3_animations

May 28, 2024

1 Animations

File as PDF

Animations are sometimes a nice feature, if you want to have a look at how your calculations evolve over time. Especially in the case of our particle based simulations it seems useful to diplay the position of the particles over time.

There are multiple ways to animate plots and images in python. Not all of them are always transferable between computers. Matplotlib for example provides a `animate` function, which can also use the `ffmpeg` video compressor. But that requires installation of all of them on the specific system you are working on.

We want to use the `ipycanvas` module, because it delivers easy drawing of shapes like circles or rectangle in a canvas directly in our Jupyter Notebooks. Have a look at the (documentation). The module provides even tools to interact with a mouse.

1.1 Import Modules

Before we start, lets have a look at the modules we import this time. Except the NumPy modules we never used them before. We import `time`, `threading`, and `ipycanvas`.

```
import numpy as np
from time import sleep,time
from threading import Thread

from ipycanvas import MultiCanvas, hold_canvas,Canvas
```

The `time` module is python standard module, which contains timing-related functions like the `sleep` and `time` function for example.

- The function suspends execution of the current thread for a given number of seconds.
- The `time()` function returns the number of seconds passed since epoch. For Unix system, `January 1, 1970, 00:00:00` at UTC is epoch (the point where time begins).

The `threading` module is a module which allows you to specify how the processes in your notebook are executed.

The `ipycanvas` module is the one which helps us to draw the objects of our simulation.

The following lines test if the notebook is running in google colab and install/enable ipycanvas there.

1.2 Particle class

We start by using out colloidal particle class, which we developed in the last section.

1.3 Create a set of particles

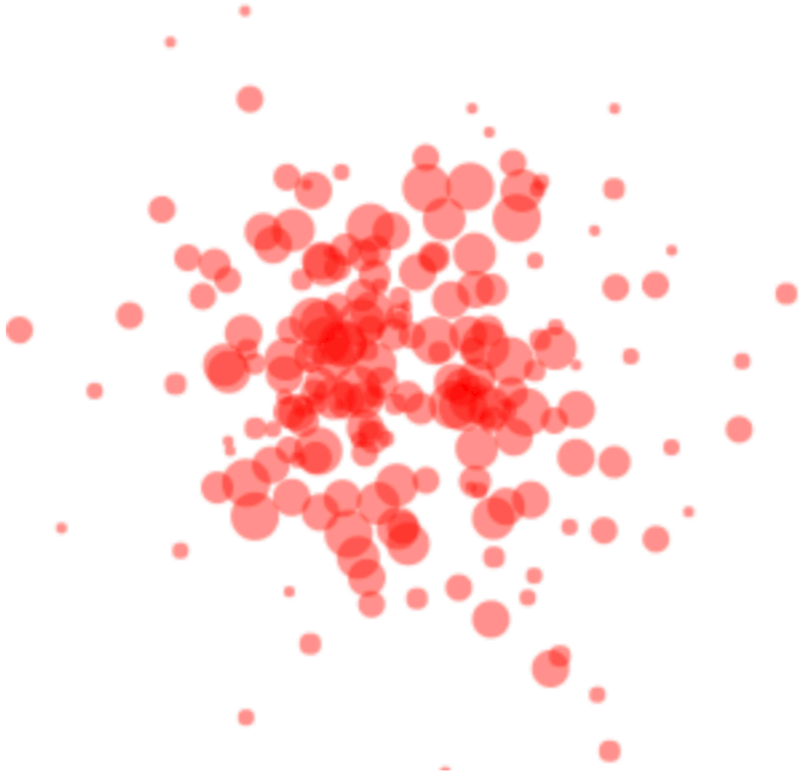
We want to animate the Brownian motion of many particles. The best is therefore to create a `list` which contains all the individual `Colloid` objects. We start by creating 200 colloids at the position `(0,0)` to see how the spread from the origin. They are stored in the list `p`.

1.4 Canvas and drawing function

Next, we need a canvas, in which we draw our particles and we need the drawing function.

The canvas is created by the `Canvas()` constructor if `ipycanvas`. The `display` command displays the canvas below the the cell.

```
Canvas(height=300, width=300)
```



We realize the drawing in a `for` loop, which is first updating all particle positions and then drawing all the particles.

The loop contains one interesting statement, which is the `with hold_canvas(canvas):`. Its useful to know that this statement halts the execution of all subsequent drawing comments in the following code block to create a "batch" of drawing commands send at one to the `ipycanvas` module. This will allow fast drawing of the whole scene. All the rest of the commands are shortly explained in the loop comments.

Talk about the with statement next time

1.5 Threading for animation

It is useful to start a simulation as a background process, which is running while you keep calculating in your Jupyter notebook. This can be achieved by setting up a thread.

A computer program is a collection of instructions. A process is the execution of those instructions. A thread is a subset of the process. A process can have one or more threads. The `Thread` function of the `threading` module can start a process in background, such that your Jupyter notbook is not blocked for the specific time the process is executed. We will talk about how to use this module later in this section.

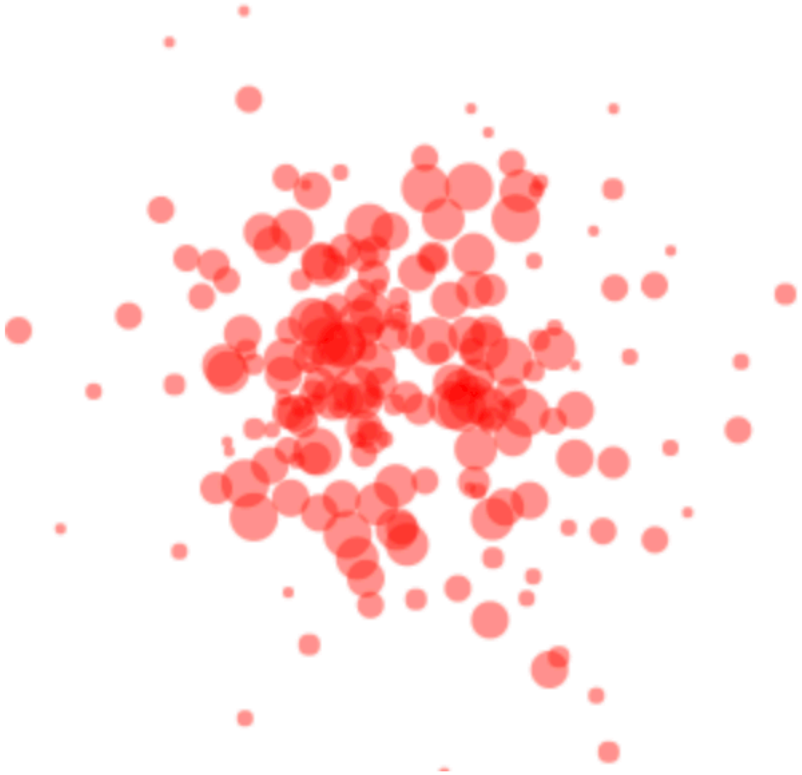
To setup such a background process you first need to setup a function that should be executed as a thread. This is the updating and drawing of the colloidal particles. The `Thread()` function of the `threading` module is setting up everything for you and assigning that to the variable `simulation`. The `target=draw` statement thereby points the thread to take the right function. Once you start the thread with `simulation.start()` the function draw is executed in background until its finished.

```
def draw():
    # do your drawing code here

simulation = Thread(target=draw)

simulation.start()
```

That's all you need. So lets wrap all our drawing before into a draw function.

We create a new canvas here, even though we could use the one on the top. It's just nice to not have to scroll up.

```
Canvas(height=300, width=300)
```



One of the intruiging things of this type of threading is that all of the parameters of the Colloids may still be changed on the fly while the process is running. So lets just reset the particle positions to the origin and see what is happening in the canvas.

Now that we have a nice way of simulating particle motion you can extend that a bit. Here are three additions you may want to make:

1) **Introduce boundary conditions:** This is a different way of keeping the particles inside the simulation box. They are just reflected by a boundary.

2) **Introduce a drift velocity:** Particles may not only move diffusively but also with a constant drift velocity in a certain direction. We want to introduce that feature to tackle a COVID-19 spreading.

3) **Introduce collisions:** To study the spreading of an infection, we have to introduce collisions between particles.

1.5.1 Threading and Stopping

Threading is nice, but gets easily confusing if running a thread multiple times without closing and stopping. You will end up with many threads that rush around somewhere in memory, which is of course not nice. Here is a small code snippet, you may want to consider, when you running your tasks in threads. It helps you to end a thread properly.

The `self._stop_event.set()` method is used to set the internal flag of a threading.Event object to true.

In the context of the `StoppableThread` class, `self._stop_event` is a threading.Event object that is used to signal the thread to stop executing.

When `self._stop_event.set()` is called, it sets the internal flag of the Event object to true. The thread checks this flag in its `run` method with `self._stop_event.is_set()`. If the flag is true, the thread stops executing.

Here's a simplified explanation of how it works:

1. Initially, the `self._stop_event` flag is false.
2. When the `stop` method is called, it sets the `self._stop_event` flag to true using `self._stop_event.set()`.
3. In the `run` method, the thread continually checks whether `self._stop_event.is_set()` is true. If it is, the thread stops executing. If it's not, the thread continues to execute.

This is a common pattern for stopping a thread in Python because Python does not provide a built-in way to stop threads directly.

```
Doing something
Doing something
Doing something
Doing something
Doing something
```

1.6 Animations with Matplotlib

While we have done all animations before with the help of the `ipywidgets` module, it is also possible to use the `matplotlib` module as well. There are several options,e.g., precalculating and displaying the animation. Yet, we want to use is the direct drawing. The code below demonstrates this option for 400 particles.