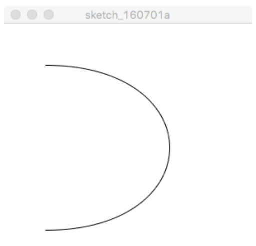
Worksheet 3 — basic curves and transformations

Drawing a Bézier curve

Open Processing.

Get a simple Bézier curve on the screen.

```
float x1=50,y1=50,x2=250,y2=50,x3=250,y3=250,x4=50,y4=250;
void setup() {
  size(300, 300) ;
}
void draw() {
  background(255);
  stroke(0);
  noFill();
  bezier(x1, y1, x2, y2, x3, y3, x4, y4);
}
```



Remember to press the run button at the end of each section in this worksheet, to see what the code does.

Find a point on the curve, using `bezierPoint()`, and draw a dot at that point

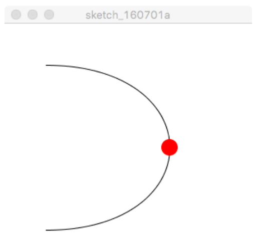
Add a variable at the top of the program to tell Processing at what parameter value along the curve you want the dot to be drawn. The parameter value should be between 0.0 (which is the value at the start of the curve) and 1.0 (the value the end of the curve):

```
float t = 0.5 ;
```

Add code to find the location of the point at that parameter value:

```
float x = bezierPoint(x1, x2, x3, x4, t);
float y = bezierPoint(y1, y2, y3, y4, t);
```

Notice that we handle `x` and `y` completely independently. There is one function call to find the x-coordinate, based on the x-coordinates of the Bézier control points; and there is a second function to call to find the y-coordinate, based on the y-coordinates of the Bézier control points.



Now draw a red circle at that point:

```
noStroke();
fill(255, 0, 0);
ellipse(x, y, 20, 20);
```

Animate the red circle moving along the curve

To do this you need to increment the parameter t on each time through the `draw()` function, and make sure that it is always between 0.0 and 1.0:

```
t += 0.01 ;
if ( t>1.0 ) { t=0.0; }
```

Can you modify this to make the circle bounce backwards and forwards along the curve, rather than jumping to the start when it gets to the end?

Drawing a tangent vector using `bezierTangent()`

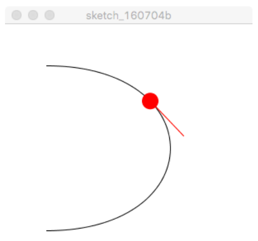
We next want to make a line that points in the direction that the circle is moving. We do this by finding a vector that is tangent to the curve at the current point and then drawing a line in the direction of that vector. Processing has a function to find the tangent vector:

```
float xTangent = bezierTangent(x1, x2, x3, x4, t);
float yTangent = bezierTangent(y1, y2, y3, y4, t);
```

As with the `bezierPoint()` function, we use `bezierTangent()` to calculate the `x` and `y` components separately.

To draw the tangent line, we draw a line from the current point to the current point plus the tangent vector. So:

```
stroke(255,0,0) ;
line( x, y, x+xTangent, y+yTangent ) ;
```



Notice how long that tangent vector is. A shorter line would be more useful to us to indicate the direction and size of the vector. To get a shorter line in the right direction, replace `xTangent` and `yTangent` with `xTangent/6` and `yTangent/6` (see diagram at right). You will be able to see that the tangent vector changes length as it moves.

Using translation

Instead of drawing our circle centred at point (`x`,`y`), we can get the same effect by drawing the circle centred at (0,0) and using a translation to get it to the right place. Replace the existing `ellipse()` command with:

```
translate(x, y) ;
ellipse(0, 0, 20, 20);
```

And replace the existing `line()` command with:

```
line( 0, 0, xTangent/6, yTangent/6 ) ;
```

You get the same result as before but you no longer need to use `x` and `y` in either of the shape definitions. Notice how this has simplified the parameters in the call to the `line()` function.

Using rotation

We can also remove `xTangent` and `yTangent` from the shape definition using a rotation.

Replace the `line()` function with a line starting at the origin and pointing along the x-axis:

```
line( 0, 0, 50, 0 );
```

This line does not change length as it moves but you'll notice that it also just keeps pointing to the right: it does not point along the curve. We can get it to point along the curve by adding another transformation:

```
rotate(angle) ;
```

But what should `angle` be? We can get the angle from the tangent vector (`xTangent`,`yTangent`) by using the function:

```
    float angle = atan2( yTangent, xTangent ) ;
```

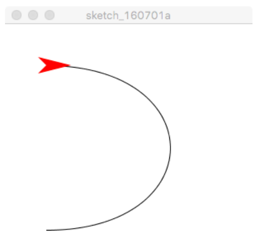
The name of the function, `atan2`, derives from “arctangent” with “2” arguments. The arctangent function returns the angle whose tangent is `yTangent`/`xTangent`. We need two arguments to distinguish between angles 180 degrees apart, which have the same numerical value for tangent but which point in opposite directions. By giving it `yTangent` and `xTangent` separately, it can distinguish between these cases (e.g., consider the difference between atan2(1,1) = 45 degrees and atan2(-1,-1) = 215 degrees).

So now we have shapes defined relative to the origin and pointing along the x-axis, which are put into their correct position and rotation using the functions `translate()` and `rotate()`.

A more complicated shape

This use of `translate()` and `rotate()` may not seem to be much of a win in this case, but it helps enormously if you have a more complicated shape. For example, replace the `ellipse()` and the `line()` with this shape:

```
    beginShape();
        vertex(0,0);
        vertex(-10,-10);
        vertex(30,0);
        vertex(-10,10);
    endShape(CLOSE);
```



You should get a red arrow shape that moves smoothly along the curve and points along the curve. If you tried to do this without using `translate()` and `rotate()`, you would need a lot of tedious mathematics to work out exactly where to put the four vertices.

Put the point of the arrow on the curve, rather than the tail

Can you modify the shape definition so that the front point of the arrow lies on the curve?

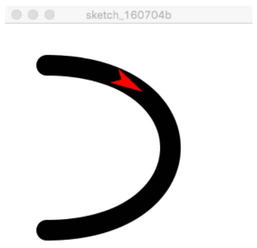
Moving a curve point using the mouse

As an example of interaction, we will write code to allow the user to move the starting point of the curve. We can do this by using the `mouseDragged()` callback function.

```
    void mouseDragged(){
      x1 = mouseX ;
      y1 = mouseY ;
    }
```

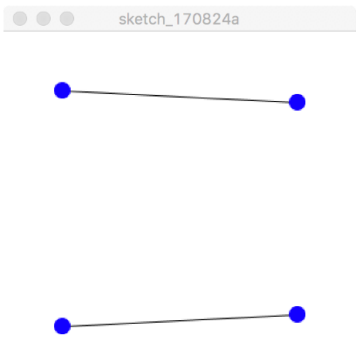
This allows you to move only the first point, (`x1`,`y1`). If you want to drag more points, you need to write code for the `mousePressed()` callback that finds which point the mouse is closest to, and remembers that point, then use the `mouseDragged()` callback to move that point. In this case, it would be convenient to have an array of the points’ locations rather than the separate variables we have used in this worksheet. Also, it helps if some sort of indicator mark is drawn at the points’ locations, such as a little square, so that the user knows where the points are.

Drawing the line differently



Processing uses the `strokeWeight()` to set the width of a line. Experiment with `strokeWeight()` to see if you can make it appear as if the shape is moving down a black pipe.

Intersecting two lines



In Assignment 3, you are going to have to find the intersection between two lines. So let's do a bit of work on handling a couple of lines.

In your code, get rid of the Bézier curve and get the code to draw just two lines. The example to the right also draws circles on the end points.

```
        stroke(0);
        noFill();
        line(x1,y1, x2,y2);
        line(x3,y3, x4,y4);
```

The mathematics of finding the intersection of two lines

How do we now find the intersection between these two lines P_1P_2 and P_3P_4? We went through this in lectures. What we are going to do is represent one of the lines *parametrically*, as a pair of equations giving x and y as a function of a parameter s; we then represent the other line *implicitly*, as a function that is equal to zero when a point lies on the line $ax + by + c = 0$. We substitute the parametric form of the first line into the implicit form of the second line and solve for the value s, which will tell us the location on the first line that is also on the second line.

First we write one of the lines parametrically in the form $P(s)=(1-s)P_1+sP_2$, which expands out to the two equations $x(s)=(1-s)x_1+sx_2$ and $y(s)=(1-s)y_1+sy_2$.

For the other line, P_3P_4, we use the implicit form of the line equation: $ax+by+c=0$.

We now substitute into that implicit equation the parametric equations for the first line, giving us:

$$a[(1-s)x_1+sx_2] + b[(1-s)y_1+sy_2] + c = 0$$

The value of s tells us where on line P_1P_2 there is a point that also lies on line P_3P_4. We need to find the value of s. We can rearrange that equation to solve for s:

$a[x_1+s(x_2-x_1)] + b[y_1+s(y_2-y_1)] + c = 0$	put all the "s" things together in each [] section
$\Rightarrow ax_1 + sa(x_2-x_1) + by_1 + sb(y_2-y_1) + c = 0$	expand out
$\Rightarrow ax_1 + by_1 + c + s[a(x_2-x_1)+b(y_2-y_1)] = 0$	group together "s" things and "non-s" things
$\Rightarrow s = -[ax_1 + by_1 + c] / [a(x_2-x_1)+b(y_2-y_1)]$	get s on its own on one side of the equals sign

Now, we've done all that manipulation without revealing how to calculate the values of a, b, c. They are calculated from the positions of P_2 and P_3, as explained in the lecture notes:

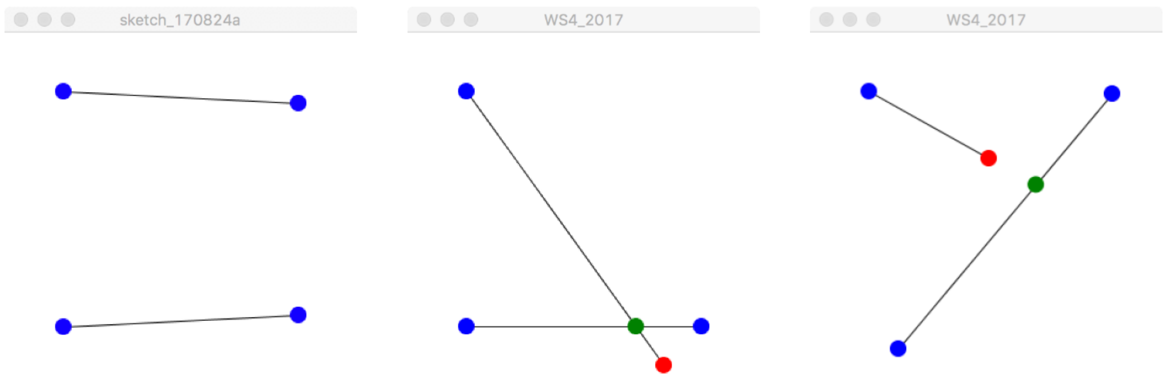
$$a = -(y_4-y_3) \qquad b = (x_4-x_3) \qquad c = x_3y_4-y_3x_4$$

That is a lot of maths to get a single intersection point. To program this up, we can do the following, which gets us the intersection point (`xi`,`yi`).

```
        float a = ...
        float b = ...
        float c = ...
        float s = ...
        float xi = (1-s)*x1 + s*x2;
        float yi = (1-s)*y1 + s*y2;
        fill(0,128,0);
        ellipse( xi, yi, 10, 10 );
```

You need to work out what to put in place of the "`...`"

If you run this, you won't see any difference at first because the two lines are nearly parallel (see picture below) so the intersection point is way outside the window to the right. Move one of the points so that the two lines intersect and you'll see the green intersection point appear in the window.

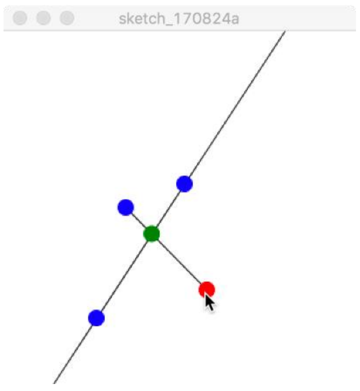


At left: what you'll see when the sketch starts: the intersection point is a long way off to the right, so you cannot see it.

In the middle: one of the end points (red) has been dragged down and the intersection point is drawn in green.

At right: the intersection is between infinitely long lines so it appears even if the line segments themselves don't intersect.

Challenges



Challenge 1: how do you change the code so that you only draw the intersection point if the intersection lies between point P_1=(`x1`,`y1`) and point P_2=(`x2`,`y2`)?
Hint: consider what the value of s will be for this case.

Challenge 2: how do you change the code to draw a line that goes through the two end points right to the edges of the window?
Hint: This requires you to find the intersection point of the infinite line with each of the four edges of the window and then draw a line between the middle two of those edge-intersections.