Generating and Videolizing Agglomerative Processes

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.0, April 2023

Abstract

This short article explains how to efficiently simulate the evolution of agglomerative processes, and visualize their behavior with data animations. I use a generic, simple model for illustration purposes: atoms, initially consisting of one electron, collide and merge over time, with a pre-specified maximum number of electrons per atom: the maximum limit. Given enough time, small atoms eventually disappear, leading to a universe with heavy atoms only.

The focus is on the distribution of atom sizes over time and the number of collisions, becoming rarer and rarer as atoms get bigger and the density of atoms per unit volume decreases. You can use the method and Python implementation in various contexts by updating the algorithm accordingly, and changing the terminology: replacing atoms by particles, or small celestial bodies in the birth of a start system with planets, or molecules, or even the soap bubbles merging together. A potential simple improvement is to allow the atoms to not only merge and grow in size, but also to split or lose electrons.

Perhaps the most interesting feature is that you can simulate the evolution and interactions of the 10^{80} atoms in our universe without working with individual atoms. Indeed, I use very fast and efficiently simulations based on arrays with fewer than 100 cells, to study the macro behavior. The implementation allows you to simulate either one or hundreds of evolution paths in parallel. The second option leads to the theoretical evolving distribution of atom sizes over time. It can be customized to a variety of agglomerative processes.

Contents

1 Introduction

I use the keywords "atom" and "electron" to help people materialize what I mean by agglomerative process. In short, a system that starts with elementary particles – atoms with one electron – and evolves over time as the result of mergers or collisions. In this case, atoms with more and more electrons as they combine together. But the idea is not restricted to atoms only, and even in the case of atoms, the agglomeration mechanism described here may not correspond to the reality. Think of it as a synthetic universe where atoms combine according to laws that you can customize and not necessarily compatible with standard physical laws.

The size of an atom is defined as its number of electrons. I start with N atoms all with one electron, at time $t_0 = 0$. A collision occurs at time t_1, t_2 and so on, randomly between two atoms of any size, resulting in the loss of the two atoms in question, and the creation of a new larger atom. The size of the new atom is the sum of the sizes of the two atoms involved in the collision. The maximum size allowed in a collision is set to 50 in the Python implementation. This upper limit is customizable and denoted as `limit` in the code.

The probability of a collision does not depend on the size of the atoms involved. However, it would be easy to change this rule, or even to allow the atoms to break apart into smaller atoms. The timing of the collisions (t_1, t_2 and so on) depends on the atom density in the system at any given time. Here, it is inversely proportional to the number of atoms in the system. Thus collisions become rarer and and rarer over time. In other words, the time between successive collisions increases over time. Eventually, after a very long time period, the system

gets stuck in a final configuration where no more collisions can take place. The final configuration is random and will be different in each simulation, even if starting with the same number of atoms.

The purpose of this article is to study the trajectory (also called path or orbit) of the system: the atom size distribution at any given time, the timing of the collisions, the mean, maximum and minimum atom size at any given time, the time to equilibrium, the speed of the transitions and so on. Both for a single simulation, as well as averages across a very large number of simulations all starting with the same number of atoms.

2 Atom size distribution

You can infer the atom size distribution from the model described in section 1. I first describe how to perform the computations based on a single simulation. Then I show how you can approximate the expected or theoretical distribution when blending a large number of simulations, all starting with N atoms, each with one electron. Figure 1 shows the observed distribution based on one simulation, at four different times: soon after the beginning (top left), after some time (top right), after even more time (bottom left) and the final configuration or attractor (bottom right). No more collision occur once reaching the final configuration. This configuration is reached in a finite but very large number of steps.

In Figure 1, the X-axis represents the size of the atoms, while the Y-axis represents the frequencies, given the size. For instance, the proportion of atoms of size 1 is 100% at the beginning. At the end this proportion is 0%, and the proportion of atoms of size 50 (the maximum size allowed) is about 9%. However the 9% may slightly vary depending on the simulation. A data animation featuring how the distribution evolves over time can be seen in this video. I produced it by averaging the distribution across 100 simulations. In the video, the time is rescaled so that successive video frames correspond to visible changes in the distribution. Otherwise, the video would look static for a very long time, and show dramatic changes over very short time periods. Time rescaling evens out these changes.

In this example, $N = 8000$, and the number of iterations is a lot larger. This is because in the implementation, when approaching the final state, collisions become very rare as most of them (if allowed) would result in atoms with a size larger than 50.

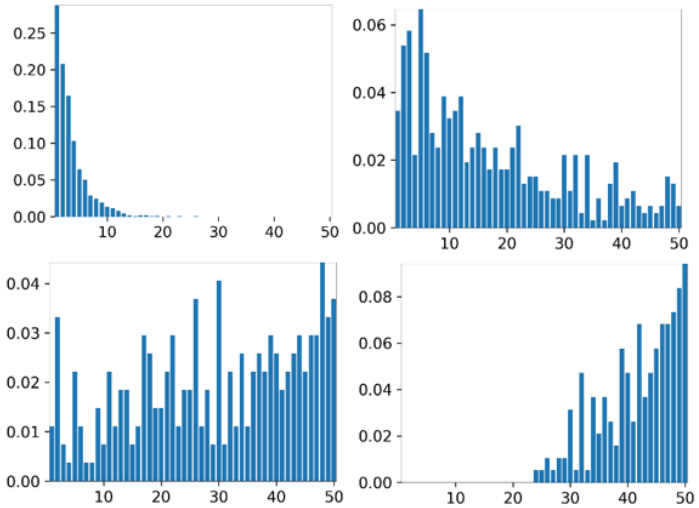

Figure 1: Atom sizes over time, ranging from 1 to 50 electrons

2.1 Core algorithm: single simulation path

For a comprehensive study of particle aggregation, collision rate, speed of particles and collision theory [Wiki] in the context of statistical mechanics, see the books "A Kinetic View of Statistical Physics" [3], and "Statistical Mechanics" [1]. The goal here is considerably more modest but also very different: to provide a practical implementation, easy to customize depending on the context, leading to useful visualizations, and videos in particular. Also, the focus is on the macro-behavior, not the interactions of individual particles. Finally, the context may be very different from statistical physics and could even include mergers and acquisitions in the business world, celestial mechanics, or applications in chemistry.

At the core of the algorithm is the sampling procedure: randomly selecting two atoms of arbitrary size, to generate a collision. Not all sizes may be available. At the beginning, we only have small atoms. Towards the end, small atoms are gone. The average and maximum size of an atom is pictured on the left plot in Figure 2,

for a typical evolutionary path, where the X-axis represents the time. The right plot shows the cumulative number of collisions over time. If an attempted collision results in an atom with more than 50 electrons (the `limit` parameter in the Python code), it is rejected. This explains the convex shape and plateauing of the curve on the right plot. However, in case of failure, the Python code, via the parameter `max_trials`, allows you to try multiple times at any given step until a collision is accepted. This parameter is set to 1 in Figure 2.

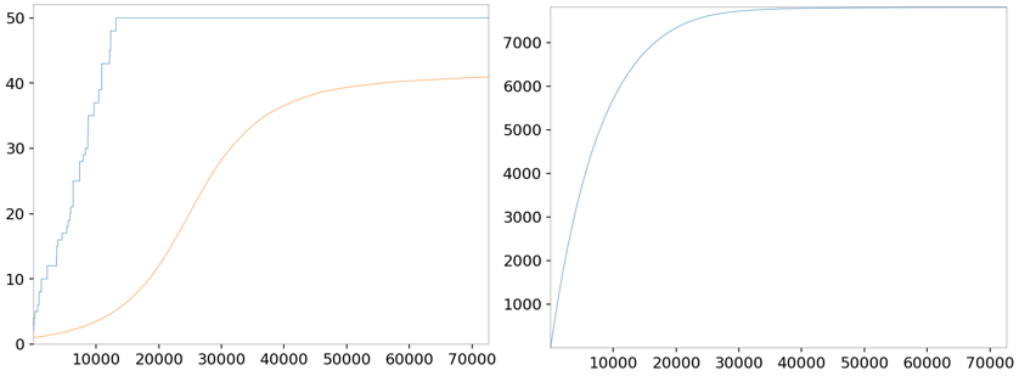

Figure 2: Mean vs max atom size (left), collisions over time (right); X-axis is time

To sample candidate atoms for a collision, you first sample the two atom sizes, denoted as `k` and `l` in the code (see section "core of the algorithm"). The frequency table for atom sizes is stored in the array `pvals`. To perform one single simulation, set `N_simul=1`. Then `pvals` is a one dimensional array. Otherwise there is a separate entry for each simulation, and `pvals` is thus a 2D array. Both cases are treated the same way: there is no distinction between one or multiple simulations.

Sampling `k` and `l` is performed the classic way, using the empirical CDF (cumulative distribution function) attached to the histogram `pvals`. To sample deviates from the empirical CDF, call the function `sample_size`. If some size is missing (no atom with the size in question), it will not cause any problem: missing sizes are implicitly ignored by the sampling procedure. After sampling the first size `k`, one atom of size `k` must be removed from `pvals` before sampling `l`. Note that `k` and `l` may be identical, especially at the beginning.

Now that `k` and `l` have been selected (and the sum is smaller or equal to 50, possibly after a few trials), you need to update the atom counts in the main array `atom_sizes`. This array controls the evolution of the system. In particular `atom_sizes[simul,k]` is the number of atoms of size `k`, for a particular simulation `simul` (here there is just one simulation), at any given iteration. We now need to update this array after the collision, as follows:

- Decrease the number of atoms of size `k` by 1,
- Decrease the number of atoms of size `l` by 1,
- Increase the number of atoms of size `k+l` by 1,
- Update the time.

Again, note that we might have `k=l`. Thus a collision of two atoms of the same size requires that we have at least two atoms of that size prior to the collision. Also, prior to the collision, it is possible that the number of atoms of size `k+l` is zero. I now discuss how the time is updated, in section 2.2.

2.2 Time scale

To mimic some natural processes, the collision times t_1, t_2 and so on are updated so that the difference between two subsequent events is proportional to the inverse of the total number of atoms in the system prior to the collision. Also, not all attempts result in an actual collision, due to the constraints. The time may or may not be updated when an attempted collision is rejected or impossible. To increase the chance of an actual collision at each step, you can increase the parameter `max_trials`. However, towards the end, once you entered into a final configuration (the equivalent of an absorbing state in a Markov chain), no matter the number of trials, it is not possible to change the system.

To give a sense of timing, the top left plot in Figure 1 corresponding to "early times" is supposed to represent the universe (atom distribution) after dozens of billions of years. That is, a lot older than the current universe. Hydrogen (atoms of size 1) still dominate, but they represent less than 30% of all atoms. Today, hydrogen represents 90% of all atoms. In a few trillion years, the distribution will look like the bottom right plot, where heavy elements now dominate, and hydrogen, helium and all the light elements are gone. Of course,

this is science fiction and not a prediction, as the actual laws of evolution are not the same as those used in the Python code. I set the upper limit to 50 for the size of an atom, as the size can not grow indefinitely, heavy elements eventually decay over time, and elements heavier than iron (size 26) are harder to produce. A more realistic model is to assign a non-uniform probability p_{kl} for two elements of size k and l, to merge. And allow atoms to disintegrate at various speeds depending on size.

The evolution mechanism implemented here is more appropriate in contexts other than astronomy, especially regarding the long term. But you can customize it to your needs. In the Python code, the array `arr_t` contains the timing of all attempted collisions, for each simulation. It is updated after any attempted collision. The index `iter` in the main loop counts attempted collisions.

2.3 Averaging across multiple simulations

Figure 1 shows results obtained with one simulation. It is also easier to understand the implementation when there is only one simulation. In the video posted here, the distribution is averaged across 100 simulations. It is a lot smoother, and a good approximation of the theoretical distribution. By contrast, my first video involved one simulation, and can be viewed here.

In the implementation, the number of simulations is determined by `N_simul`. The main loop is over `iter` (also called step or iteration in this article), with each increment corresponding to an attempted collision. The inner loop is over the simulations, indexed by `simul` in the code. Thus, at each iteration, a collision is attempted separately for each of the `N_simul` simulations. The parallel implementation works as follows: all the important statistics are stored in arrays with one extra dimension: the simulation number. This includes historical data, attempted collision times, cumulated number of collisions, the atom counts for each atom size, and the total number of atoms at the current iteration, with one set of values attached to each simulation. These arrays are respectively denoted as `history_min`, `history_max`, `history_mean`, `history_time`, `history_collisions`, `atom_sizes`, and `arr_N` in the code.

2.4 Collision graphs

In order to study collision graphs in details, you need to look at individual atoms and their interactions. This is easy when you have only a few thousand atoms. In this article, the goal is to investigate global properties rather than local behavior, so I did not look at collision graphs. However, I studied the local behavior in my book on synthetic data [2]. The goal was to produce synthetic collision graphs, in the context of star collisions. Figure 3 shows an example of such graph. The number in each circle represents the star involved, and the number attached to each arrow represents the time when the collision occurred.

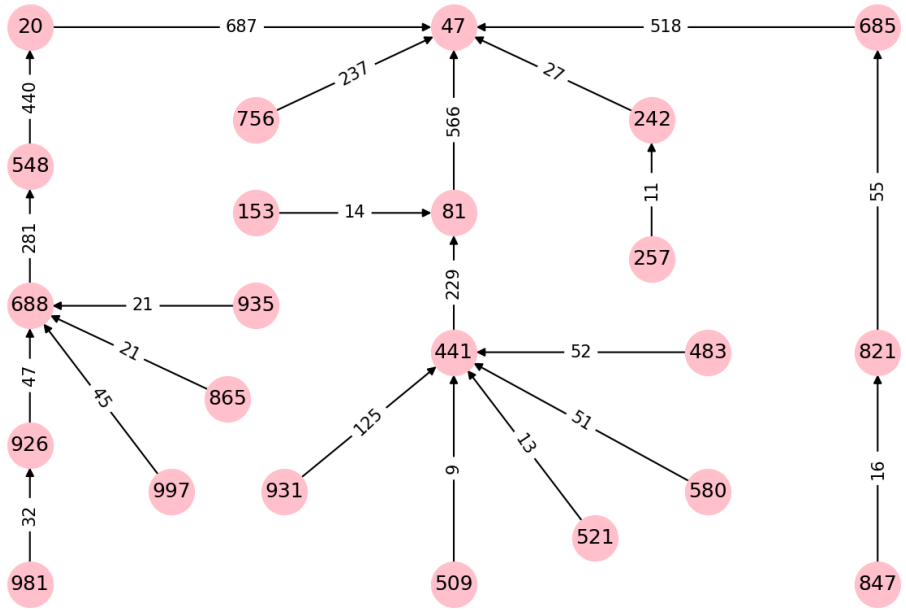

Figure 3: Example of collision graph

For instance, the arrow with weight 237 between stars 756 and 47 means that stars 756 and 47 collided and merged at time 237. The newly formed star kept the label 47, because it was the largest of the two in the

collision, having absorbed more stars in the past. In this example, stars were moving according to some physical laws in a synthetic universe, and a collision occurred each time two stars were passing too close to each other.

3 Python implementation

The code, also available on GitHub here, is much simpler than it looks at first glance. The core of the algorithm is in the small section entitled "core of the algorithm". It is easy to understand what it does by first assuming that the number `N_simul` of simulations is 1, thus eliminating the inner loop. If you also limit the number of trials to 1 when attempting a collision, it eliminates the deepest loop, reducing it to one step.

The code is somewhat long because of the numerous options. A good chunk focuses on producing high-quality videos and images, including the functions `my_plot_init` and `save_image`. The latter resizes the images so that they all have the same size and an even number of pixels, a requirement to produce the video. Quite a bit of code is devoted to collecting historical data: this also includes the function `get_summary_stats`.

Figure 4 is a screenshot of the output generated. Frame indicates the number of video frames produced so far: you can choose the schedule for frame generation, to not save an image at each iteration. Otherwise, the code would produce a large number of images, including many subsequent images that are almost identical. The time increases a lot faster than the iteration counter after a while, due to many attempted collisions being rejected as they result in atoms larger than allowed. Also collisions become rarer over time, as the likelihood of a collision is determined by the number of atoms present in the system, decreasing by one after each collision. In short, the smaller the number of atoms, the longer the time elapsed between collisions: this is handled by updating the time accordingly.

```
Frame: 362 Iteration:  8965 Time: 71249 Mean size: 40.90
Frame: 362 Iteration:  8966 Time: 71289 Mean size: 40.91
Frame: 362 Iteration:  8967 Time: 71330 Mean size: 40.91
Frame: 362 Iteration:  8968 Time: 71371 Mean size: 40.91
Frame: 362 Iteration:  8969 Time: 71412 Mean size: 40.91
Frame: 362 Iteration:  8970 Time: 71453 Mean size: 40.91
Frame: 363 Iteration:  8971 Time: 71494 Mean size: 40.91
Frame: 363 Iteration:  8972 Time: 71535 Mean size: 40.92
Frame: 363 Iteration:  8973 Time: 71576 Mean size: 40.93
Frame: 363 Iteration:  8974 Time: 71617 Mean size: 40.93
Frame: 363 Iteration:  8975 Time: 71658 Mean size: 40.93
Frame: 363 Iteration:  8976 Time: 71699 Mean size: 40.93
Frame: 363 Iteration:  8977 Time: 71740 Mean size: 40.93
Frame: 363 Iteration:  8978 Time: 71780 Mean size: 40.93
Frame: 363 Iteration:  8979 Time: 71821 Mean size: 40.93
Frame: 363 Iteration:  8980 Time: 71862 Mean size: 40.94
```

Figure 4: Summary stats starting at video frame 362

Finally, the mean size represents the average atom size at a given iteration (averaged over all simulations). If the maximum size allowed is 50 and the current mean is above 40, you know that you are close to the final configuration: the absorbing state, after which no more collision can take place. In this example, the initial number of atoms was 8000. The mean size in the final configuration is around 42.

To set the frame generation schedule, and thus determine how the final video will look like, you have three options, offered by the parameter `mode`:

- `mode='time'` allows you to create a new frame after a fixed time increment. It also allows you to skip the early times when very little action is visible.
- `mode='n_collisions'` allows you to create a new frame after a fixed number of attempted collisions. However, this number should be large at the beginning and small at the end, otherwise the video will have some sections moving very slowly, and some sections moving very fast. You can also skip the first few hundred collisions, as action is not very visible during this time period (in other words, things are moving slowly initially).
- `mode='atom_size'` allows you to create a new frame each time the mean atom size has increased by a pre-specified threshold. It creates the best videos visually speaking, but also the least realistic as time gets considerably distorted. The benefit is that everything seems to be moving at the same speed throughout the evolution.

The parameter `yaxis` allows you to choose between a fixed Y-axis with values between 0 and 1, or an Y-axis that self-adjusts over time. With a fixed Y-axis, changes are not slow initially, but quite rapidly, the distribution of sizes considerably spread between 1 and the maximum authorized, making the bar chart looks smaller and smaller, eventually barely visible. Choosing a variable Y-axis is the best solution in my opinion,

but it makes the beginning of the evolution process seem rather static. This effect can be countered by setting mode='atom_size'.

```
# atoms.py
# Generating and videolizing agglomerative processes

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import moviepy.video.io.ImageSequenceClip

N     = 8000  # initial number of particles (hydrogen atoms)
n_iter = 9000 # number of iterations (time)
limit = 50    # max number of electrons/atom allowed
N_simul = 100 # number of systems (atom configurations) running in parallel
max_trials = 1 # trials allowed until 1 collision happens
seed = 87
np.random.seed(seed)

atom_sizes = np.zeros((N_simul,limit+1))
collisions = np.zeros(N_simul)
arr_N = []
arr_t = []
N_0 = N
for simul in range(N_simul):
   atom_sizes[(simul,1)] = N
   arr_N.append(N)
   arr_t.append(0.0)

def sample_size(pvals):
   u = np.random.uniform()
   k = 0
   cdf_val = pvals[0]
   while cdf_val < u:
      k = k + 1
      cdf_val = cdf_val + pvals[k]
   return(k)

def my_plot_init(plt):
   # produces professional-looking plots
   axes = plt.axes()
   [axx.set_linewidth(0.2) for axx in axes.spines.values()]
   axes.margins(x=0)
   axes.margins(y=0)
   axes.tick_params(axis='both', which='major', labelsize=12)
   axes.tick_params(axis='both', which='minor', labelsize=12)
   return()

def save_image(fname,frame):
   global fixedSize
   plt.savefig(fname, bbox_inches='tight')
   # make sure each image has same size and size is multiple of 2
   # required to produce a viewable video
   im = Image.open(fname)
   if frame == 0:
      # fixedSize determined once for all in the first frame
      width, height = im.size
      width=2*int(width/2)
      height=2*int(height/2)
      fixedSize=(width,height)
   im = im.resize(fixedSize)
   im.save(fname,"PNG")
   return()

def get_summary_stats(pvals):
   min = -1
```

```
    mean = 0
    for k in range(limit+1): # k is the atom size
        if pvals[k] > 0:
            if min == -1:
                min = k # minimum size
            max = k    # maximum size
            mean += k * pvals[k]
    return(min, max, mean)


#---
mode = 'atom_size' # options: 'n_collisions', 'time' or 'atom_size'
yaxis = 'variable' # options: 'fixed' or 'variable'
show_last_frame = False # options: True or False
n_frames = -1
t_last = 0.0
n_collisions_last = 0
mean_last = 1
my_dpi = 300     # dots per each for images and videos
width = 2400     # image width
height = 1800    # image height
flist = []       # list image filenames for video
history_time = []
history_min = [] # min atom size over time
history_max = [] # max atom size over time
history_mean = [] # mean atom size over time
history_collisions = []

for iter in range(n_iter+1):

    #-- core of the algorithm (agglomeration)
    for simul in range(N_simul):
        old_N = arr_N[simul]
        trial = 0
        while old_N == arr_N[simul] and trial < max_trials:
            arr_t[simul] += N_0/arr_N[simul]
            pvals = np.copy(atom_sizes[simul,:])
            pvals = pvals / old_N
            k = sample_size(pvals)
            aux = np.copy(atom_sizes[simul,:])
            aux[k] = aux[k] - 1 # must be >= 0
            pvals = aux /(old_N - 1)
            l = sample_size(pvals)
            if k + l <= limit and N > 1:
                atom_sizes[(simul,k)] = atom_sizes[(simul,k)] - 1 # must be >= 0
                atom_sizes[(simul,l)] = atom_sizes[(simul,l)] - 1 # must be >= 0
                atom_sizes[(simul,k+l)] = atom_sizes[(simul,k+l)] + 1
                arr_N[simul] = old_N - 1
                collisions[simul] += 1
            trial +=1

    #-- compute summary stats across the N_simul simulations and over time
    pvals = np.zeros(limit+1)
    for k in range(limit+1):
        for simul in range(N_simul):
            pvals[k] += atom_sizes[simul,k]/arr_N[simul]
        pvals[k] /= N_simul # proba a particle (atom) is of size k
    mean_time = np.mean(arr_t)
    mean_collisions = np.mean(collisions)
    (min, max, mean) = get_summary_stats(pvals)
    print("Frame:%4d Iteration:%6d Time:%6.0f Mean size:%6.2f"
        % (n_frames, iter, mean_time, mean))
    history_time.append(mean_time)
    history_min.append(min)
    history_max.append(max)
    history_mean.append(mean)
```

```
    history_collisions.append(mean_collisions)

    #-- visualizations
    show_image = False
    if mode == 'time':
       if mean_time - t_last > 20 and mean_time > 50:
          show_image = True
          t_last = mean_time
    elif mode == 'n_collisions':
       if mean_collisions - n_collisions_last > 30 and mean_collisions > 500:
          show_image = True
          n_collisions_last = mean_collisions
    elif mode == 'atom_size':
       if mean - mean_last > 0.1:
          show_image = True
          mean_last = mean
    if show_last_frame and iter == n_iter - 1:
      show_image = True
    if show_image:
       if n_frames == -1:
          n_frames = 0
       plt.figure(figsize=(width/my_dpi, height/my_dpi), dpi=my_dpi)
       my_plot_init(plt)
       if yaxis == 'fixed':
          if n_frames == 0:
             y_axis_lim = max(pvals)
          plt.ylim(0, y_axis_lim)
       x_axis = np.linspace(0,limit,limit+1)
       plt.bar(x_axis[1:], pvals[1:])
       fname='histo_frame'+str(n_frames)+'.png'
       flist.append(fname)
       save_image(fname,n_frames)
       n_frames += 1
       plt.close()

clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=10)
clip.write_videofile('histo.mp4')

plt.close()
my_plot_init(plt)
plt.ylim(0,max+2)
plt.plot(history_time,history_max, linewidth=0.4)
plt.plot(history_time,history_mean, linewidth=0.4)
plt.show()
plt.close()

my_plot_init(plt)
plt.plot(history_time,history_collisions, linewidth=0.4)
plt.show()
plt.close()
```

References

[1] Paul Beale. *Statistical Mechanics*. Academic Press, third edition, 2011. 2

[2] Vincent Granville. *Synthetic Data and Generative AI*. MLTechniques.com, 2022. [Link]. 4

[3] Pavel Krapivsky, Sidney Redner, and Eli Ben-Naim. *A Kinetic View of Statistical Physics*. Cambridge University Press, 2010. [Link]. 2