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:

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

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 (when $t = 0$):

   $(x_1, y_1) = (a_1, b_1)$
   $(x_2, y_2) = (a_2, b_2)$

2. Try to spot patterns in the form of each differential equation, and use these to solve the equations.

3. Don't worry about necessarily finding the constants explicitly in your final equations, or about fully simplifying and factorising them.

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 bare-bones simulation. To start the program, you can go to `main.py` and hit the run button in the top right corner.

Launch the simulation and try out the controls to get a feel for it.

Key	Action
`MOUSE LEFT CLICK`	Create a negative circulation vortex.
`MOUSE RIGHT CLICK`	Create a positive circulation vortex.
`SPACE`	Start/ stop the simulation.
`S`	Change the circulation (enter the new value and then press `ENTER`.)
`R`	Reset the simulation.

You will notice 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`. This contains a `Vortex` class, with each vortex having the following properties:

- `pos`, a tuple `(x,y)` specifying its location
- `circulation`, the circulation of the vortex
- `velocity`, a tuple `(v_x,v_y)` specifying the current velocity of the vortex

The functions you need to complete, which handle the behaviour, are `getInducedVelocity`, `computeVelocity` and `move`. You can ignore everything else. In each loop of the simulation, we run `computeVelocity` for each vortex, and then `move` for each vortex.

Using your mathematical knowledge of the N-Vortex problem, fill these functions in and make the simulation start working again. Below are some important things to note about each function:
`getInducedVelocity`:

- `otherPos` is a tuple `(a,b)` specifying the point where you should compute the velocity this vortex induces.
- This function should return a tuple.

`computeVelocity`:

- `vortexArray` is a list containing all the vortex objects present in the simulation at any given time.
- This function should not return anything.

`move`:

- `timePeriod` (a.k.a. the *time step*) is the duration over which we assume the velocity stays the same, and over which the vortex moves each time the method runs.
- This function should not return anything.

As a final note, for those who care deeply about types (if this is not you, then do ignore this), note that `pos` and `velocity` being tuples is not *strictly* required. Any array-like structure is fine to use, so long as the code works.

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).

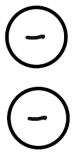
Now, we have a variety of vortex setups for you to observe and experiment with. Try out each, and note down anything you find interesting about the behaviour.

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 frame rate, 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*

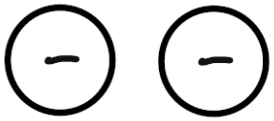
Almost chaotic

Partner swapping

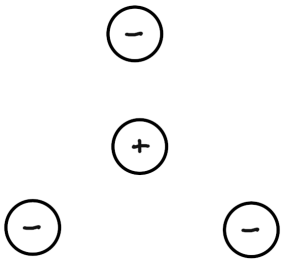
Leapfrogging *and* partner swapping

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

You have now seen all the situations I came across that I thought were worth sharing.

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. Show us if you find anything cool!

((You could even present these to the rest of the class!))

Finished?

Let us know if you reach this point (and well done for making it this far!)

We have one main extension you can work through at this point: turtles, L-systems and fractals!

An L-system simply consists of just a set of characters, and some replacement rules for each character that turn one string into another (generally more interesting) string.

Here is an example:
Characters: $\{a, b\}$
Rules:
$a \mapsto b$
$b \mapsto aa$

System (applying the rules repeatedly):
$a \rightsquigarrow b \rightsquigarrow aa \rightsquigarrow bb \rightsquigarrow \cdots$

With only a few characters, a couple of simple rules, and a few iterations of the associated L-system, you can generate some incredibly long strings which also have interesting patterns or structure to them. What makes these highly intriguing though is what happens when you take a drawing turtle, and associate every character in the output string with a turtle instruction. It turns out that in many cases the output string gives rise to a growing fractal-like shape.

Your job is to write the code for an L-system and drawing turtle, and investigate the outputs of different L-systems. There are three functions in the `L_system` class for you to fill in.

You may find it useful to refer to https://docs.python.org/3/library/turtle.html.

`add_rule` should incorporate the provided rule into the system by creating an entry in the `self.rules` dictionary with key `char` and value `string`
`update` should apply any rules in the system to each character in the current string to build the new string (which replaces the current string).
`print` should move the turtle using `forward`, `left` and `right` based on the presence of certain characters in the current string. The standard associations are listed below.

Symbol	Meaning
`+`	Rotate right and move forwards.
`-`	Rotate left and move forwards.

Once the functions have been filled in, the code should run, and produce a fractal-esque tree. You can now change the system to `square_snowflake` as well, or even search for other interesting systems online (e.g. on Wikipedia), and observe how they evolve!

If you somehow finish this exercise too, the two final, *final* extensions are as follows:

- Add some more turtle functions to `print`. One common addition is the following: `[` and `]`, associated with saving the current position (to a stack), and loading the saved position respectively.
- 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.