Tangent Fields

Many differential equations – even linear ones – can't readily be solved analytically (that is, we can't define a 'closed form' solution in terms of elementary functions such as sin, cos, tan, e and ln). But if we know the gradient function for any given point in terms of x and y, we can create a 'tangent field' – a tiny segment of a tangent to the curve at every point.

In practice, we define a small step length, h, and draw a line of fixed length with the appropriate gradient at every one of a set of lattice points, such as every point (a, b) for $a, b \in \mathbb{Z}$, or every point of the form $\left(\frac{a}{10}, \frac{b}{10}\right)$, within a given range. In p5, we could improve on this still further by colour-coding our tangents, then investigate different tangent fields by simply modifying the gradient function used.

The gradient function

Both the most important and the simplest part of this project, we start by defining a gradient function:

```
function gradient(x, y){
  return x*exp(-1*x) + sin(y);
}
```

This function is defined in the main *sketch.js* file, near the top, so we can more easily modify it later. As long as it includes functions that are recognized, it can be pretty much anything.

Setting up the grid

The tangent field should fill the entire screen, but often we want to zoom in on just a small section of the graph (eg if the family of solutions includes sine waves, we don't want detail to be lost just because the height only varies by 1 or 2 pixels. Since we'll need to switch back and forth between our (x, y) coordinates and the *i* and *j* indices of pixels, I'm going to create a couple of helper functions.

```
let k = 2;
let [xMin, xMax, yMin, yMax] = [-k, k, -k, k];

function ijToxy(i, j){
  let x = map(i, 0, width, xMin, xMax);
  let y = map(j, 0, height, yMax, yMin);
  return createVector(x, y);
}

function xyToij(x, y){
  let i = map(x, xMin, xMax, 0, width);
  let j = map(y, yMax, yMin, 0, height);
  return createVector(i, j);
}
```

I've used an extra variable, *k* since I imagine in most cases I'll want a 1:1 ratio between the axes, and the origin in the centre.

The helper functions take care of mapping either coordinates to pixel indices or vice versa, returning a vector in each case, which I can unpack for myself as needed within my code.

Note: it is possible to get the same effect by making use of the global translate *and* scale *functions before every call to draw, but I prefer this to a global behind-the-scenes coordinate transformation, not least because it also throws off things like* strokeWeight *when we mess with* scale.

Drawing a tangent

I need my tangent lines to be of fixed length. If I define a length (in pixels), I'll then need a way to easily draw them based on their position and gradient. Another helper function is in order here, I think.

```
function drawTangent(x, y){
  let m = gradient(x, y);
  let v = createVector(1, m);
  let sf = 10 / width * (xMax - xMin);
  v.normalize().mult(sf);
  let p1 = xyToij(x, y);
  let [i1, j1] = [p1.x, p1.y];
  let p2 = xyToij(x + v.x, y + v.y);
  let [i2, j2] = [p2.x, p2.y];
  line(i1, j1, i2, j2);
}
```

This function creates a vector to represent the direction of the function at the given point, but its length will depend on the gradient, so I reduce *v* to a unit vector and multiply by an appropriate scale factor to give it length of 10 pixels (note that I scale according to the width and the max and min x values).

Then I use my helper functions to convert the *x* and *y* values to *i* and *j* values, and finally draw a line between them.

The tangent field

Next up: iterate through a range of points and draw in the tangents. So that I can either run this once in *setup* or, potentially, modify the tangent field during *draw*, I'll make this a separate function too.

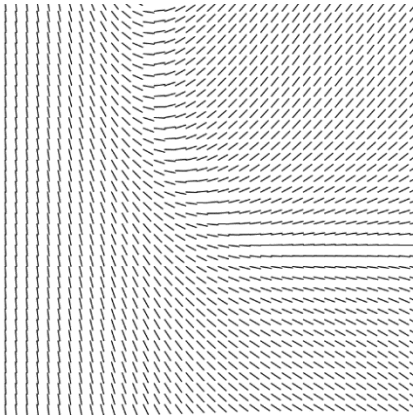
```
function drawTangentField(){
  for (let i=0; i<width; i+=10){
    for (let j=0; j<height; j+=10){
      let p = ijToxy(i, j);
      drawTangent(p.x, p.y);
    }
  }
}

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

function draw() {
  background(255);
  drawTangentField();
}
```

The *drawTangentField* function doesn't have to do much thanks to the *drawTangent* function we already defined. It iterates through values of *i* and *j* (going in steps of 10), generates the appropriate coordinates, and lets the helper function do the rest.

Finally, add a call to *drawTangentField()* in *draw()*.

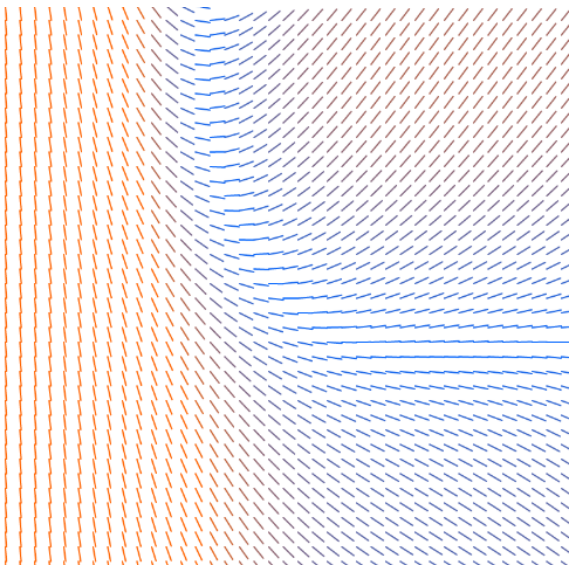


Colour coding

In addition to showing the slope at each point by tangent line, we could build in some 'conditional formatting' that gives each line a colour based on its gradient.

```
function drawTangent(x, y){
  let m = gradient(x, y);
  let v = createVector(1, m);
  let sf = 10 / width * (xMax - xMin);
  v.normalize().mult(sf);
  let p1 = xyToij(x, y);
  let [i1, j1] = [p1.x, p1.y];
  let p2 = xyToij(x + v.x, y + v.y);
  let [i2, j2] = [p2.x, p2.y];
  let r = map(abs(m), 0, 2, 0, 255);
  let b = 255 - r;
  stroke(color(r, 100, b));
  line(i1, j1, i2, j2);
}
```

Within the *drawTangent* function, I set the colour to be red if the size of the gradient is greater than 2, blue if it's zero, and something in between otherwise.



Visualising flow

This tangent field can be really helpful in visualising all possible solution curves for a given differential equation, but it doesn't actually show us any of them. It might be nice if we could choose a point for our starting value, and watch it move in the direction of the tangent lines. By throwing a bunch of points onto the sketch all at once, we would get a nice illustration of the overall flow, in the same way that the vector field view of air currents makes more sense if there's movement involved. First, I'll make a 'mover':

```
class Mover{
  constructor(i, j){
    this.p = ijToxy(i, j);
    this.live = true;
  }
  update(h=0.1){
    let m = gradient(this.p.x, this.p.y);
    this.v = createVector(h, h*m);
    this.p.add(this.v);
    if (this.p.x > xMax || this.p.x < xMin ||
        this.p.y > yMax || this.p.y < yMin){
      this.live = false;
    }
  }
  display(){
    let p = xyToij(this.p.x, this.p.y);
    stroke(255);
    fill(0);
    circle(p.x, p.y, 5);
  }
}
```

I will be able to create these movers by giving them pixel values (eg the coordinates of the mouse when clicked), and *this.live* will help me get rid of these objects once off-screen.

The *update* function has a default step length of 0.1 built in (note: this is in terms of x, not i) and I create a vector to determine motion based on *h* and the gradient. If this proves to be too fast or too slow, I can scale it here.

I also check to see if the point is on the screen, and if not, switch the *live* flag to *false*.

Finally, converting back from *x* and *y* to *i* and *j*, I draw the circle on the screen.

Creating movers

We can use p5's *mousePressed* function to create new movers, but we'll need an array to store them in, and we'll need *draw* to loop through, update, display and cull as needed.

```
let [xMin, xMax, yMin, yMax] = [-k, k, -k, k];
let movers = [];
```

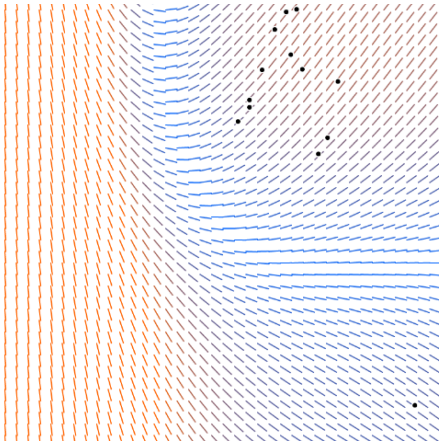
First, outside any functions, we declare *movers*, an empty array.

```
function mousePressed(){
  movers.push(new Mover(mouseX, mouseY));
  return false;
}
```

Next, we set up a *mousePressed* function (returning a *false* value so this works ok with all browsers and touchscreens), which adds a new mover wherever you click.

```
function draw() {
  background(255);
  drawTangentField();
  let newMovers = [];
  for (let mover of movers){
    mover.update();
    mover.display();
    if (mover.live){
      newMovers.push(mover);
    }
  }
  movers = newMovers;
}
```

And finally, in *draw*, we loop through the movers, updating and displaying each one. We also check to see if they're still *live*, and if so, they get added to the *newMovers* array, which will overwrite the *movers* array every frame.



Making movers automatically

It's great to be able to click on a spot and have the point follow the tangent field, but if we want to get a real sense of the flow through this field, we need new points being generated randomly for us.

```
  movers = newMovers;
  while (movers.length < 50){
    movers.push(new Mover(random(width), random(height)));
  }
}
```

At the end of the *draw* loop, I've added an extra bit that will add more movers to the array until it reaches a certain size (eg 50).

Take care when writing while *loops in p5 – if your code is set to update automatically, you may get stuck in an infinite loop before you finish coding it! Hit the 'stop' button, save, write the loop, check it, then 'play'!*

Viewing the curves

In addition to the big picture view that lots of moving dots gives us, it may be nice to see the actual curve traced out by a given point. Let's build in an optional *showTrace* argument that we can activate only for points generated by mouse clicks, so that those points display their path.

```
class Mover{
  constructor(i, j, showTrace=false){
    this.p = ijToxy(i, j);
    this.live = true;
    this.showTrace = showTrace;
    this.pts = [];
  }
```

First, *showTrace* is added as an optional argument to the *constructor*, and its default is *false*.
We save this attribute to be used later, and, if we do need it, we'll make an array of points to use.

```
...
update(h=0.1){
  let m = gradient(this.p.x, this.p.y);
  this.v = createVector(h, h*m);
  this.p.add(this.v);
  this.pts.push(this.p.copy());
```

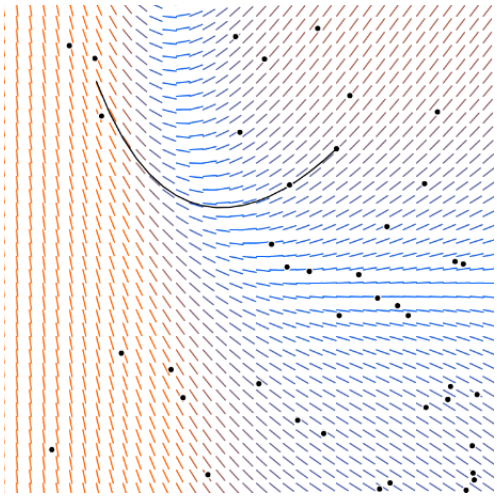
In *update*, whenever we generate a new position, we store a copy of that position vector in *this.pts*.

```
...
display(){
  let p = xyToij(this.p.x, this.p.y);
  stroke(255);
  fill(0);
  circle(p.x, p.y, 5);
  if (this.showTrace){
    noFill();
    stroke(0);
    beginShape();
    for (let pt of this.pts){
      let v = xyToij(pt.x, pt.y);
      vertex(v.x, v.y);
    }
    endShape();
  }
}
```

And in *display*, we use the *beginShape* and *endShape* commands while looping through the points stored, first converting from *x, y* to *i, j*, and then using as a vertex for the path we are drawing.

```
...
function mousePressed(){
  movers.push(new Mover(mouseX, mouseY, true));
  return false;
}
```

Lastly, update the *mousePressed* function so that any movers added to the screen by clicking automatically have the *showTrace* attribute.



Experiment

Now you have a tangent field generator, try different functions to see what kinds of properties each tangent field might have.