The N-Vortex Problem

Vortices appear in numerous places in nature, such as in tornadoes, hurricanes, in the wake of aeroplanes, or even in your kitchen sink. It is useful and interesting to study the interaction of simplified vortices (which we call "point vortices") in a plane, as these behaviours can help us to predict analogous situations which arise in the real world.

A point vortex simply rotates everything around it uniformly. The closer you are to a point vortex, the stronger the rotation. Associated with each point vortex is a measure of its 'strength' which we call circulation.

N vortices in a plane satisfy the following differential equations:

$$\frac{dx_i}{dt} = -\frac{1}{2\pi}\sum_{j\neq i}^{N}\frac{\Gamma_j(y_i - y_j)}{d_{ij}^2}$$
$$\frac{dy_i}{dt} = \frac{1}{2\pi}\sum_{j\neq i}^{N}\frac{\Gamma_j(x_i - x_j)}{d_{ij}^2}$$

where $i = 1, \ldots, N$, $d_{ij}^2 := (x_i - x_j)^2 + (y_i - y_j)^2$ (the distance squared) and Γ_i is the circulation.

Activity 1

It is possible to solve the 2-Vortex problem by hand, using techniques you know already. Give this a go!

Specifically, you are able to find a relationship between x_i and y_i, which describes the path the vortex will follow.

1. Write out the differential equations we have in this case. Our initial conditions will be the initial positions of each vortex ($(a_1, b_1), (a_2, b_2)$):
   $x_1 = a_1$
   $y_1 = b_1$
   $x_2 = a_2$
   $y_2 = b_2$
2. Try to spot patterns in the form of each differential equation, and use these to solve the equations.

Once you have a solution, think about what this tells you about the paths of the vortices in different situations. In particular, consider the following cases:

- $\Gamma_1 = -\Gamma_2$
- $\Gamma_1 = \Gamma_2$
- $\Gamma_1 \neq \Gamma_2$

Activity 2

It turns out that the 2-Vortex problem is the most complicated N-Vortex problem you can solve by hand. For anything else, you actually have to *numerically* solve the problem, using things like computer simulations. This is precisely what we are going to do next.

I have provided you all with a barebones simulation. Launch it, and try out the controls, just to get a feel for it.

Key	Action
`MOUSE LEFT CLICK`	Create a negative circulation vortex.
`MOUSE RIGHT CLICK`	Create a positive circulation vortex.

Key	Action
`SPACE`	Start/ stop the simulation.
`S`	Change the circulation (enter the new value and then press `ENTER`.)
`R`	Reset the simulation.

You will notice however that nothing happens when you start the simulation. This is because all the vortex behaviour has been removed from the code. Your job is to fix this.

You will only have to work inside `vortex.py`. The two functions that handle the behaviour are `getVelocity` and `moveVortex`. Using your mathematical knowledge of the N-Vortex problem, fill these functions and the make the simulation start working again.

With any simulation, to ensure it runs smoothly, we want to limit how many computations we do, and also make the computations as efficient as possible. Whilst the latter is the ideal approach, it does make the code less readable and straightforward, so we will accommodate the former.

In `moveVortex`, we want to compute the velocity of a given vortex caused by the surrounding vortices, and move the position accordingly. It would be slow to do this entire computation every time we want to move only a small distance, so instead, this function should do the computation once, and then move with the same velocity a number of times (equal to the `timeStep` specified). The other parameter in the function, `vortexArray`, is a list of all the vortices currently in the simulation.

In `getVelocity`, we simply want to work out what velocity any object would have at a specified point in the plane, due to the rotation of the vortex in question.

Activity 3

You finally have a working simulation, so it is now time to mess around with your vortex playground!

First, check the cases that we saw in Activity 1 for the 2-Vortex problem, to see if our predictions were correct (and verify the maths has indeed not lied to us).

Now, we have a variety of vortex setups for you to observe, and experiment with. Try out each, and note down anything you find interesting about the behaviour.

The simulation we have produced gives us a good indication of vortex dynamics, but it is not always the most accurate (the underlying implementation is not particularly efficient, and in order to guarantee a smooth framerate, the underlying code assumes that the velocity of a vortex stays the same for longer than is reasonable). For this reason, the simulation comes coupled with an "accurate mode", toggled by `A`, which runs more slowly, but makes the simulation far more precise.

Some of the situations below will only work as intended in accurate mode. These have been starred. It may be useful to also try accurate mode in the other situations, as an extra confirmation that the observed dynamics are indeed correct.

Leapfrogging

Flower*

Almost chaotic*

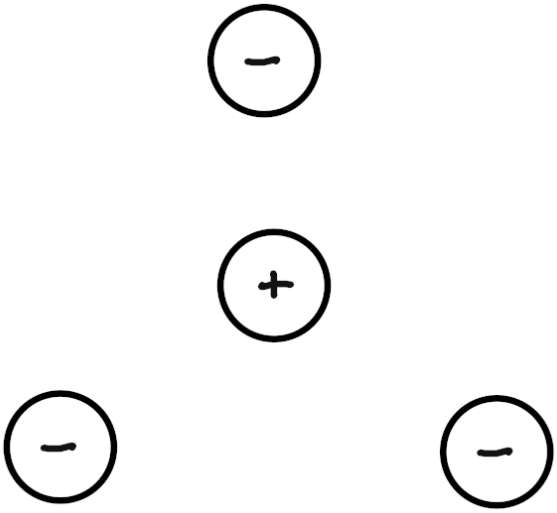
Partner swapping

Leapfrogging *and* partner swapping

Meta vortices

You'll have realised by now that sometimes collections of vortices can have interesting local dynamics, but at a distance, they behave exactly like a singular vortex. We can call these meta vortices. You've already seen the following meta vortex:

What about the following?



Try out these meta vortices with previous setups to see if they do behave exactly as you would expect.

Are these interactions always stable, or do they sometimes break down?

Chaos!

Place many vortices of the same circulation on the plane. Periodically add a singular vortex of the opposite circulation, and see how it behaves.

How many of the previous patterns can you see arising in this larger, far more chaotic system?

Free reign

Those are all the situations I had in mind for you to explore.

At this point, if you have any ideas for new or different situations that you think would result in intriguing dynamics, feel free to continue playing around with the simulation. You could even present these to the rest of the class!

Finished?

Let us know if you reach this point and are looking for something further to do. We will talk through one of the below, and give you the relevant resources.

We have some extensions in mind:

- Look at the code in `main.py` (ignore any references to `pygame` and drawing, since these are not relevant). The current implementation for movement is straightforward, but not particularly efficient. Look up the Runge-Kutta method, and see if you can implement it to make the simulation more efficient.
- Haskell and proving mathematically that functions do indeed act as intended, primarily using induction.
- Aliquot sequences.