow down the simulations if createVideo=True. Set it to False when testing parameters, until you are ready to produce the video.
saveData	The output file nbody.txt contains N rows per video frame, each with 13 columns. It can be very large and slow to produce. Set saveData=False if you don't need it. Regardless, the much smaller file nbody_graph.py, summarizing the collisions (if any), is always produced.
fps, my_dpi	Respectively the number of frames per second, and dots per each in the video. A value above 240 for my_dpi produces high resolution, but a bigger file. A value above 20 for fps produces much shorter videos than (say) fps set to 3.
adjustVel	If True, converts velocities to centroid frame. For compatibility with original version.

Table 1: Description of top parameters used in the star cluster simulator

2.3 Interesting parameter sets

In other to show the possibilities of the algorithm, I created a table featuring eight parameter sets covering a wide range of situations. These sets are broken down into the following categories:

- The first two sets feature a universe with a non-standard law of gravity (law=0.5). In addition, negative masses are allowed. The first set has a massive red star with a negative mass (starBoost=-30): the star cluster expands and contracts regularly, with wilder oscillations and sub-clusters forming over time. In the second set, oscillations are even faster, but much more predictable.

- The third set is the standard universe. It starts very much like the second set, but never contracts. Instead, it expands more and more, but the expansion pace and star velocities considerably slow down over time.
- The fourth set, again with a massive red star (negative mass) at the center, has the most spectacular behavior, thanks to `law=-0.5`. Oscillations become incredibly fast over time, with significant expansion and scattered stars rotating wildly around the massive red star, seemingly ending in a singularity.
- The fifth set is similar to the third one (standard universe), but this time star collisions are allowed. Orange stars are those that have collided, with a few of them growing bigger by eating more stars over time. There are many collisions initially, but eventually, due to expansion, collisions become very rare and even absent. There is a visible drift in the video. Occasionally, rogue stars escape at high speed. There are a number of stars that have a companion star for some time until the paths diverge.
- The sixth and seventh sets also correspond to the standard universe with collisions, but it starts with three star clusters that coalesce over time. In the sixth set, I zoom-in (`expand=-0.2`) on a specific location during the course of the video, while on the identical system in the seventh set, I zoom out (`expand=0.2`).
- The seventh set corresponds to a standard universe with collisions allowed, as well as new star creation over time. It is not realistic in the sense that the mass of the whole system is not constant over time. But the addition of new stars prevent the single star cluster from expanding, creating some stability. Also, it starts with just one star and ends with several hundreds: it allows you to see the behavior when the number of stars is smaller, with more regular patterns partly caused by a massive star at the center, this time with positive mass.

At the top of Table 2, the clickable blue links (one per parameter set) point to the corresponding videos on YouTube, featuring the evolution of the star systems in question. It is surprising to see that sometimes, even some of the biggest stars can be ejected from the system.

Parameter	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6	Set 7	Set 8
`N`	100	100	100	100	500	1000	1000	500
`t`	0	0	0	0	0	0	0	0
`tEnd`	20	20	20	15	40	40	40	20
`dt`	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.01
`softening`	0.1	0.1	0.1	0.1	0.01	0.1	0.1	0.1
`G`	1	1	1	1	0.1	0.1	0.1	0.1
`starBoost`	−30	0	0	−30	0	0	0	5
`law`	0.5	0.5	3	−0.5	3	3	3	3
`speed`	0.2	0.2	0.2	0.8	0.8	0.8	0.8	0
`zoom`	40	3	3	5	10	10	4	2
`seed`	58	58	58	58	58	58	58	58
`adjustVel`	False	False	False	False	True	False	False	False
`negativeMass`	True	False	False	True	False	False	False	False
`collisions`	False	False	False	False	True	True	True	True
`collThresh`	0.0	0.0	0.0	0.0	0.1	0.9	0.9	0.9
`expand`	1.0	0.0	2.0	0.0	0.0	−2.0	2.0	0.0
`origin`	‘Centroid’	‘Centroid’	‘Centroid’	‘Star_0’	‘Centroid’	‘Centroid’	‘Centroid’	‘Star_0’
`threeClusters`	False	False	False	False	False	True	True	False
`p`	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2
`Nstars`	0	0	0	0	0	0	0	1
`fps`	20	20	20	20	20	20	20	20
`my_dpi`	240	240	240	240	240	240	240	240

Table 2: Eight selected parameter sets covering various situations

3 Analysis of star collisions and collision graph

Before diving into the analysis of star collisions, I provide details about the `nbody.txt` dataset generated by the Python code when `saveData=True`. It consists of one row per star per time frame, with 13 fields in the following order: time, star ID, mass, color and position of the star at the time in question, centroid of the global star cluster and velocity of the star at the same time. The last three features (star position, centroid, and star velocity) are 3D vectors, with components corresponding to the X, Y and Z axis. Each time frame corresponds to a video frame.

This rather large synthetic dataset can be used to perform various analyses and benchmark predictive algorithms. It can be split into multiple subsets of stars or time periods, for cross-validation purposes. Subsets used to train a predictive model are called training sets, while those used to test the model are called validation sets. Due to the chaotic nature of the star system – it is indeed a chaotic dynamical system – some features such as individual star paths or drift of the whole system (similar to Brownian motion) may be difficult to predict, especially long-term. However some general features may be easier to predict such as the decreasing expansion rate of the star cluster, the number of rogue and twin stars over time, the average distance between neighboring stars, or the decaying rate of star collisions. Patterns such as star filaments or star clustering may be identifiable. In most cases, the stochastic processes at play are not stationary [Wiki]: they clearly exhibit trends.

In this section, I focus on star collisions and the collisions graph, attached to the star system corresponding to the seventh parameter set in Table 2. In particular, three distinct star clusters are initially created. All the data needed for this analysis is stored in a smaller dataset named `nbody_collisions.txt`. It contains the following fields: collision ID (the key to this table), the collision time or video frame, the IDs of the two stars involved, the cluster IDs the stars were assigned to at creation time (represented by the color), the masses of the two stars involved in the collision, and the distance between the impact location and the centroid of the whole system at the time of impact. The collision data set is available on my GitHub repository, here. The larger dataset `nbody.txt`, not analyzed in this article, is available here (87 MB compressed, 2 million rows).

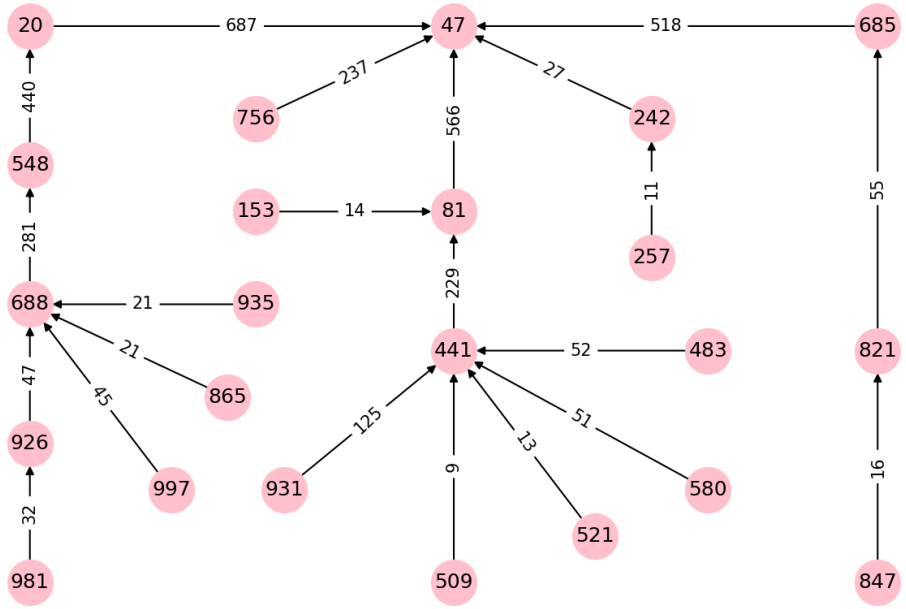

Figure 1: Collisions graph for the biggest star eater (star 47) in video 7

3.1 Weighted directed graphs: visualization with NetworkX

The directed graph in Figure 1 is produced with the networkx library in Python. The code is listed in section 5.2. The numbers in the pink circles represent a star ID. The weight attached to each arrow represents the time frame when the collision happened. The last collision for star 47 happens at time $t = 687$ (that is, in video frame 687, out of 2000 frames for the whole video). Note the chain of collisions $509 \mapsto 441 \mapsto 81 \mapsto 47$, starting at video frame 9. Star 47 ends up with a mass of 59.21, while star 509 has a mass of 2.90 before being eaten: the mass has grown by a factor 20 after all the collisions!

The total number of collisions across all stars is 349. At creation time ($t = 0$) they were 1000 stars. Mass accumulation resulting from collisions happens very fast in the early days, but rapidly reaches a maximum. In this case, star 47 becomes the largest one with the largest number of collisions, thus the reason to choose it for illustration. There are a few other stars with many collisions, though most stars experience zero or one collision.

Detecting all the collision graphs amount to detecting all the connected components of the whole graph [Wiki]. To that effect, I used the `PB_NN_graph.py` program described in my book on stochastic processes and simulations [5]. The Python code in question is also available on GitHub, here. Make sure that you input file is symmetric: if A, B is one of the collisions (between stars A and B), also include B, A in the input file. The sub-graph associated to star 47 in Figure 1 represents one of these connected components: the largest one. The list of all connected components, ordered by size, can be found here, with the star 47 merger series at the top; each number in each component represents a star.

There are several tools such as GraphViz [Wiki] to visualize the type of graph displayed in Figure 1. Here I used the NetworkX library in Python [Wiki]. However it is not trivial to make nice visualizations. The `draw` function offers many different layouts, illustrated here: spectral, spring, random, or Fruchterman-Reingold [Wiki]. Each layout chooses some optimum locations for the nodes, but none of them produces good results. In the end, I manually computed the locations of the nodes, though this process can be automated. Note that the graph in question is actually a tree [Wiki], with 47 being the root node. See here for an alternative solution to visualize this type of graph.

3.2 Interesting findings: how the universe got started

Figure 2 illustrates the behavior of the collisions when using the seventh parameter set in Table 2. The X-axis represents the time frame, each instance of time corresponding to a specific video frame. The time axis is truncated in the top left image as collisions become very rare over time. In the two other images, it is not on a linear scale but compacted to the right, for the same reason.

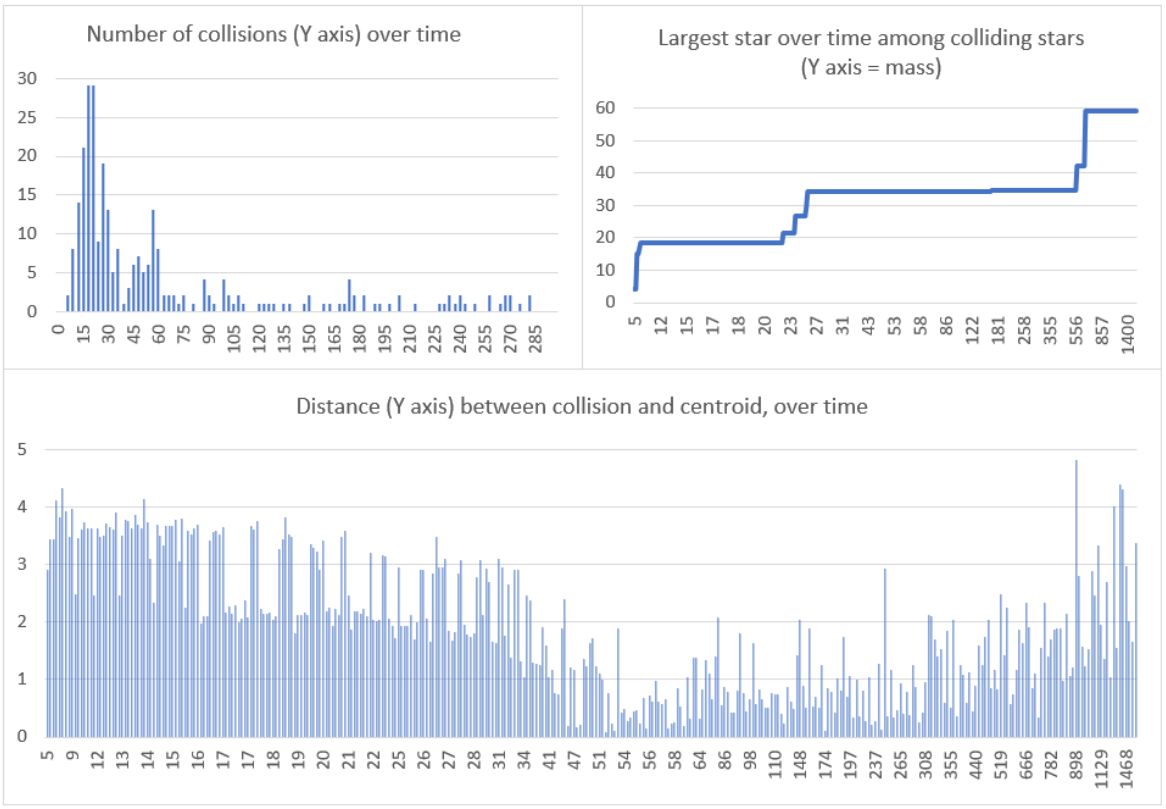

Figure 2: Summary statistics for the whole collision structure: the X axis represents the time

The behavior is typical for a standard universe. In particular, while collisions become rare over time, they come in waves, with each new wave being less intense than previous ones, and increased spacing between successive waves over time. The plot on the top right corner shows how quickly the star masses increase at the beginning due to collisions, with the largest mass 6 times above what it was at the beginning, in less than 30 video frames. The video has 2000 frames: 20 per second, and thus, it lasts one minute and 40 seconds. So 30 frames represents the first 1.5 second of the video.

The picture at the bottom of Figure 2 shows the complexity of the process. The Y axis represents the distance between a collision site and the evolving centroid of the global star cluster, for all the collisions occurring during the time frame. At the beginning, by design we have three star clusters, and collisions happen locally within each cluster. The collisions take place relatively far away from the global centroid, which is outside the three clusters. But very quickly, these three clusters coalesce, thus the distances to the centroid change, and many collisions eventually take place near the new centroid where larger stars are forming. The result is a decrease in the average distance to the centroid. But after a while (about 54 time frames, that is less than 3 seconds in the video), the distances start increasing again, as the universe expands and many collisions are not taking place near the centroid.

4 Animated data visualizations

The pictures in this section feature snapshots taken from the various synthetic universes generated using the parameter sets in Table 2. The full videos are on GitHub, here. Look for the MP4 files starting with `nbody` in the filename. The videos are also on YouTube, here. The parameter sets that I tested are described in section 2.3.

Common themes for standard universes (positive star masses with `law=3`) include decreasing expansion and reduced star velocities over time, twin stars that remain bonded only for so long, filaments, rogue stars ejected at high speed from a central location, small local clusters of stars moving around, drift of the whole system, and the creation of massive stars over time when collisions are allowed. Another question worth addressing is whether or not there is a dominant rotation sign: clockwise or anti-clockwise, depending on the initial configuration. In other words, are these systems anisotropic [Wiki]?

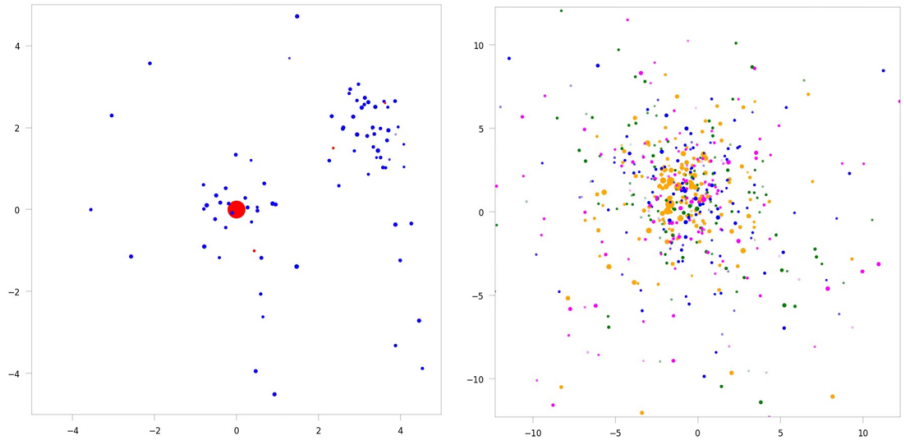

Figure 3: Snapshots of universe 4 (left) and universe 7 (right)

Figure 3 shows snapshots for two different universes, corresponding to parameter sets 4 and 7. The full videos can be watched respectively here and here. I encourage you to zoom in on the pictures to get a better view.

Universe 4 has a massive central star of negative mass, in red. In addition, the law of gravity is not inverse-square: instead, I use `law=-0.5` rather than the standard `law=3` in the Python code. Blue stars have a positive mass, and the whole system starts with one single star cluster. However it ends up with multiple clusters. The cluster in the top right corner, formed in the early stages, moves around the red star at wildly increasing speeds and eventually breaks apart. This pulsating universe exhibits a fast-growing entropy [Wiki] and ends up in a singularity [Wiki]: it actually crashed the Python program in the end, before completing the 2000 video frames. This explains why the corresponding video is 25 seconds shorter than the other ones. If you zoom in, you will notice that there are a few small stars with negative mass, besides the central one: they are pictured in red.

To the contrary, universe 7 is the classic, well-behaved version as we know it in real life. It starts with three separate clusters, but ends up with just one, as the clusters quickly coalesce. Entropy decreases over time, as the universe expands, albeit more and more slowly in the end. The green, blue and magenta colors indicate which cluster a star originally belongs to. The snapshot, taken in the middle of the video, shows that the three clusters are already well blended, forming a single cluster at this point. A number of twin stars can be seen, and may involve stars from different clusters. Once two stars collide, the color of the resulting star turns orange, explaining the concentration of orange stars (typically larger) near the center.

5 Python code and computational issues

There are two separate pieces of code: the main program for the simulations and video production in section 5.1, and the auxiliary program for graphs representation (visualizing the collision tree) in section 5.2.

5.1 Simulating the real and synthetic universes

The Python code is also on GitHub, here. The main parameters are described in section 2.2.

The bottleneck is the computation of all pairwise interactions. One way to dramatically improve performance is to ignore stars that are far away from each other. Instead of using a square matrix to store all the distances between stars, one could use a hash table [Wiki], where the key is a pair of stars and the value is the separating distance. If the number of stars is n, the size of the distance matrix is $n \times n$, but the hash table (a dictionary in Python) could be limited to (say) $20 \times n$ entries if $n > 200$. The use of a very coarse grid for star locations can help detect when two stars are getting closer to each other, requiring to add a new entry in the hash table, and possibly delete some entries.

Another improvement consists of embedding multiple universe simulations (each with its own video) as "subplots" into a single video. I describe how to do it, with an example, in my article on terrain generation [2]. Also, when the number of stars is very small, you could join the locations of a same star on adjacent frames with line segments, to show the orbit in the video. See how to do this in chapter 4 in my book "Intuitive Machine Learning and Explainable AI" [3].

Finally, I resize all images (the PNG files) before inclusion in the video. The video generator requires that they all have the same size.

```
# nbody.py | www.MLTechniques.com | vincent@MLTechniques.com | 2022

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import moviepy.video.io.ImageSequenceClip # to produce mp4 video

def getAcc( pos, mass, G, law, softening, col ):

   # Calculate the acceleration on each particle due to Newton's Law
   # pos is an N x 3 matrix of positions
   # mass is an N x 1 vector of masses
   # G is Newton's Gravitational constant
   # softening is the softening length
   # a is N x 3 matrix of accelerations
   #
   # Also: update collisionTable

   global ncollisions

   # positions r = [x,y,z] for all particles
   x = pos[:,0:1]
   y = pos[:,1:2]
   z = pos[:,2:3]

   # matrix that stores all pairwise particle separations: r_j - r_i
   dx = x.T - x
   dy = y.T - y
   dz = z.T - z

   # matrix that stores 1/r^(law) for all particle pairwise particle separations
   inv_r3 = np.sqrt(dx**2 + dy**2 + dz**2 + softening**2)
   inv_r3 = inv_r3**(-law)

   # detect collisions
   if collisions:
      threshold = collThresh * softening**(-law)
      for i in range(N):
        for j in range(i+1,N):
           if inv_r3[i][j] > threshold and mass[i] != 0 and mass[j] !=0:
              print("Collision between body",i,"and",j)
              dist=np.linalg.norm(pos[i] - centroid) # distance to centroid
              collData = str(ncollisions)+ " "+str(frame)+" "+str(i)+" "+str(j)
              collData = collData + " "+col[i]+" "+col[j]
              collData = collData +" "+str(mass[i])+" "+str(mass[j])+" "+str(dist)
              # collData = collData +" "+str(centroid)
              collisionTable.append(collData)
              ncollisions += 1
              mass[i]=mass[i]+mass[j]
```

```
                mass[j]=0
                col[i]='orange'
                col[j]='white'

    ax = G * (dx * inv_r3) @ mass
    ay = G * (dy * inv_r3) @ mass
    az = G * (dz * inv_r3) @ mass

    # pack together the acceleration components
    a = np.hstack((ax,ay,az))
    return a

def vector_to_string(vector):
    # turn numpy array entry into string of tab-separated values
    string = str(vector)
    string = " ".join(string.split()) # multiple spaces replaced by one space
    string = string.replace('[ ','').replace('[','')
    string = string.replace(' ]','').replace(']','')
    string = string.replace(' ',"\t") ## .replace("\t\t","\t")
    return string


#--- main

# Simulation parameters
N          = 1000    # Number of stars
t          = 0       # current time of the simulation
tEnd       = 40.0    # time at which simulation ends
dt         = 0.02    # timestep
softening  = 0.1     # softening length
G          = 0.1     # Newton's Gravitational Constant
starBoost  = 0.0     # create one massive star in the system, if starBoost > 1 or < -1
law        = 3       # exponent in denominator, gravitation law (should be set to 3)
speed      = 0.8     # high initial speed, above 'escape velocity', results in dispersion
zoom       = 10      # output on [-zoom, zoom] x [-zoom, zoom ] image
seed       = 58      # set the random number generator seed
adjustVel  = False   # always True in original version
negativeMass = False # if true, bodies are allowed to have negative mass
collisions = True    # if true, collisions are properly handled
collThresh = 0.9     # < 1 and > 0.05; fewer collisions if close to 1
expand     = -2.0    # enlarge window over time if expand > 0
origin     = 'Centroid' # options: 'Star_0', 'Zero', or 'Centroid'
threeClusters = True # if true, generate three separate star clusters
p          = 0.0     # add one new star with proba p at each new frame if p > 0
Nstars     = 0       # if p > 0, start with Nstars; will add new stars up to N, over time
fps        = 20      # frames per second in video
my_dpi     = 240     # dots per inch in video
createVideo = True   # set to False for testing purposes (much faster!)
saveData   = False   # save data to nbody.txt if True (large file!)

# Handle configurations that are not supported
if threeClusters and p > 0:
    print("Error: adding new stars not supported with threeClusters set to True.")
    exit()
if Nstars >= N:
    print("Error: Nstars must be <= N.")
    exit()

# Generate Initial Conditions
np.random.seed(seed)
if negativeMass:
    mass = 1.25 + 0.75*np.random.randn(N,1)
else:
    mass = np.random.exponential(2.0,(N,1))
adjustedMass = 1
if starBoost > 1 or starBoost < 0:
```

```
   mass[0]= starBoost * np.max(abs(mass))
col=[] # bodies with positive mass in blue; other ones in red
for k in range(N):
   if mass[k] > 0:
      col.append('blue')
   else:
      col.append('red')
   if p > 0 and k >= Nstars:
      mass[k] = 0  # make room for future stars
      col[k] = 'darkviolet' # newly added stars appear in pink

pos = np.random.randn(N,3) # randomly selected positions and velocities
if threeClusters:
   for k in range(int(N/3)):
      pos[k] += [5.0, 0.0, 0.0]
      col[k] = 'green'
   for k in range(int(N/3),int(2*N/3)):
      pos[k] += [0.0, 5.0, 1.0]
      col[k] = 'magenta'
vel = speed * np.random.randn(N,3)

# Convert to Center-of-Mass frame
if adjustVel:
   for k in range(N):
      vel[k] -= np.mean(abs(mass[k]) * vel[k]) / np.mean(abs(mass))

# calculate initial gravitational accelerations
frame=-1
acc = getAcc( pos, mass, G, law, softening, col )

# number of timesteps (or frames in the video)
Nt = int(np.ceil(tEnd/dt))

# prep figure
fig = plt.figure(figsize=(4,5),dpi=80)
ax1 = fig.gca() # or ax1 = plt.subplot() ??
plt.setp(ax1.spines.values(), linewidth=0.1)
plt.rc('xtick', labelsize=5) # fontsize of the tick labels
plt.rc('ytick', labelsize=5) # fontsize of the tick labels
ax1.xaxis.set_tick_params(width=0.1)
ax1.yaxis.set_tick_params(width=0.1)

flist=[]      # list of image filenames for the video
collisionTable=[] # collision table
ncollisions=1

if Nt > 2000:
   print("About to generate", Nt, "images.")
   answer = input ("Type y to proceed: ")
   if answer != 'y':
      exit()

# Simulation Main Loop

if saveData:
   OUT=open("nbody.txt","w")

for frame in range(Nt):
   if p > 0 and Nstars < N: # add new star with proba p
      if np.random.uniform() < p:
         mass[Nstars] = np.random.exponential(2.0,1)
         Nstars += 1

   vel += acc * dt/2.0 # (1/2) kick
   pos += vel * dt # drift
   acc = getAcc( pos, mass, G, law, softening, col ) # update accelerations
```

```
    vel += acc * dt/2.0 # (1/2) kick
    t += dt # update time

    image='nbody'+str(frame)+'.png' # filename of image in current frame
    if frame % 10 == 0:
        print("Creating image",image) # show progress on the screen

    plt.sca(ax1)
    plt.cla()
    centroid = np.zeros(3)
    totalMass=np.sum(abs(mass))
    if origin == 'Star_0':
        centroid = pos[0]
    elif origin == 'Centroid':
        for k in range(N):
            centroid += abs(mass[k]) * pos[k] / totalMass

    # save results
    if saveData:
        for k in range(N):
            line=str(frame)+"\t"+str(k)+"\t"+str(float(mass[k]))+"\t"+str(col[k])+"\t"
            string1 = vector_to_string(pos[k])
            string2 = vector_to_string(centroid)
            string3 = vector_to_string(vel[k])
            line=line+string1+"\t"+string2+"\t"+string3+"\n"
            OUT.write(line)

    adjustedMass /= (1.0 + expand/Nt) # for visualization only
    plt.scatter(pos[:,0]-centroid[0],pos[:,1]-centroid[1],
                          s=adjustedMass*abs(mass),color=col)
    zoom *= (1.0 + expand/Nt)

    ax1.set(xlim=(-zoom, zoom), ylim=(-zoom, zoom))
    ax1.set_aspect('equal', 'box')

    if createVideo and frame>0:
        # plt.axis('off')
        plt.savefig(image,bbox_inches='tight',pad_inches=0.2,dpi=my_dpi)
        im = Image.open(image)
        if frame == 1:
            width, height = im.size
            width=2*int(width/2)
            height=2*int(height/2)
            fixedSize=(width,height)
        im = im.resize(fixedSize)
        im.save(image,"PNG")
        flist.append(image)
    plt.pause(0.001)
if saveData:
    OUT.close()

# output collision table
OUT2=open("nbody_collisions.txt","w")
for entry in collisionTable:
    OUT2.write(vector_to_string(entry)+"\n")

# output video / fps is number of frames per second
if createVideo:
    clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=fps)
    clip.write_videofile('nbody.mp4')
```

5.2 Visualizing collision graphs

This code is also on GitHub, here. For explanations, see section 3.1.

```
# nbody_graph.py | www.MLTechniques.com | vincent@MLTechniques.com | 2022
# collision history for star # 47 (biggest star eater) using parameter set # 7

import networkx as nx
# https://www.python-graph-gallery.com/322-network-layout-possibilities
# graph layouts: https://networkx.org/documentation/stable/auto_examples/index.html

# importing matplotlib.pyplot
import matplotlib.pyplot as plt

G = nx.DiGraph(directed=True)

# define the graph; each entry is (node, next node, weight)
E = [(509, 441, 9),
    (257, 242, 11),
    (521, 441, 13),
    (153, 81, 14),
    (847, 821, 16),
    (821, 685, 55),
    (865, 688, 21),
    (935, 688, 21),
    (242, 47, 27),
    (981, 926, 32),
    (997, 688, 45),
    (926, 688, 47),
    (580, 441, 51),
    (483, 441, 52),
    (821, 685, 55),
    (931, 441, 125),
    (441, 81, 229),
    (756, 47, 237),
    (688, 548, 281),
    (548, 20, 440),
    (685, 47, 518),
    (81, 47, 566),
    (20, 47, 687)]

G.add_weighted_edges_from(E)

# specify locations of the nodes on the graph
pos = {441: (3, 2.6),
     931: (1.8, 2),
     483: (4.8, 2.6),
     580: (4.8, 2),
     521: (4, 1.8),
     509: (3, 1.6),
     81: (3, 3.2),
     153: (1.6, 3.2),
     47: (3, 4),
     756: (1.6, 3.6),
     685: (6, 4),
     821: (6, 2.6),
     847: (6, 1.6),
     20: (0,4),
     548: (0, 3.4),
     688: (0, 2.8),
     997: (1, 2),
     935: (1.6, 2.8),
     865: (1.4, 2.4),
     926: (0, 2.2),
     981: (0, 1.6),
     242: (4.4, 3.6),
     257: (4.4, 3.0)}

nx.set_node_attributes(G, pos, 'coord')
```

```
nx.draw(G, pos, with_labels=True, node_size=700, node_color='pink') ### ,
    font_weight="bold")
edge_weight = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_weight)
plt.savefig("graph.png")
plt.show()
```

References

[1] Vincent Granville. The art of visualizing high dimensional data. *Preprint*, pages 1–17, 2022. MLTechniques.com [Link]. 2

[2] Vincent Granville. Dynamic clouds and landscape generation: Morphing and evolutionary processes. *Preprint*, pages 1–9, 2022. MLTechniques.com [Link]. 8

[3] Vincent Granville. *Intuitive Machine Learning and Explaible AI*. MLTechniques.com, 2022. To be published in October 2022 [Link]. 2, 8

[4] Vincent Granville. New perspective on the Riemann Hypothesis. *Preprint*, pages 1–23, 2022. MLTechniques.com [Link]. 2

[5] Vincent Granville. *Stochastic Processes and Simulations: A Machine Learning Perspective*. MLTechniques.com, 2022. [Link]. 5