Lorenz Attractor

If you heat a shallow layer of fluid (for instance, when heating a pan of oil on the hob), the motion of an individual particle within the fluid is dictated by a system of differential equations determining the position (x, y, z) at time t. Lorenz, who was studying the mind-bogglingly complex behaviour of weather systems ('hot air rises', it turns out, is a slight over-simplification), reduced the problem to that of a closed cuboidal container heated uniformly from below and cooled uniformly from above. The resulting solution is one of the classic examples of what has come to be known as 'strange attractors' – instead of approaching a fixed position as t increases, particles exhibit a curious not-quite-periodic behaviour. But to fully appreciate it, you have to see it in 3D…

3D p5 sketches

3D graphics are nothing more than projections from 3D into 2D, which just means a bunch of matrix transformations behind the scenes. Thankfully, p5 takes care of all of that, and all we need to do to tell it we want to work in 3D rather than 2D is include *WEBGL* as a third argument in the *createCanvas* function:

```
function setup() {
  createCanvas(400, 400, WEBGL);
}

function draw() {
  background(220);
  orbitControl();
  normalMaterial();
  box(100);
}
```

So that we can see that we really are dealing with a 3D rendering, I've included a call to the *orbitControl* function in *draw*, allowing interaction with the 3D model with the mouse. Give it a try – when you first run it, you'll just see a square in the middle of the screen, but if you try dragging the canvas with the mouse, it'll turn around and reveal itself to be a cube.
normalMaterial() just colours the faces prettily. Try *noFill()* to see the 'wire-frame' version, with just edges shown.

Axes

Drawing a box is all very well, and there are a few other so-called 'primitives' that you can create for yourself if you check out the p5 references here: https://p5js.org/examples/3d-geometries.html, but we want a bit more precision for our project, so let's start by drawing in a set of axes.

```
function drawAxes(){
  let d = 200;
  stroke("red");
  line(-d, 0, 0, d, 0, 0);
  stroke("green");
  line(0, -d, 0, 0, d, 0);
  stroke("blue");
  line(0, 0, -d, 0, 0, d);
}
```

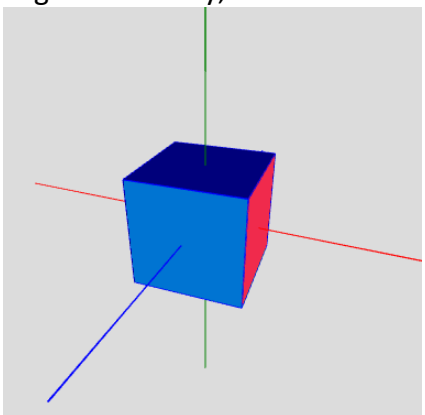
Don't forget to include this line in your *draw* function:

```
function draw() {
  background(220);
  orbitControl();
  normalMaterial();
  drawAxes();
  box(100);
}
```

In 2D, lines are defined using *line*(x_1, y_1, x_2, y_2). In 3D this generalises to *line*$(x_1, y_1, z_1, x_2, y_2, z_2)$.

Rotation

Mouse control is handy, but if we want to manipulate the position of objects within our 3D space programmatically, we should learn to rotate.



```
drawAxes();
rotateY(frameCount*0.01);
box(100);
```

If you try to use *rotate* function as in 2D you'll get an error. In 2D, the default centre of rotation is the origin, but in 3D we need a default *axis* of rotation as well. If we use *rotateZ* we'll actually get the same result as traditional rotation in 2D, since the axis of rotation is straight out of the screen.

By using the built-in *frameCount* variable I can set the total rotation to a slightly larger value each frame, causing a turning motion.

The Equations

The simplified system of equations boils down to this set of three coupled first order differential equations in x, y and z, where x represents convective flow, y the horizontal temperature distribution and z the vertical temperature distribution:

$$\frac{dx}{dt} = \sigma(y - x) \qquad \frac{dy}{dt} = x(\rho - z) - y \qquad \frac{dz}{dt} = xy - \beta z$$

The constants σ, ρ and β represent the ratio of fluid viscosity to thermal conductivity, the temperature difference between the top and bottom, and the ratio of box width to box height respectively. For more detail, a good article on this can be found here: https://science.howstuffworks.com/math-concepts/chaos-theory4.htm
In the original version of the Lorenz attractor, these numbers were set to: $\sigma = 10, \quad \rho = 28, \quad \beta = \frac{8}{3}$.

Representing the curve

Although eventually I want to view the whole curve, let's start by just making a point which moves in 3D space according to the equations. We need to define the variables x, y and z, along with dx, dy and dz, and update them numerically by taking very small time steps dt.

```
let [x, y, z] = [1, 2, 3];
let [sigma, rho, beta] = [10, 28, 8/3];
let dx, dy, dz;
let dt = 0.01;

function lorenz(){
  let sf = 2;
  dx = sigma * (y - x);
  dy = x * (rho - z) - y;
  dz = x * y - beta * z;
  x += dx * dt;
  y += dy * dt;
  z += dz * dt;
  point(sf*x, sf*y, sf*z);
}

function draw() {
  background(220);
  orbitControl();
  drawAxes();
  stroke(0);
  noFill();
  box(200);
  lorenz();
}
```

I define (arbitrary) starting values for x, y and z. I use Lorenz's values for σ, ρ and β. I declare the variables dx, dy and dz, and set a time step dt.

The *lorenz* function updates the derivatives according to the equations, then changes the variables accordingly. Finally, it represents a point on screen (with a scale factor to scale nicely).

The *draw* function has also been updated to make a call to *lorenz*, and to make the box a black transparent wire frame.

Speeding things up

An alternative

The alternative to reducing 'time resolution' is having the process of updating values of x, y and z take place multiple times per frame. It's not very computationally expensive to update their values, so let's calculate plenty of values but only display the point after every 10th iteration:

```
function lorenz(n=10){
  let sf = 2;
  for (let i=0; i<n; i++){
    dx = sigma * (y - x);
    dy = x * (rho - z) - y;
    dz = x * y - beta * z;
    x += dx * dt;
    y += dy * dt;
    z += dz * dt;
  }
  point(sf*x, sf*y, sf*z);
}
```

This is the same code but wrapped in a loop so that the calculations occur ten times before representing the point on screen. Since this function is called once in the *draw* loop, we will get ten steps for every frame.

You might want to experiment with different values for *n*, or even make it depend on the time-step used, so you can see the effect of different 'resolutions' but preserve the same speed.

Making the curve

While watching the point move through time is fun, it would be nice to see the characteristic 'butterfly curve' that comes up when you google 'Lorenz attractor'. To do this, we'll need to store the coordinates of points we reach as we go along. Let's make a slightly modified version of our previous *lorenz* function that generates the first *n* points and stores them all in an array.

```
function lorenzPoints(x, y, z, n){
  let pts = [];
  let sf = 2;
  for (let i=0; i<n; i++){
    dx = sigma * (y - x);
    dy = x * (rho - z) - y;
    dz = x * y - beta * z;
    x += dx * dt;
    y += dy * dt;
    z += dz * dt;
    pts.push(createVector(sf*x, sf*y, sf*z));
  }
  return pts;
}
```

I copied and pasted my previous function, then changed a few things – the main bit that does the calculating of points is unchanged, but now that loop will iterate through *n* times, and return an array of every point.

Note that, instead of using the global *x*, *y* and *z* variables, I'm taking values as inputs. This is so that this function won't interfere with the other one – I'd like to see both potentially, with the moving point 'drawing' the curve.

Showing the curve

I now have a function that will give me the first *n* points of the Lorenz curve. I want to display it, but ideally only up to wherever my moving point is. I could run this function every draw loop, up to a larger and larger value of *n*, but it would be much more efficient to create an array just once, in *setup*, and then draw a curve using the first *n* points during *draw*.

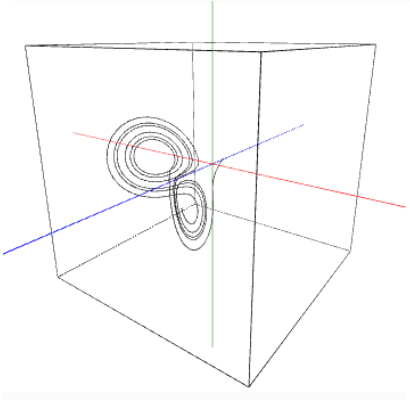
```
let [x, y, z] = [15, -10, 12];
let [sigma, rho, beta] = [10, 28, 8/3];
let dx, dy, dz;
let dt = 0.001;
let n = 0;
let pts = [];

function setup() {
  createCanvas(400, 400, WEBGL);
  pts = lorenzPoints(x, y, z, 10**6);
}
```

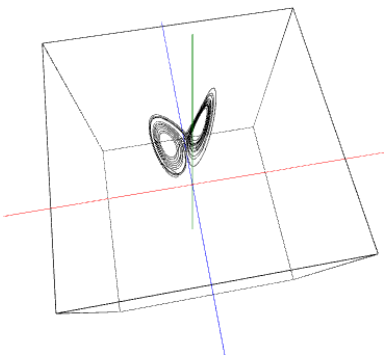
At the start, along with the other variables being declared, I'll make an *n* to represent how many iterations have taken place, and the *pts* array which will be used to populate the first, say, million points of the curve.
I then use my new *lorenzPoints* function within *setup* to populate this array.
Note: I've also changed x, y and z after some playing around, since it gives a better shape.

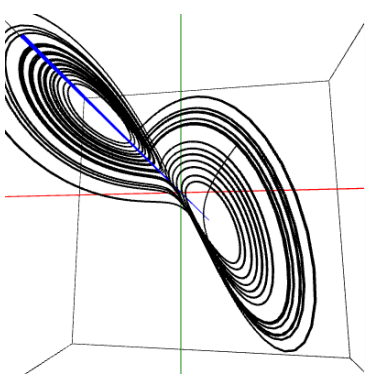
```
function draw() {
  background(220);
  orbitControl();
  drawAxes();
  stroke(0);
  noFill();
  box(200);
  lorenz(10);
  n+=10;
  beginShape();
  for (let i=0; i<n; i++){
    vertex(pts[i].x, pts[i].y, pts[i].z);
  }
  endShape();
}
```

Within *draw*, in addition to displaying a point with *lorenz*, I increase the value of *n* (going up in 10s to match the number of iterations that take place each frame), then use *beginShape* and *endShape* to create a curve from the first *n* points of my million-point array.

Remember you can rotate the canvas in 3D with the mouse, and even zoom with a scroll-wheel.







Improving efficiency

You may find that this is still rather slow – we're looping through ever longer stretches of a very large array in every draw loop, after all. We can make use of the same trick we used in the original *lorenz* function here, and have the original points array only update every few steps along.

```
function lorenzPoints(x, y, z, n){
  let pts = [];
  let sf = 2;
  for (let i=0; i<n; i++){
    for (let j=0; j<10; j++){
      dx = sigma * (y - x);
      dy = x * (rho - z) - y;
      dz = x * y - beta * z;
      x += dx * dt;
      y += dy * dt;
      z += dz * dt;
    }
    pts.push(createVector(sf*x, sf*y, sf*z));
  }
  return pts;
}
```

I've just hard-coded a loop of size 10 here, since that number seemed to work alright before. We'll also have to amend the increment by which *n* increases if both functions are making 10 steps per saved / displayed point:

```
box(200);
lorenz(10);
n++;
strokeWeight(0.5);
beginShape();
```

Note: I've also changed the *strokeWeight* so that the line is thinner and easier to make out. Although we may observe what appear to be overlaps, the curve never repeats itself.

More of them!

One of the fascinating things about the Lorenz attractor is that it is chaotic, in the technical sense that a small perturbation in initial conditions causes significant deviation in long term behaviour. We can try to demonstrate this by creating a few different attractors and displaying them all at once…

```
function draw() {
  background(255);
  orbitControl();
  drawAxes();
  stroke(0);
  noFill();
  box(200);
  lorenz(10);
  n++;
  strokeWeight(0.5);
  drawCurve(pts, n);
  stroke("red");
  drawCurve(pts2, n);
  stroke("green");
  drawCurve(pts3, n);
}

function drawCurve(pts, n){
  beginShape();
  for (let i=0; i<n; i++){
    vertex(pts[i].x, pts[i].y, pts[i].z);
  }
  endShape();
}
```

First of all, if we're going to draw a curve for more than one version, we should make it its own function, so pull it out of *draw* and change the reference to a function call with *pts* and *n* as arguments. I've also put in different colours so we can distinguish between them.

Then we just need to pop back up to *setup* and make a couple of similar, but not quite identical, sets of points:

```
let pts, pts2, pts3;

function setup() {
  createCanvas(400, 400, WEBGL);
  pts = lorenzPoints(x, y, z, 10**6);
  pts2 = lorenzPoints(x-0.1, y, z, 10**6);
  pts3 = lorenzPoints(x+0.1, y, z, 10**6);
}
```

You can also draw in planes by adding these lines to the bottom of *draw* (width and height are the parameters – use the rotation functions to position them) which aids visualisation I think:

```
drawCurve(pts3, n);
fill(220, 220, 220, 50);
rotateX(PI/2);
plane(200, 200);
rotateY(PI/2);
plane(200,200);
rotateX(PI/2);
plane(200, 200);
```