Simulating Life

Computer simulations are a vital part of analysing the impact of changes to populations. As with many complex or chaotic systems, while a precise analytical answer may not be possible, multiple numerical simulations can lend insight without the necessity for slow, costly trials. The basic building blocks of any such simulation are objects which exhibit the basic properties of living things. In this introductory project we'll make basic objects which can then be developed further and modified to form the backbone of a wide range of population simulations.

What is life?

If you study Biology, you may already have a neat answer, but it's surprisingly difficult to pin down exactly what counts as a fundamental property of living things. This may boil down to the fact that, although most people would agree that dogs are alive and rocks are not, for most practical purposes you could argue that trees have more in common with rocks than with dogs (other than the bark). Fortunately for our simulation, we get to decide which features of life are important for our code, and which we can conveniently ignore. But the fairly standard model used by biologists is a good starting point:

Living things:

- Grow
- Move
- Reproduce
- Eat
- Respond to stimuli

These things can all be simulated pretty effectively. Let's start with the basics: I want to simulate a basic looking creature which I will call a *Cell*. For starters, let's give it a position, a colour (so we can distinguish it from others) and a way of being displayed on the screen.

Making cells appear

```
class Cell{
  constructor(){
    this.p = createVector(random(width), random(height));
    this.color = color(0, int(random(1, 10))*25, int(random(1, 10))*25);
  }
  display(){
    noStroke();
    fill(this.color);
    ellipse(this.p.x, this.p.y, 10, 30);
    ellipse(this.p.x, this.p.y, 30, 10);
    circle(this.p.x, this.p.y, 20);
  }
}
```

For now, our cells require no parameters to construct: they appear at a random position on the screen, and are given a random colour. Note that the random numbers I'm generating for the colour are multiples of 25, so different colours are more distinct.

The display code uses two overlapping ellipses in addition to a circle to generate a more distinctive shape.

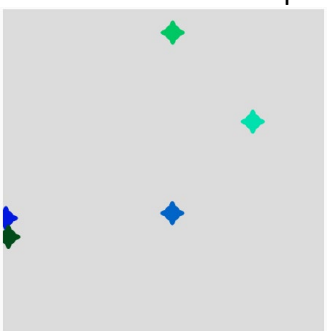
We now need to create some cells to display:

```
let cells = [];
let n = 5;

function setup() {
  createCanvas(400, 400);
  for (let i=0; i<n; i++){
    cells.push(new Cell());
  }
}
```

And use the *draw* loop to display them all:

```
function draw() {
  background(220);
  for (let cell of cells){
    cell.display();
  }
}
```



Making the cells grow

It has been wisely said that death is an essential part of life, and if we want our cells to mimic real populations we should build in an expiry date. Perhaps we can make the additional features like their spikes only appear after a certain age. Since we'll be updating a lot more than just their age, we should make a dedicated *update* method that is run every time the *draw* loop runs:

```
1  class Cell{
2    constructor(){
3      this.p = createVector(random(width), random(height));
4      this.color = color(0, int(random(1, 10))*25, int(random(1, 10))*25);
5      this.age = 0;
6      this.lifespan = 100;
7      this.alive = true;
8    }
9    update(){
10     this.age += 0.1;
11     if (this.age > this.lifespan){
12       this.alive = false;
13     }
14   }
15   display(){
16     noStroke();
17     fill(this.color);
18     let size = map(this.age, 0, this.lifespan, 5, 20);
19     let spike = map(this.age, this.lifespan/2, this.lifespan, 0, 10);
20     if (this.age < this.lifespan/2){spike = 0}
21     ellipse(this.p.x, this.p.y, size-spike, size+spike);
22     ellipse(this.p.x, this.p.y, size+spike, size-spike);
23     circle(this.p.x, this.p.y, size);
24   }
25 }
```

Lines 5 to 7 give our cells an expiry date. Note that, in future populations, something other than old age may kill a cell, so I'm making the attribute *alive* separately.

The *update* method is new: we simply increase the age gradually until we reach the lifespan.

The *display* method has also had some changes: the size of the main circle now depends on age, as do the spikes, which don't appear at all until middle-age.

Note that, until we do something about dead cells, these are simply going to take death on the chin and just keep on growing. We could make them only grow if alive, or turn grey or something, but another simple solution is just to check for dead cells in the *draw* loop and remove them as they occur.

```
function draw() {
  background(220);
  for (let i=0; i<cells.length; i++){
    cells[i].update();
    cells[i].display();
    if (!cells[i].alive){
      cells.splice(i, 1);
    }
  }
}
```

Unlike Python's *remove* method, JavaScript requires the index to remove an element. We could find the index for a given object, but it's probably simpler to change our *for* loop back into the more standard version, with an index. In place of *cell*, write *cells[i]* and the rest should all work fine. Note that ! is the JavaScript character for 'not' (just like != is 'not equal to'), and the *splice* method removes a section of the array for us, starting at index *i* and removing *1* element. Note that if the length of *cells* changes during the loop, *cells.length* will also change, so the loop works fine.

Making cells move

Motion can be as simple or as complicated as you like, and you are encouraged to play around with more sophisticated motion models here, including incorporating acceleration, or even avoidance of other cells. For now, we'll just have them move around the screen, bouncing off the sides if they get too close.

```
class Cell{
  constructor(){
    this.p = createVector(random(width), random(height));
    this.v = p5.Vector.random2D().mult(0.5);
```

We start by giving the cell a velocity vector in the *constructor* method. I'm using the *random2D* method built into the *p5.Vector* class, which yields a random unit vector, combined with multiplication to give a speed of 0.5 pixels per frame.

```
  update(){
    this.age += 0.1;
    if (this.age > this.lifespan){
      this.alive = false;
    }
    this.v.rotate(random(-PI/12, PI/12));
    this.p.add(this.v);
    if (abs(this.p.x - width/2) > width/2){
      this.v.x *= -1;
      this.p.x += this.v.x;
    }
    if (abs(this.p.y - height/2) > height/2){
      this.v.y *= -1;
      this.p.y += this.v.y;
    }
  }
```

In the *update* method, I first allow the velocity to change direction by up to 15 degrees in one direction or the other, then I add the current velocity to the position. The rest of the code takes care of reversing the relevant component of the velocity if the cell reaches the boundary, and also shunts it back one frame's worth to ensure we don't get glitches from a cell over-shooting the edge of the screen (and permanently changing its velocity sign without ever getting back into view properly!)

Making cells reproduce

If we're dealing with single-celled organisms, we simplify this problem quite a bit, since they can quite happily reproduce all on their own. Like yeast, we can have them 'bud' – generating a clone – provided conditions are right. For starters, let's just build in a 'fertility' value, and have cells above a certain age randomly reproduce according to that probability.

```
let cells = [];
let n = 5;
let fertility = 0.01;
```

The *fertility* variable should be declared as a global variable, so we can access it anywhere and change it readily.

```
    this.lifespan = 100;
    this.alive = true;
    this.reproducing = false;
  }
  update(){
    this.age += 0.1;
    if (this.age > this.lifespan/2){
      if (random() < fertility){
        this.reproducing = true;
      }
    }
    if (this.age > this.lifespan){
      this.alive = false;
    }
```

We should also include a marker of sorts in the *constructor* for a *Cell* object, so that we can easily detect cells that need to breed and generate new cells in the *draw* loop.

And in the *update* method, as well as checking to see if the cell is still alive, we can check to see if a) it's old enough and b) it's lucky enough to breed. If so, it toggles its *reproducing* marker.

```
    if (!cells[i].alive){
      cells.splice(i, 1);
    }
    if (cells[i].reproducing){
      cells.push(new Cell(cells[i]));
      cells[i].reproducing = false;
    }
  }
```

Now in the *draw* loop, we check each cell to see if it is flagged as *reproducing*. If so, we push a new cell into the array, and switch off the *reproducing* flag (so it doesn't automatically reproduce again every frame from then onwards).

Notice that I'm now passing an argument to *Cell*. This is because of the next improvement I plan to make to our *Cell* code…

Inheritance

There's a whole lot you can do with inheritance: we could simulate evolution by allowing not only the passing on of information from parent to child, but also allow for some random mutation. Fast parents could spawn even faster children, for instance, and parents with shorter lifespans would be less likely to breed. For now, we'll just focus on passing on basic properties to descendants.

```
class Cell{
  constructor(parent=null){
    if (parent != null){
      this.p = parent.p.copy();
      this.v = parent.v.copy();
      this.color = parent.color;
    }
    else{
      this.p = createVector(random(width), random(height));
      this.v = p5.Vector.random2D().mult(0.5);
      this.color = color(0, int(random(1, 10))*25, int(random(1, 10))*25);
    }
    this.age = 0;
    this.lifespan = 100;
    this.alive = true;
    this.reproducing = false;
  }
```

By making the *parent* argument optional, we can still generate new cells with no arguments (that happens at the start, when we create life from nothing). But if a cell does have a parent, its position and velocity (initially, at least), will be the same as those of its parent. They'll also share the same colour, so we may be able to track families.

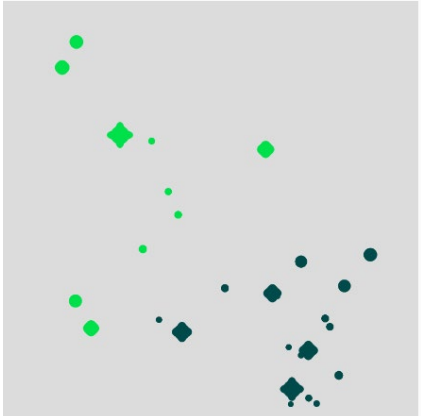
A word of warning: make sure you save your code – once you allow your creations to breed, you're starting off a chain reaction which grows exponentially. It's quite possible that your population, with no constraints, will grow so large and so fast that your browser crashes.

To limit this, it is probably sensible to build in a maximum population:

```
let cells = [];
let n = 2;
let maxN = 100;
let fertility = 0.01;
```

And, in the draw loop, only allow reproduction if we're not at max yet:

```
if (cells[i].reproducing && cells.length < maxN)
```



Next steps

An important element of an ecosystem is self-limiting. We don't want an arbitrary top-down rule against breeding, but what if cells could only clone themselves if there were enough space around them? We would need functionality within our *Cell* class which could check all other cells to see how close they are.