Real Space, Virtual Space

Until now we have been drawing and moving in the physical world where location is defined relative to *our* location and *our* facing in space. In our minds, we are the point of origin and forward is always in front.

We are now about to move into virtual, computational space where we need an objective, numerically-based way to define location on what we will call a **canvas** for the programs you will write, which we will refer to as **sketches**. Hence, the Cartesian Plane or the x,y coordinate system.

If you are new to programming, you may wish to pause here and watch these videos first. You are also welcome to move on and follow along conceptually.

Videos 1-10, 12-14 (~2.5 hours)

Below is 20 x 20 canvas. 20 what? 20 pixels. And each pixel has an (x,y) coordinate that locates the top-left corner of the pixel.

The x-dimension (horizontal) begins at 0 and ends at 20.
The y-dimension (vertical) begins at 0 and ends at 20.

The coordinates of the bottom right corner are always equal the width and height of the canvas (20, 20). However, since the coordinate locates the top-left corner of the pixel, the (20,20) pixel is actually offscreen and invisible. If you draw a point at (20,20), you will not see it.

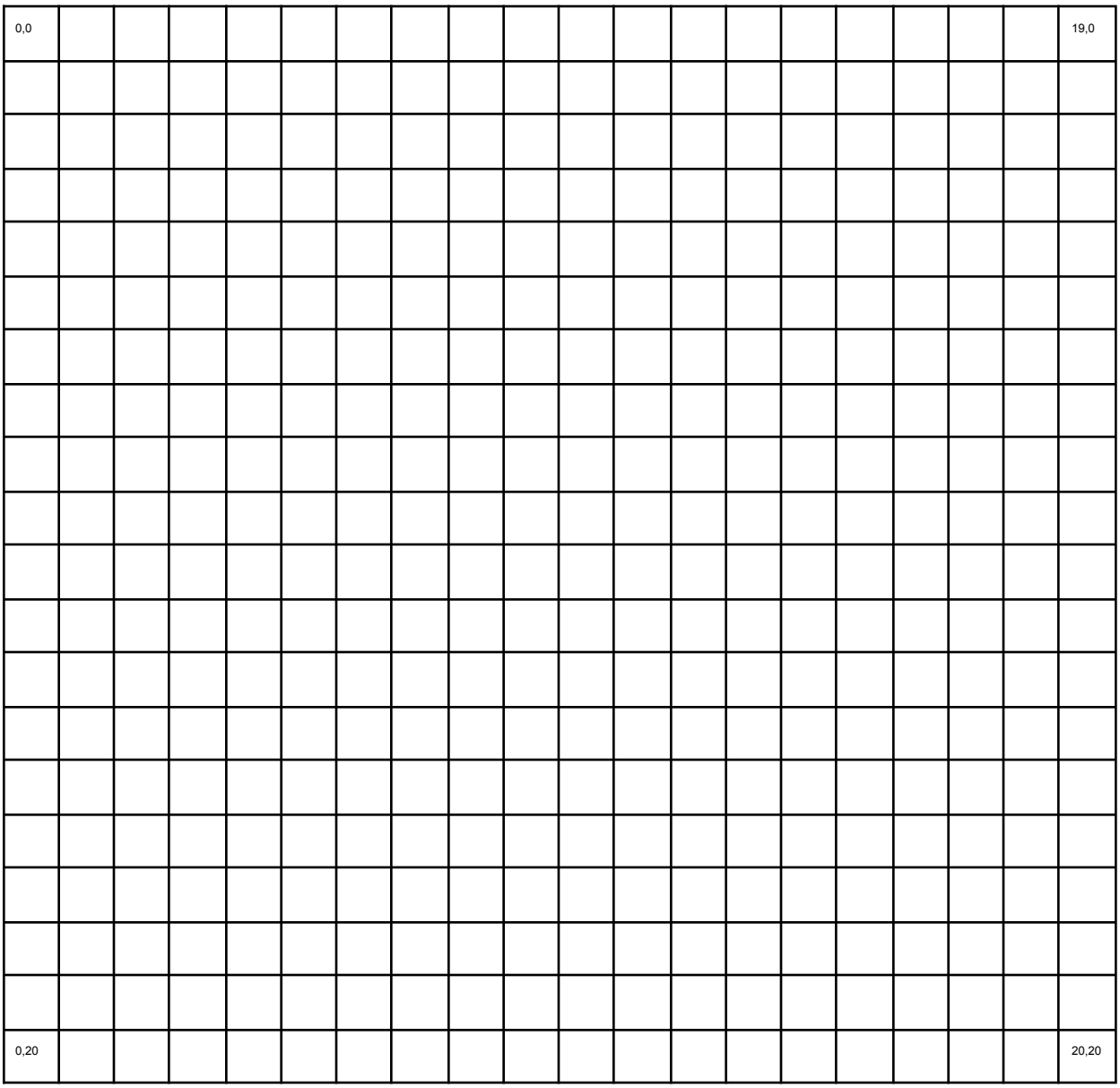
If this feels strange to you, it's because the x,y coordinate system in graphical programming is different from the x,y coordinate system you remember from math. For starts, the point of origin (0,0) is in the top-left corner rather than in the middle of the page. And the y-dimension is flipped upside down. The numbers increase as you move down.

This might seem arbitrary, but given what you're trying to accomplish here in code (draw things on a screen) versus what you're trying to accomplish in math (graph equations and data), this is much more sane interpretation of the Cartesian Plane. For one thing, in math, there is no conception of canvas size. The Cartesian Plane extends forever in every direction.

So, if you wanted to draw a point in the middle of of your 20 x 20 canvas, you might try something like:

```
point(10, 10);
```

x-dimension →



If you're feeling a bit underwhelmed by the x,y coordinate system as an invention, keep in mind that both the grid plan for cities and the lat-long geographic coordinate system have only been around for 2,000+ years of the 200,000 years of human existence. Such a rectilinear and objective view of the world is far from a given as we will see in Chapter 1 when we fool-hardedly attempt to do something as simple as draw a circle in the rectilinear universe of x,y coordinates.

Connect the dots: Without looking at the grid, can you come up with 5 points that would yield a bear-like drawing if you connected the points? Try it to confirm.

How do we move?

At this point you should at least know how to draw things on the screen in p5 that do not move by providing coordinate values.

We need variables `x` and `y` to replace those coordinates so the coordinates can change over time.

```
x = x + 1;
y = y + 1;
```

Code Example 1.1: Linear Pathway

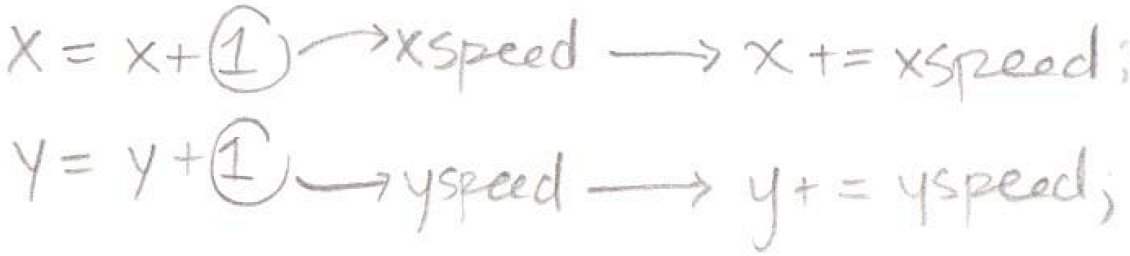
Play with the code.

You may have noticed that you can change how the line moves by changing the number that determines how quickly x and y change. Can you figure out what you need to do to change speed and what you need to do to change direction?

Can you draw how the line would move in a single frame of animation for each of these `xspeed` and `yspeed` combinations?

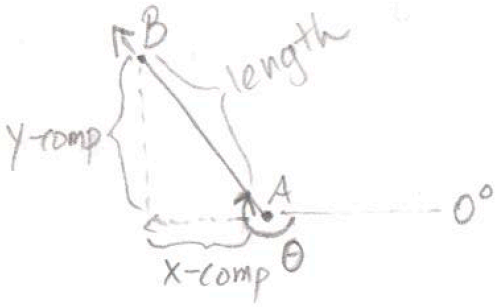
`xspeed = 3;` `yspeed = 9;`	`xspeed = -3;` `yspeed = -15;`	`xspeed = 3;` `yspeed = -15;`	`xspeed = -3;` `yspeed = 9;`

And since we're going to manipulate these numbers, let's make them variables `xspeed` and `yspeed` so they're easier to reference and change.



To help us visualize what's going on, let's bring back that diagram of a line.

- Start point
- End point
- Length
- Direction as Angle
- Direction as Slope
  - x-component
  - y-component



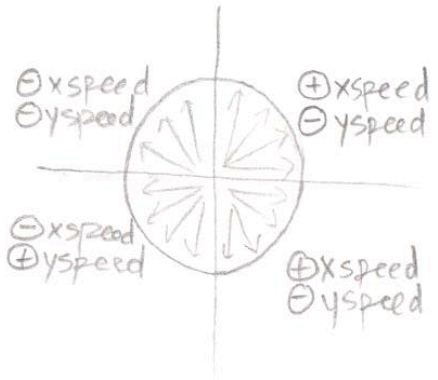
If a line moves from point A to point B in 1 frame of animation, you could say the x-component and the y-component are the `xspeed` and `yspeed` respectively.
Quiz:

What would the `xspeed` and `yspeed` be if it took 60 frames to move from point A to point B?

Speed versus Direction

Speed is determined by the size of the numbers.

Direction is determined at a macro level by the sign of `xspeed` and `yspeed`.

The sign of `xspeed` and `yspeed` determines the direction quadrant. - `xspeed` always go left. + `xspeed` always go right. - `yspeed` always go up. + `yspeed` always go down.	

Direction at a micro level by the relative size of `xspeed` and `yspeed`.

If `xspeed` is 4x `yspeed`, the line will move across the screen at twice the rate that it will move down the screen resulting in a mostly horizontal line. If `yspeed` is 4x `xspeed`, the line will move down the screen at twice the rate that it will move across the screen resulting in a mostly vertical line. If `xspeed` is equal to `yspeed`, you get a line moving at a 45-degree towards the bottom right of the screen.

`xspeed = yspeed`	`xspeed = 4;`
	`yspeed = 4;`
	`yspeed = xspeed*1`
			45◦

`xspeed > yspeed`	`xspeed = 40;`
	`yspeed = 4;`
	`yspeed = xspeed/10`
			Almost horizontal.

`xspeed < yspeed`	`xspeed = 4;`
	`yspeed = 40;`
	`yspeed = xspeed*10`
			Almost vertical

Quiz:

How would get a line that always moves to the corner of the screen regardless of the aspect ratio of your sketch canvas?

Re-writing Linear Motion

Now let's rewrite our algorithm for linear motion so that we can control speed independently of direction. This is something we will continue to do throughout the workbook. At first it may feel like we're making things unnecessarily complicated by making things more abstract. However, it is important that we do this for 2 reasons:

1. It will give us more expressive control over our animations.
2. It will make it easier for us to hook sensor data up in the future. Right now, if you want to change direction, you need to change both `xspeed` and `yspeed`. Ideally what you want is to be able to hook up say the position of your right hand to the direction.

```
x += xspeed;
y += yspeed;
```

What we want is a single variable to control speed and a single variable we will call `yscl` to control direction or more precisely the slope or slant of the line. We call it `yscl` because we are going to use that value to scale `xspeed` to in order to determine `yspeed`.

```
speed = 1;
yscl = 1/10; // yspeed / xspeed
```

As a cheat, we can simply let `xspeed` determine the speed and calculate `yspeed` based off of `xspeed`.

```
xspeed = 1;
yscl = 0.1;
yspeed = yscl * xspeed;
```

If we want to change just the speed without affecting direction, we adjust `xspeed` and `yspeed` will adjust automatically and maintain a 1:10 relationship to `xspeed`.

To go twice as fast:

```
xspeed = 2;
yscl = 0.1;
yspeed = yscl * xspeed;
```

If we want to adjust direction without affecting speed, we adjust slope and change the `yspeed : xspeed` relationship.

To create a line that is twice as steep:

```
xspeed = 1;
yscl = 0.2;
yspeed = yscl * xspeed;
```

Full example:

https://editor.p5js.org/move.mimi/sketches/fA--n2kyg

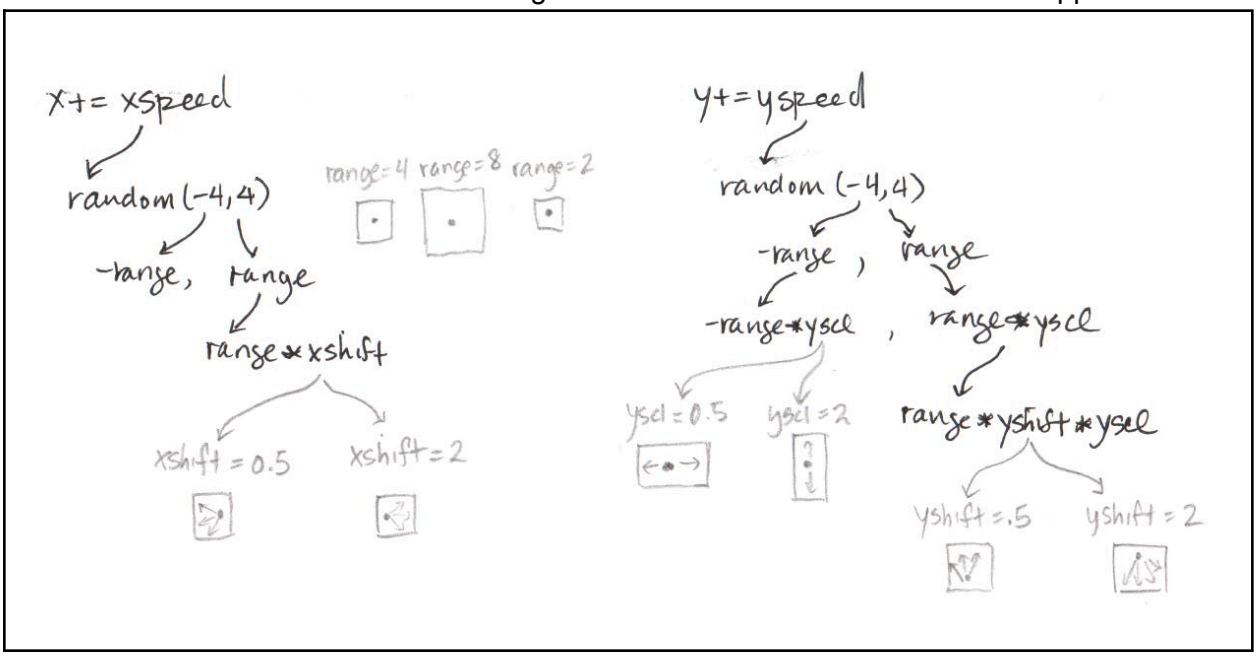
Summary: The Parameters of Linear Motion

Translated into motion on a 2-dimensional plane:

`xspeed`		Step size of horizontal movement.
`yspeed`		Step size of vertical movement.
`yscl`	A value >= 0 centered around 1.	Size of step

Random Motion

If linear motion is motion that never changes direction then random motion is its opposite.



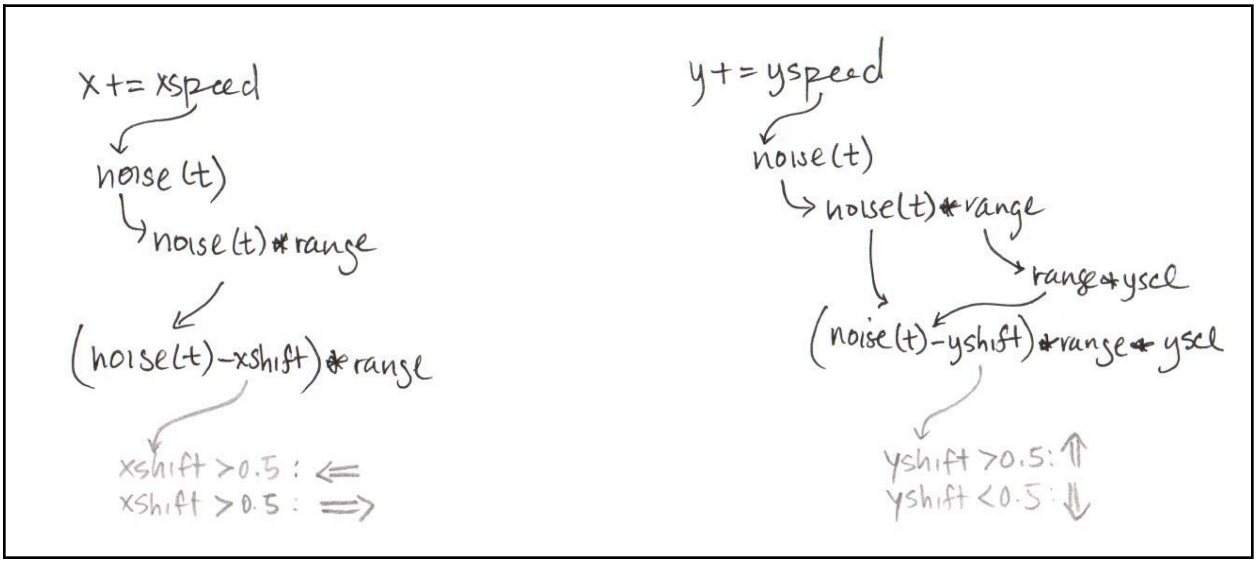
Summary: The Parameters of Random Motion

Translated into motion on a 2-dimensional plane:

`range`	A value >= 0.	`range` determines the step size of random motion and therefore the speed of movement.
`yscl`	A value >= 0 centered around 1.	`yscl` determines how narrow or wide is the range of motion. If `yscl` > 1, motion will be more vertical. If `yscl` < 1, motion will be more horizontal.
`xshift`	A value > 0 centered around 1.	If `xshift` > 1, we will drift to the right. If `xshift` < 1, we will drift to the left.
`yshift`	A value > 0 centered around 1.	If `yshift` > 1, we will drift down. If `yshift` < 1, we will drift up.
`interval`	A value > 0.	How often we change direction.

Noisy Motion

Noisy motion is random motion that has been averaged and smoothed.



Summary: The Parameters of Noisy Motion

Translated into motion on a 2-dimensional plane:

`tspeed`	A value > 0.	How errative / smooth is the noisy motion. The greater the `tspeed`, the more erratic and random the motion. The smaller the `tspeed`, the smoother the motion.
`range`	A value >= 0.	Step size of random motion and therefore the speed of movement.
`yscl`	A value >= 0 centered around 1.	How narrow or wide is the range of motion. If `yscl` > 1, motion will be more vertical. If `yscl` < 1, motion will be more horizontal.
`xdrift`	A value > 0 centered around 1.	If `xdrift` > 0.5, we will drift to the left. If `xdrift` < 0.5, we will drift to the right.
`ydrift`	A value > 0 centered around 1.	If `ydrift` > 0.5, we will drift to the left. If `ydrift` < 0.5, we will drift to the right.
`interval`	A value > 0.	How often we change direction.

Circular Motion

If random was utterly unpredictable, constant and complete change.

And random was pretty unpredictable, constant, but incomplete change.

Then circular pathways are utterly predictable, constant, yet perfectly smooth, incremental change.

Creating circular motion in a Cartesian world is an awkward business. Circular polar coordinates (radius, angle) need to be converted to rectilinear Cartesian coordinates (x,y).

```
x = cos(angle)*range+ centerX;
y = sin(angle*yfreq)*range*yscl+ centerY;
```

Summary: The Parameters of Circular Motion

Translated into motion on a 2-dimensional plane:

`aspeed`	A value > 0.	How quickly we are moving around the circle. The faster we move, the less circular and the more angular the motion.
`range`	A value >= 0.	Size of the circle.
`yscl`	A value >= 0 centered around 1.	Orientation of the ellipse. If `yscl` < 1, we get a vertical ellipse. If `yscl` < 1 and yscl > -1, we get a horizontal ellipse.
`yfreq`	A value > 0 centered around 1.	If `yfreq` > 1, we will progress around the circle more quickly in the vertical direction than the horizontal. If `yfreq` < 1, we will progress around the circle more quickly in the horizontal direction than the vertical. If `yfreq` is not a whole number, we will create asymmetrical pathways.
`centerX`		x-coordinate of the center of the circle.
`centerY`		y-coordinate of the center of the circle.