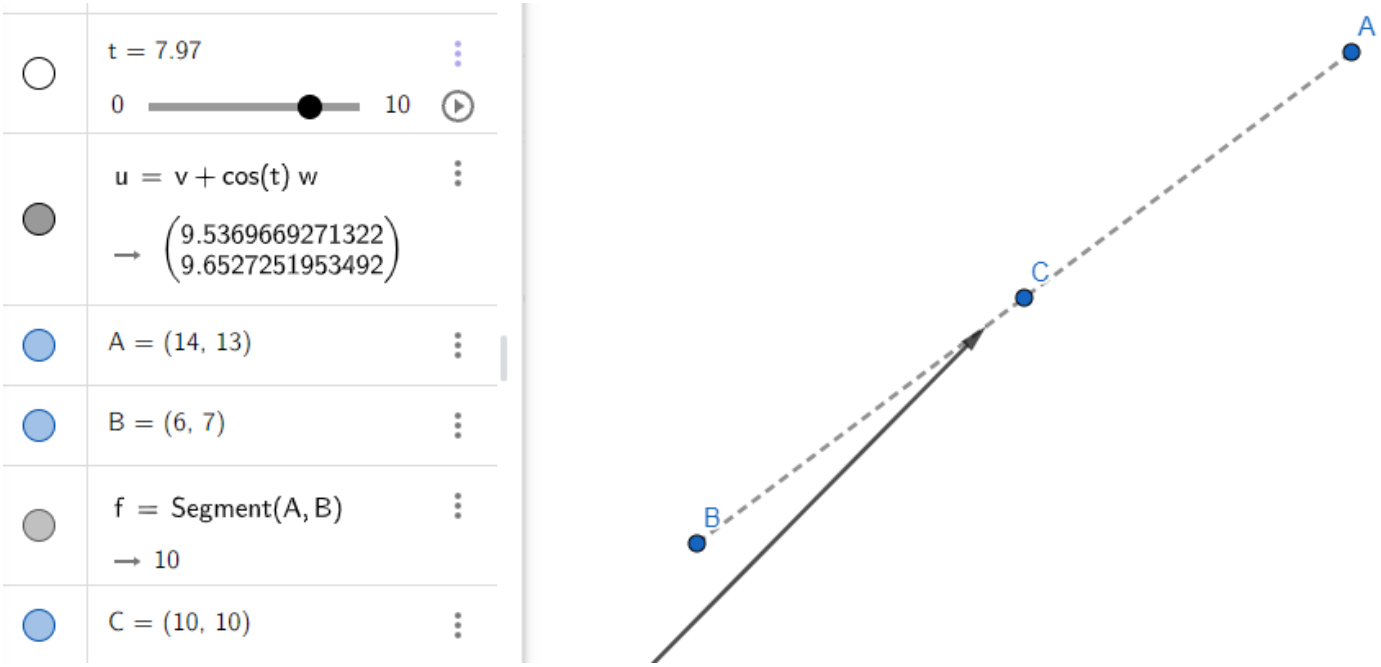
Oscillating Dots

Consider the behaviour of the following vector for different values of λ from -1 to 1:

$$\boldsymbol{r} = \begin{pmatrix}10\\10\end{pmatrix} + \lambda\begin{pmatrix}4\\3\end{pmatrix}$$

As λ varies between -1 and 1, the vector varies from $\begin{pmatrix}6\\7\end{pmatrix}$ to $\begin{pmatrix}14\\13\end{pmatrix}$.

If we make use of a trig function for the parameter λ, we will get a nice smooth transition between the two points:



We can use this concept to define motion for a particle in p5.
Let's start by defining a central point, and the vector from the centre to one extreme:

```
let t = 0;

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

function draw() {
  background(220);
  circle(width/2 + cos(t)*width/2, height/2 + cos(t)*0, 20);
  t += 0.05
}
```

Defining a global variable *t* will give us a time counter we can increment each frame, to use in our trig function.

I'm using $\begin{pmatrix}\frac{width}{2}\\\frac{height}{2}\end{pmatrix}$ as the central point, and $\begin{pmatrix}\frac{width}{2}\\0\end{pmatrix}$ as the vector to an end point.

We should see a dot oscillating wildly across the screen from far right to far left.

Now let's make a second dot: one that oscillates from top to bottom:

```
function draw() {
  background(220);
  circle(width/2 + cos(t)*width/2, height/2 + cos(t)*0, 20);
  circle(width/2 + cos(t + PI/2)*0, height/2 + cos(t + PI/2)*height/2, 20);
  t += 0.05
}
```

You may notice that, in addition to changing the end point vector to $\begin{pmatrix} 0 \\ \frac{height}{2} \end{pmatrix}$, I have also introduced a time offset, using $t + \frac{\pi}{2}$ instead of just t. This means the point will be $90°$ out of phase with the other, so it should reach its extreme point when the first dot is in the centre. *We could have used* $\sin t$ *for the same effect, but we're not going to stop at just two dots, so I want an approach which scales for many dots.*

Revisiting Classes

I want to be able to generate many similar objects: oscillating dots. Although if I keep things simple enough I could make do with loops, as our projects get increasingly complex it will be really helpful to have the functionality of a **class**.

Recall that writing a class defines the attributes and methods (properties and functionality) of an object that you create. Think of your class code as representing a blueprint that explains to the computer what it means to be, in this case, an oscillating dot.

Another way to think of classes are the 'cookie cutters': they are not in themselves objects, but once you have a class you can use them to create multiple instances of your object.

Everything you learnt about classes in Python pretty much applies to JavaScript as well, with a few syntax changes.

Compare these examples:

JavaScript:

```
class Square{
  constructor(x, y, w){
    this.x = x;
    this.y = y;
    this.w = w;
  }
  area(){
    return this.w**2
  }
  display(color){
    fill(color);
    rect(this.x, this.y, this.w, this.w);
  }
}

let s = new Square(40, 50, 10);
```

Python:

```
class Square:
  def __init__(x, y, w):
    self.x = x
    self.y = y
    self.w = w

  def area():
    return self.w**2

  def display(color):
    fill(color)
    rect(this.x, this.y, this.w, this.w)

s = Square(40, 50, 10)
```

Note that, in place of *__init__*, p5 uses the *constructor* function.
Where Python employs the prefix *self.*, p5 uses *this.*

Planning the Dot Class

Before diving in to writing our code, we should consider what is needed. Anything which is common to all dots can be coded into the class definition, and anything which will be unique to a particular dot should be made a parameter of the constructor function so the user can define it when they create the object.

It is often also helpful to think separately about what *attributes* and what *methods* will be required. What will our dot *have* and what will it *do*?

Possible attributes:	**Possible methods:**
Central point of oscillation	Update (update position based on the time that has passed)
Vector from centre to end	Display (show an image of the dot on the screen)
Current position	
Colour	
Size	
Time offset	

Not all the attributes have to be modifiable when creating the object. It often helps to work out what will be constant for all such instances, and hard-code it to avoid unnecessary duplication when using the objects.

Another useful thing to bear in mind is how you intend to create and use the objects. In this case, perhaps generating an array of *Dot* objects, then looping through it each time *draw* executes to update and then display each one. Often, programmers will write in the code that they want, and create the objects to fit the requirements afterwards:

```
16  let t = 0;
17  let dots = [];
18
19  function setup() {
20    createCanvas(400, 400);
21    let n = 4;
22    for (let i=0; i<n; i++){
23      dots.push(new Dot(i*PI/n));
24    }
25  }
26
27  function draw() {
28    background(220);
29    for (let dot of dots){
30      dot.update(t);
31      dot.display();
32    }
33    t += 0.05
34  }
```

We declare an empty array in the global scope (not just inside *setup*) so we can access it from both *setup* and *draw*.

In *setup* we run a loop which adds new *Dot* objects to the array. Note that, unlike Python, p5 requires us to use the *new* keyword when creating a new object.

The offset I'm using will serve as both the time delay between the trig functions and also enable me to position them later.

In *draw*, we loop through the dots, updating them by passing to them the current value of *t* and then displaying them.

Before continuing, try to write a *Dot* class that would fit the demands of this code. Notice that it'll need an *offset* as a parameter to the *constructor* function, and it will need both *update* and *display* methods. Don't worry too much about the details yet!

Writing the Dot Class

```
class Dot{
  constructor(offset){
    this.offset = offset;
    this.vx = width/2 * cos(offset);
    this.vy = height/2 * sin(offset);
  }
  update(t){
    this.x = width/2 + cos(t + this.offset) * this.vx;
    this.y = height/2 + cos(t + this.offset) * this.vy;
  }
  display(){
    fill(map(cos(t+this.offset)**2, 1, 0, 120, 220))
    circle(this.x, this.y, 30);
  }
}
```

My *constructor* method takes only one parameter, and to keep things simple the *offset* variable does double duty: I'm using it not only as a time offset, but also to space the dots around a circle, by defining the vector from the centre to an end.

The *width/2* and *height/2* are interchangeable since I've made my canvas square. Essentially I'm using the equivalent of $r\cos\theta$ and $r\sin\theta$ to determine the end point for each dot's path. The values of θ, as defined in the loop on the previous page, will be equally spaced between 0 and π. *(Why not* 0 *and* 2π*? What would happen in that case?)*

The *update* function sets the x and y values for my dot using the special function we investigated at the beginning. The *display* function simply renders a circle in that position.

Even better if…

- Because of how we've defined our points, we can easily stretch things. Try changing the canvas width or height to get an elliptical illusion rather than a circular one:

- Try creating an animation where the number of dots doubles every cycle. In *draw*:

```
if (t > 2*PI && n < 16){
  t -= 2*PI
  n *= 2;
  dots = [];
  for (let i=0; i<n; i++){
    dots.push(new Dot(i*PI/n));
  }
}
```

Make n a global variable, initially equal to 1. Then insert this code into *draw* so that each time t increments by 2π, the value of n doubles (until it reaches 16 at least).

- In the *display* method, you could set the colour or the size of your dot to vary as it moves, perhaps by using the *map* function along with $\cos(t + offset)$.

```
display(){
  noStroke();
  let p = cos(t+this.offset)**2
  fill(map(p, 0, 1, 220, 120));
  circle(this.x, this.y, map(p, 0, 1, 20, 40));
}
```

p is the square of a cos function, so it'll be between 0 and 1, letting me colour and size the dots based on how close to the centre they are.