Lecture 5: Parametric Curves

Lines, Circles & Ellipses — Drawing with a Dial

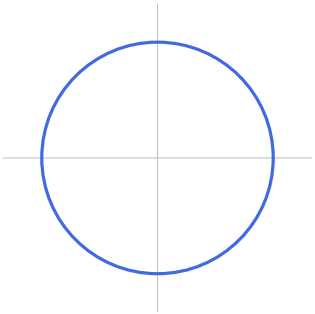
CS 453 — Computer Graphics

Fakhir Shaheen

Forman Christian University

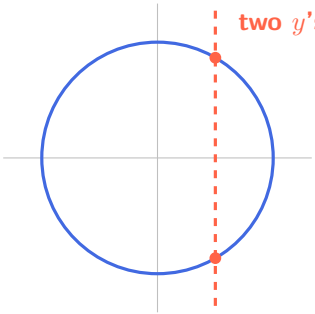
The Idea: a Dial Named t

What We Cannot Say with $y = f(x)$



A function $y = f(x)$ answers:
"at this x, how high?"

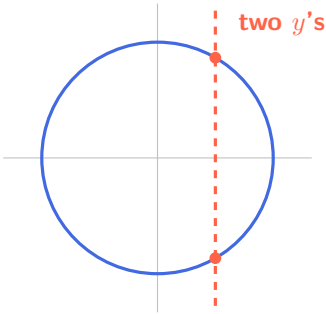
What We Cannot Say with $y = f(x)$



A function $y = f(x)$ answers:
"at this x, how high?"

- a circle has **two** answers per x —
  f is not allowed to give two

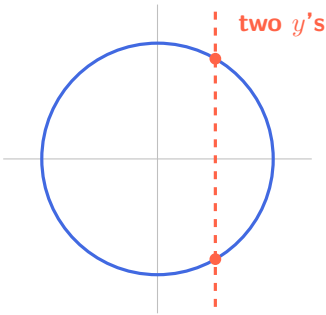
What We Cannot Say with $y = f(x)$



A function $y = f(x)$ answers:
"at this x, how high?"

- a circle has **two** answers per x — f is not allowed to give two
- week 1 flashback: tracing $y = c_y \pm \sqrt{r^2 - x^2}$ left **gaps** at the steep sides
- a *vertical* line: no f at all

What We Cannot Say with $y = f(x)$



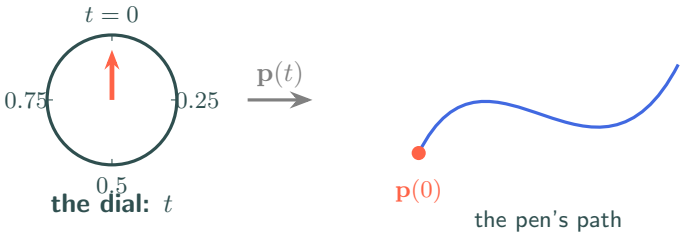
A function $y = f(x)$ answers:
"at this x, how high?"

- a circle has **two** answers per x — f is not allowed to give two
- week 1 flashback: tracing $y = c_y \pm \sqrt{r^2 - x^2}$ left **gaps** at the steep sides
- a *vertical* line: no f at all

The shape isn't the problem.
The question is.

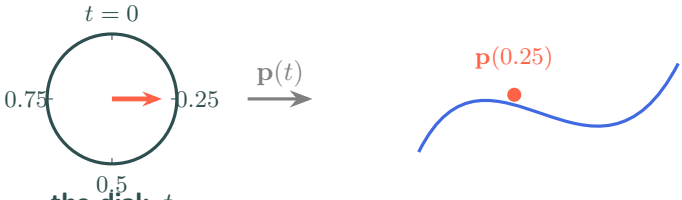
Change the Question: Describe the Journey

Stop asking *"what is y at x?"*
Ask: **"where is the pen at time t?"**



Change the Question: Describe the Journey

Stop asking *"what is y at x?"*
Ask: **"where is the pen at time t?"**

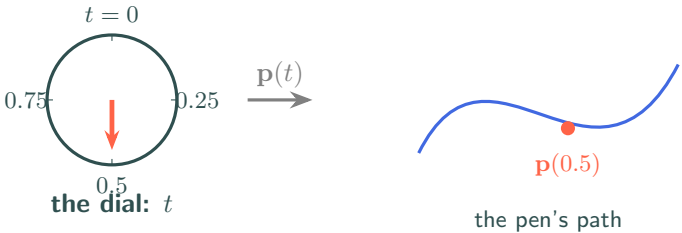


the dial: t

the pen's path

Change the Question: Describe the Journey

Stop asking *"what is y at x?"*
Ask: **"where is the pen at time t?"**

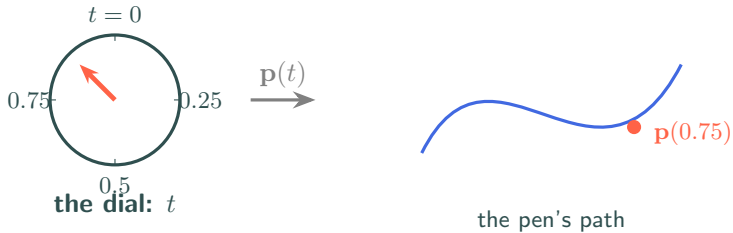


the dial: t

the pen's path

Change the Question: Describe the Journey

Stop asking *"what is y at x?"*
Ask: **"where is the pen at time t?"**

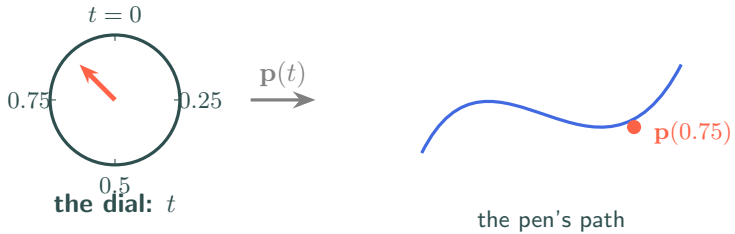


the dial: t

the pen's path

Change the Question: Describe the Journey

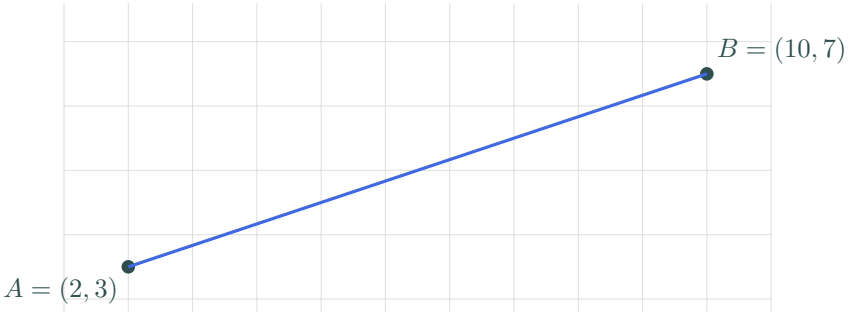
Stop asking *"what is y at x?"*
Ask: **"where is the pen at time t?"**



A curve is a **function from a number to a point**: $t \mapsto \mathbf{p}(t)$

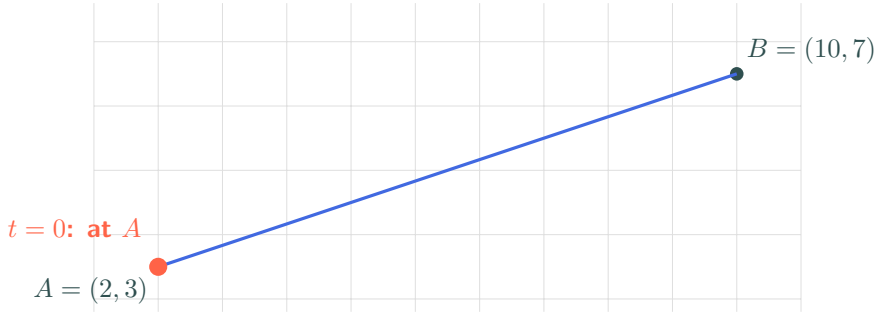
One input, *two* outputs (x and y travel together) — exactly what `Vec2` was built for.

The Simplest Journey: a Straight Line



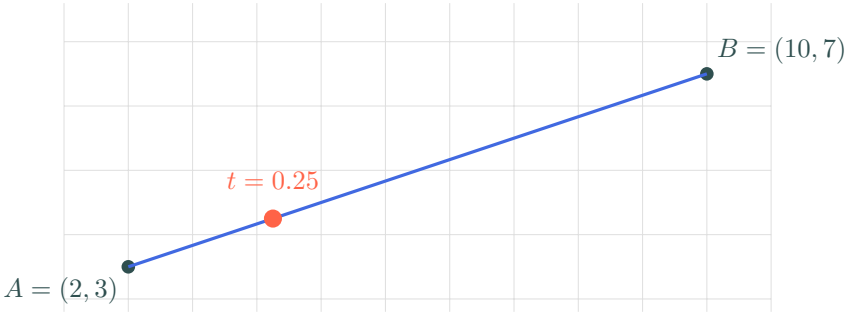
$$\mathbf{p}(t) \;=\; A \;+\; t\,(B - A)$$

The Simplest Journey: a Straight Line



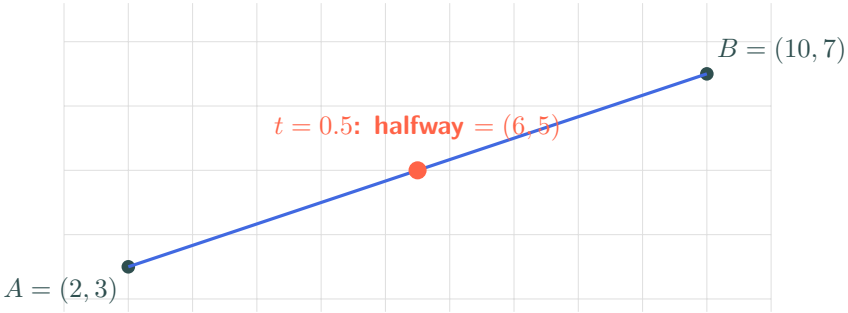
$$\mathbf{p}(t) \;=\; A \;+\; t\,(B - A)$$

The Simplest Journey: a Straight Line



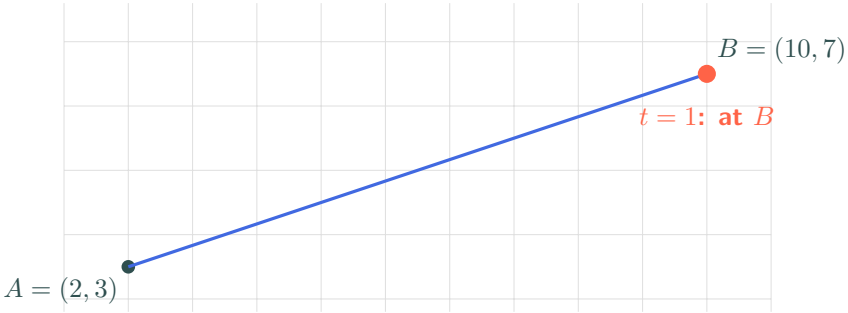
$$\mathbf{p}(t) \;=\; A \;+\; t\,(B - A)$$

The Simplest Journey: a Straight Line



$$\mathbf{p}(t) \;=\; A \;+\; t\,(B - A)$$

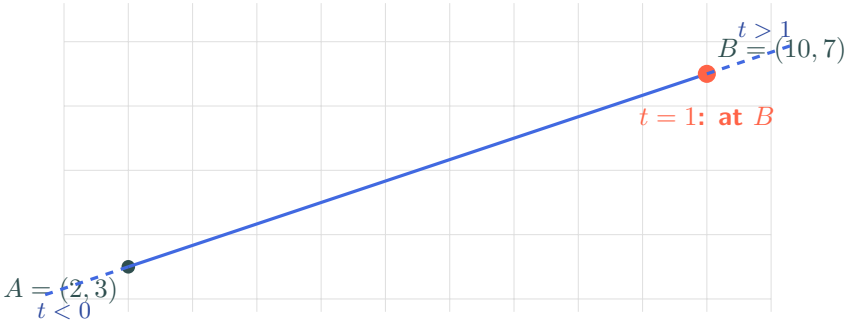
The Simplest Journey: a Straight Line



$$\mathbf{p}(t) \;=\; A \;+\; t\,(B - A)$$

"start at A, walk fraction t of the trip toward B" — **lerp**, now between *points*.

The Simplest Journey: a Straight Line



$$\mathbf{p}(t) \;=\; A \;+\; t\,(B - A)$$

"start at A, walk fraction t of the trip toward B" — **lerp**, now between *points*.

And as Code, It's Monday's Homework

```
def line_point(a: Vec2, b: Vec2,
               t: float) -> Vec2:
    """t=0 -> a, t=1 -> b."""
    return a + (b - a) * t
```

One line — because `Vec2`'s `+`, `-`, `*` already move *both* coordinates in step.

Notice who does what:

- `b - a`: point − point = **vector** (the trip)
- `* t`: scale the trip
- `a +`: point + vector = **point** (arrive)

Monday's "meaning table" typed out verbatim.

t is *not* x

t measures **progress along the journey** — 0 start, 1 end — and knows nothing about horizontal. A *vertical* line is no longer special: $A = (5, 1)$, $B = (5, 8)$ works like any other.

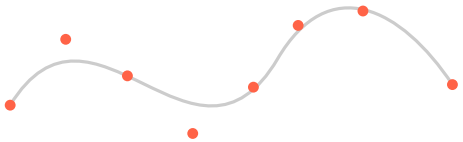
Sampling: from Formula to Pixels

Screens Don't Eat Formulas

$\mathbf{p}(t)$ is perfect and continuous; the screen wants finitely many pixels.
The bridge: **sample, then connect**.

Screens Don't Eat Formulas

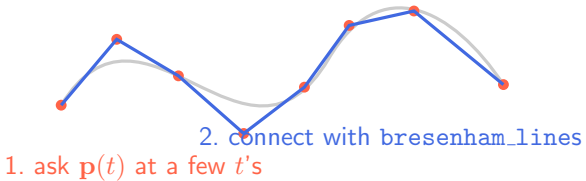
$\mathbf{p}(t)$ is perfect and continuous; the screen wants finitely many pixels.
The bridge: **sample, then connect**.



1. ask $\mathbf{p}(t)$ at a few t's

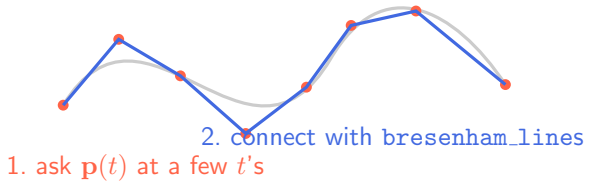
Screens Don't Eat Formulas

$\mathbf{p}(t)$ is perfect and continuous; the screen wants finitely many pixels.
The bridge: **sample, then connect**.



Screens Don't Eat Formulas

$\mathbf{p}(t)$ is perfect and continuous; the screen wants finitely many pixels.
The bridge: **sample, then connect**.

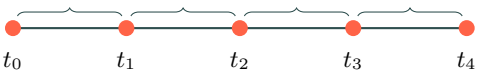


If the dots are close enough, the eye sees a **curve** —
every "curve" you have ever seen on a screen is this trick.

sample_parametric — Mind the Fence-Posts

```
pts = sample_parametric(f, t0, t1,
                        steps=4)
# -> [f(t0), ..., f(t1)]
#    5 points, BOTH ends included
```

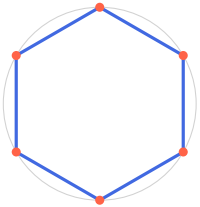
steps $= 4$ *gaps* $\Rightarrow$ 5 points



- steps counts the **gaps**, you return steps + 1 points
- same inclusive convention as A1's vertical_gradient
- steps < 1 $\Rightarrow$ return [f(t0)] alone

The off-by-one here is the classic of the genre: forget the + 1 and every circle you draw has a **notch** where the ends fail to meet.

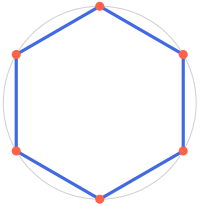
How Many Samples Are Enough?



$\texttt{steps} = 6$

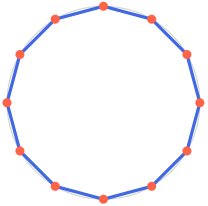
a hexagon, plainly

How Many Samples Are Enough?



$\texttt{steps} = 6$

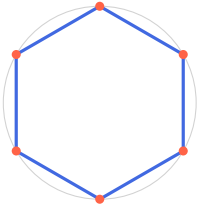
a hexagon, plainly



$\texttt{steps} = 12$

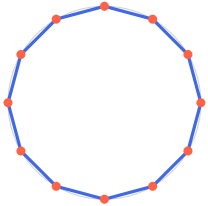
nearly there

How Many Samples Are Enough?



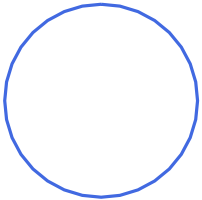
$\texttt{steps} = 6$

a hexagon, plainly



$\texttt{steps} = 12$

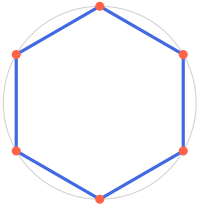
nearly there



$\texttt{steps} = 32$

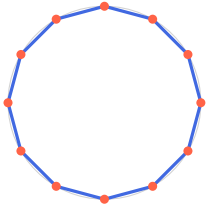
the eye gives up: circle

How Many Samples Are Enough?



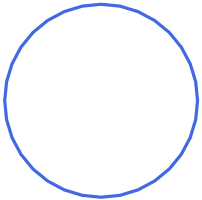
$\texttt{steps} = 6$

a hexagon, plainly



$\texttt{steps} = 12$

nearly there



$\texttt{steps} = 32$

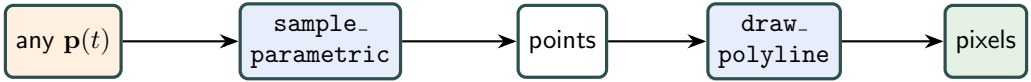
the eye gives up: circle

- too few: the polygon shows; too many: wasted work, same picture
- rule of thumb: a step every **2–3 pixels** of curve length; $\texttt{steps} \approx 64$ covers most shapes on screen

The Machine Doesn't Know What It's Sampling

`sample_parametric` never asks *what* `f` is — functions are values in Python:

```
line  = lambda t: a + (b - a) * t
orbit = lambda t: curves.circle_point(c, r, t * 2 * math.pi)

pts = sample_parametric(orbit, 0.0, 1.0, 64)    # same sampler!
```

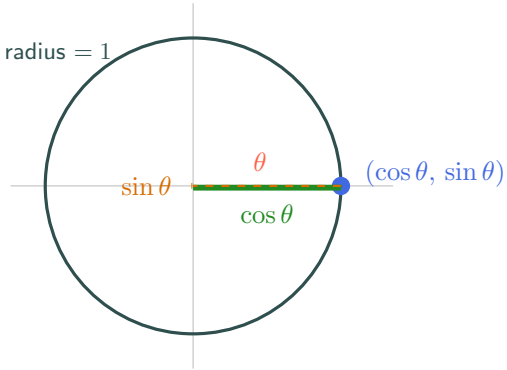


One pipeline. New curve = new `f`, **zero** new drawing code.
That is the whole payoff of thinking parametrically — Friday's Béziers will just be another `f`.

Circles & Ellipses

$\cos$ and $\sin$, Rebuilt from the Picture

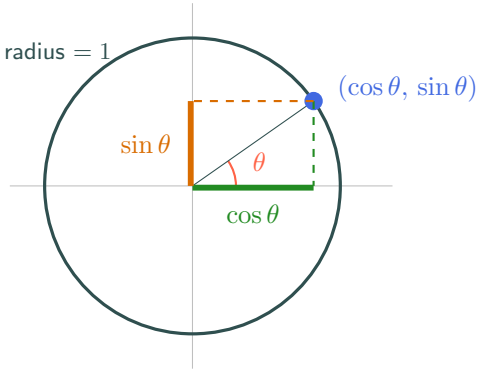
Forget triangle formulas. **This** is the definition worth keeping:



Walk angle θ around a circle of radius 1: $\cos\theta$ and $\sin\theta$ simply *are* your x and y.

cos and sin, Rebuilt from the Picture

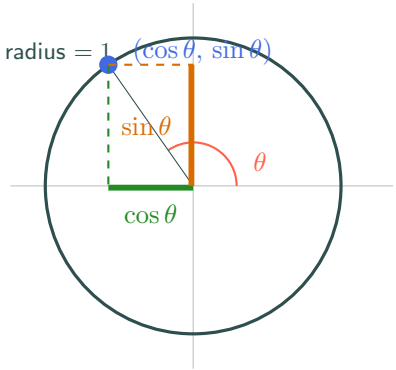
Forget triangle formulas. **This** is the definition worth keeping:



Walk angle θ around a circle of radius 1: $\cos\theta$ and $\sin\theta$ simply *are* your x and y.

$\cos$ and $\sin$, Rebuilt from the Picture

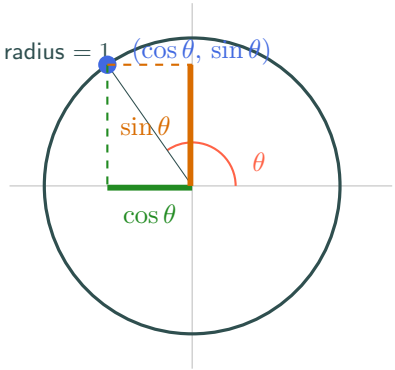
Forget triangle formulas. **This** is the definition worth keeping:



Walk angle θ around a circle of radius 1: $\cos\theta$ and $\sin\theta$ simply *are* your x and y.

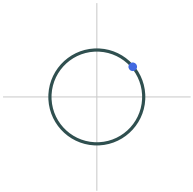
cos and sin, Rebuilt from the Picture

Forget triangle formulas. **This** is the definition worth keeping:



Walk angle θ around a circle of radius 1: $\cos\theta$ and $\sin\theta$ simply *are* your x and y. That's it. Everything circular in this course is this picture, scaled and shifted.

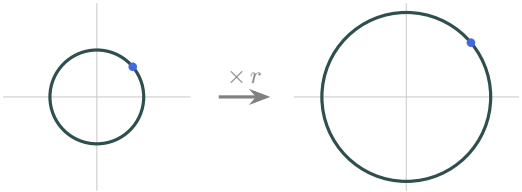
Scale It, Ship It: `circle_point`



$(\cos\theta, \sin\theta)$

radius 1, at the origin

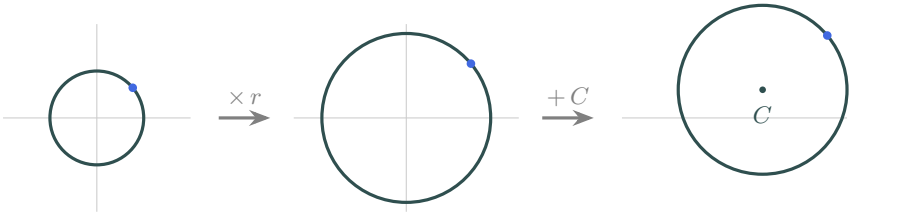
Scale It, Ship It: `circle_point`



$(\cos\theta, \sin\theta)$
radius 1, at the origin

$(r\cos\theta,\ r\sin\theta)$
radius r, still at the origin

Scale It, Ship It: `circle_point`

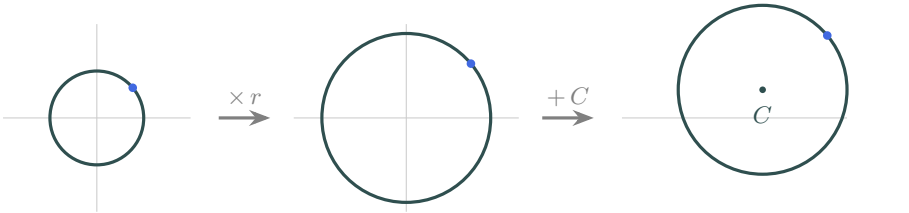


$(\cos\theta, \sin\theta)$
radius 1, at the origin

$(r\cos\theta,\ r\sin\theta)$
radius r, still at the origin

$(c_x + r\cos\theta,\ c_y + r\sin\theta)$
radius r, centered on C

Scale It, Ship It: `circle_point`



$(\cos\theta, \sin\theta)$
radius 1, at the origin

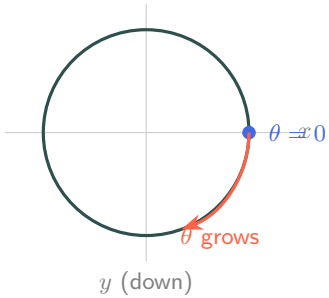
$(r\cos\theta,\ r\sin\theta)$
radius r, still at the origin

$(c_x + r\cos\theta,\ c_y + r\sin\theta)$
radius r, centered on C

$$\texttt{circle\_point}(C, r, \theta) = (c_x + r\cos\theta,\ c_y + r\sin\theta)$$

— exactly the three-arrow story above: unit picture, stretch, move.

One Screen-World Footnote

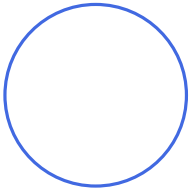


On paper, growing θ turns counter-clockwise.

On our screen, y points **down**, so the *same formula* turns **clockwise on screen**.

Nothing to fix — just don't be surprised. (Same convention flip as Monday's cross product.)

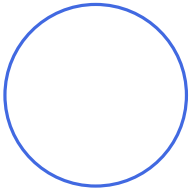
Arcs Come Free: Sample Less of the Dial



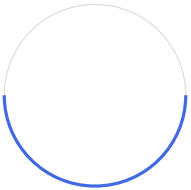
$\theta : 0 \to 2\pi$

full circle

Arcs Come Free: Sample Less of the Dial

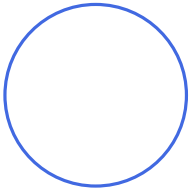


$\theta : 0 \to 2\pi$
full circle

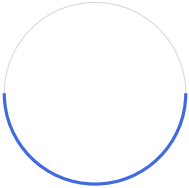


$0 \to \pi$
half (screen: lower half)

Arcs Come Free: Sample Less of the Dial



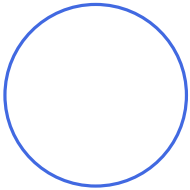
$\theta : 0 \to 2\pi$
full circle



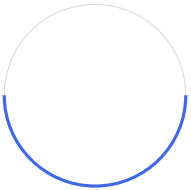
$0 \to \pi$
half (screen: lower half)

$0 \to \pi/2$
quarter

Arcs Come Free: Sample Less of the Dial



$\theta : 0 \to 2\pi$
full circle

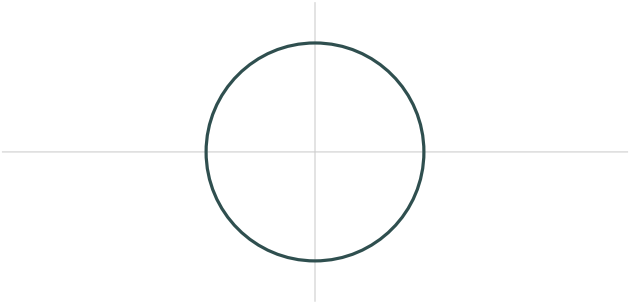


$0 \to \pi$
half (screen: lower half)

$0 \to \pi/2$
quarter

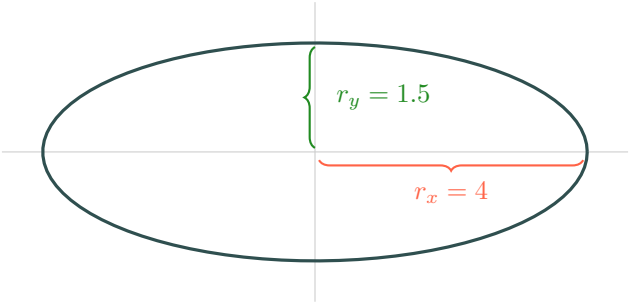
Rainbows, smiles, pie slices, clock hands — all just `sample_parametric` over a *sub-range* of θ. No new machinery.

Ellipse: Stretch Each Axis by a Different Amount



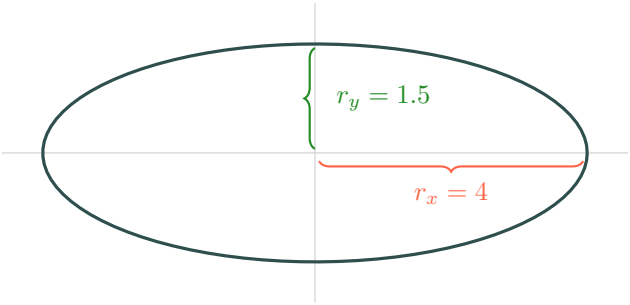
$r_x = r_y = r$: **a circle**

Ellipse: Stretch Each Axis by a Different Amount



$$(c_x + r_x \cos\theta,\ c_y + r_y \sin\theta)$$

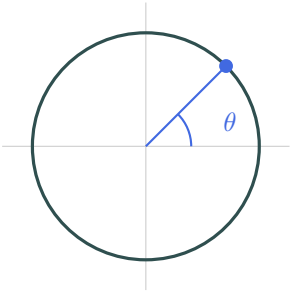
Ellipse: Stretch Each Axis by a Different Amount



$$(c_x + r_x \cos\theta,\ c_y + r_y \sin\theta)$$

One formula, one new letter: stretch x by r_x, y by r_y.
That is `ellipse_point`; `circle_point` is just the $r_x = r_y$ special case.

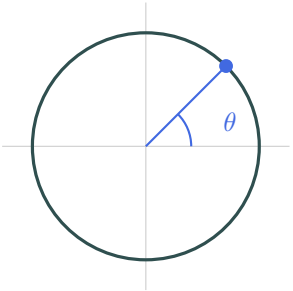
An Honest Subtlety: What Is θ on an Ellipse?



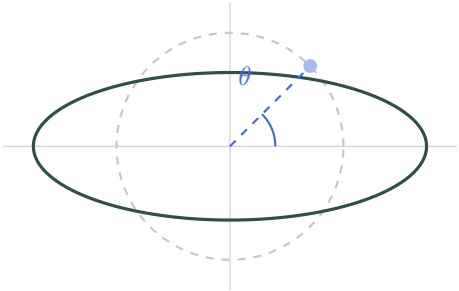
$$(r\cos\theta,\ r\sin\theta)$$

dial 45° $=$ **rim** 45°

An Honest Subtlety: What Is θ on an Ellipse?



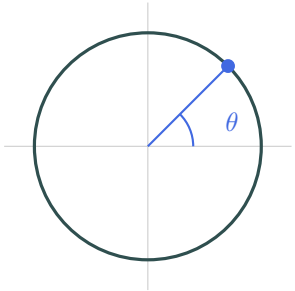
$(r\cos\theta,\ r\sin\theta)$



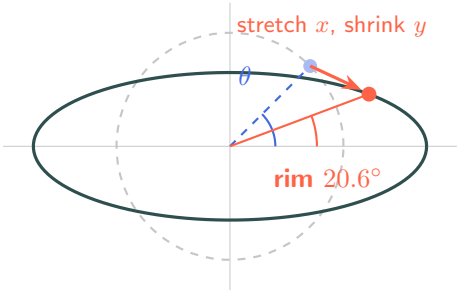
$(r_x\cos\theta,\ r_y\sin\theta)$

dial 45° = **rim** 45°

An Honest Subtlety: What Is θ on an Ellipse?



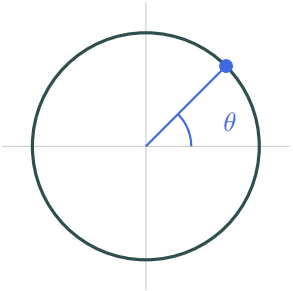
$(r\cos\theta,\ r\sin\theta)$



$(r_x\cos\theta,\ r_y\sin\theta)$

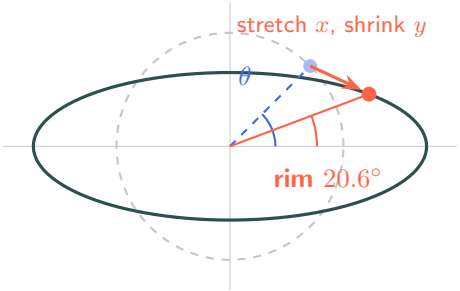
dial $45°$ $=$ **rim** $45°$

An Honest Subtlety: What Is θ on an Ellipse?



$(r\cos\theta,\ r\sin\theta)$

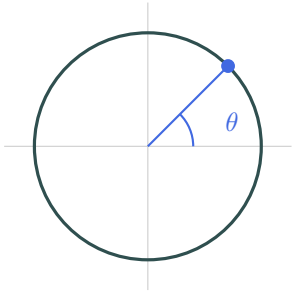
dial $45°$ $=$ **rim** $45°$



$(r_x\cos\theta,\ r_y\sin\theta)$

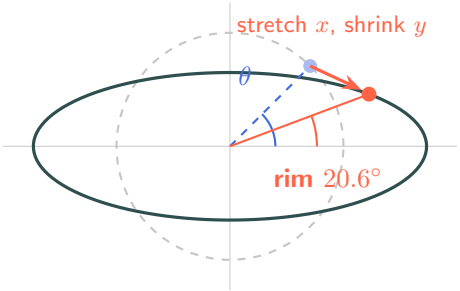
same dial $45°$**, but rim** $20.6°$

An Honest Subtlety: What Is θ on an Ellipse?



$(r\cos\theta,\ r\sin\theta)$

dial $45°$ $=$ **rim** $45°$

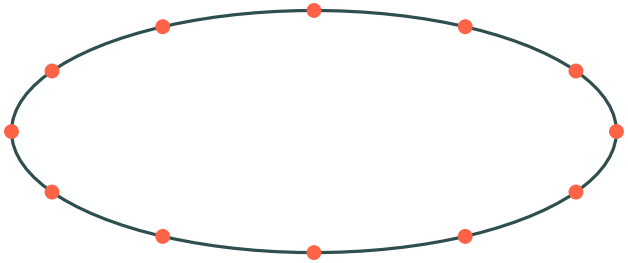


$(r_x\cos\theta,\ r_y\sin\theta)$

same dial $45°$**, but rim** $20.6°$

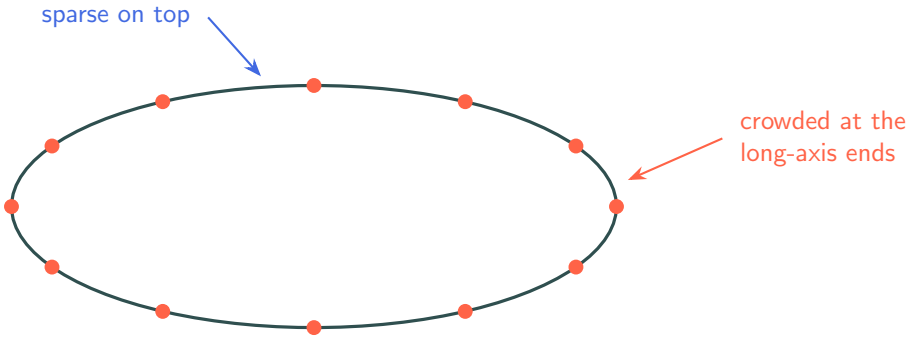
θ is the **knob you turn**, not an angle you can measure on the shape.
The stretch drags the point toward the long axis — only $r_x = r_y$ puts the two back in step.

. . . So Equal θ Steps Are Not Equal Rim Steps



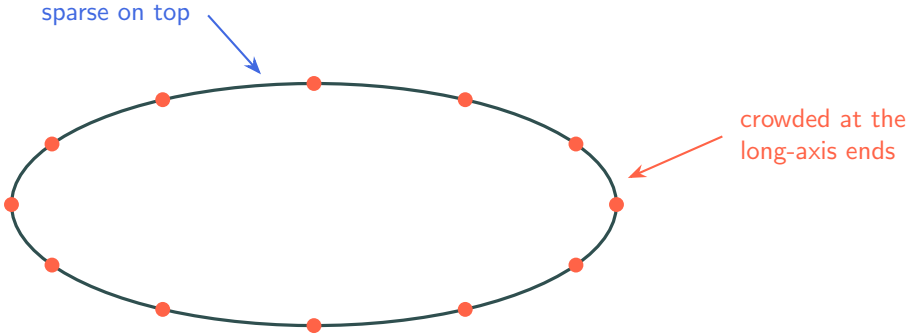
Equal steps of $\theta \neq$ equal steps *along the rim*.

. . . So Equal θ Steps Are Not Equal Rim Steps



Equal steps of $\theta \neq$ equal steps *along the rim*.

. . . So Equal θ Steps Are Not Equal Rim Steps



Equal steps of $\theta \neq$ equal steps *along the rim*.

- for **drawing**: harmless — the polyline is still the right shape
- for **motion**: a ball driven by `ellipse_point`$(\theta{=}2\pi t)$ visibly **speeds up** at the flat ends and **crawls** over the top — "constant speed" on an ellipse is not free (nothing this week needs it; your eyes will still notice)

From Points to Pixels

draw_polyline — and Where Rounding Hides

Curves live in floats; `bresenham_line` eats ints. **Someone** must round.

```
def draw_polyline(img, pts, color,
                  closed=False):
    for p, q in zip(pts, pts[1:]):
        painter.bresenham_line(img,
            round(p.x), round(p.y),
            round(q.x), round(q.y),
            color)
    # closed: also last -> first
```

What breaks with `int()`:

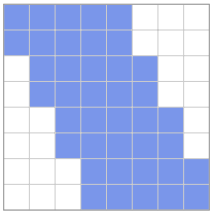
- `int()` **truncates**: $(3.7, 5.9) \to (3, 5)$ — every point dragged toward the top-left
- a whole curve shifts *half a pixel*, and unevenly — visible wobble on shallow arcs
- `round()` splits the error fairly

The rule that scales to Week 14

Keep **full float precision** through every computation; round **once**, at the last moment, where math meets pixels. (Same reason A1's `blend_pixel` added 0.5 before `int`.)

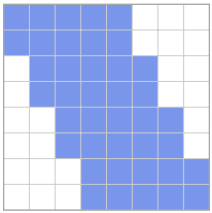
A Thick Line — the Tempting Wrong Way

"Just draw it 5 times,
shifted down a pixel each time?"



A Thick Line — the Tempting Wrong Way

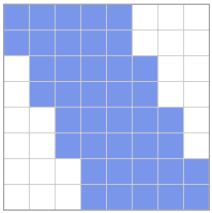
"Just draw it 5 times,
shifted down a pixel each time?"



- × on a **steep** line, "shift in x? in y?" — direction-dependent mess
- × diagonal width comes out $\sim$30% **thinner** (you shifted along an axis, not *across the line*)

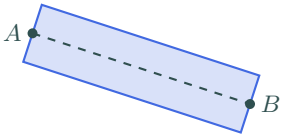
A Thick Line — the Tempting Wrong Way

"Just draw it 5 times,
shifted down a pixel each time?"



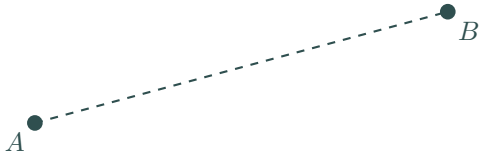
- × on a **steep** line, "shift in x? in y?" — direction-dependent mess
- × diagonal width comes out ∼30% **thinner** (you shifted along an axis, not *across the line*)

The right question:
"thick line" is not a line at all —
it's a **rectangle** *wearing a trench coat.*



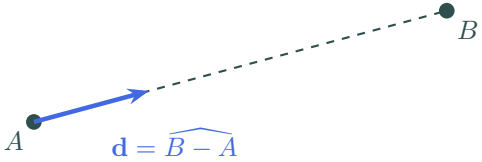
— and we know how to build rectangles from vectors. Next slide.

Building the Rectangle: `perp` Earns Its Keep



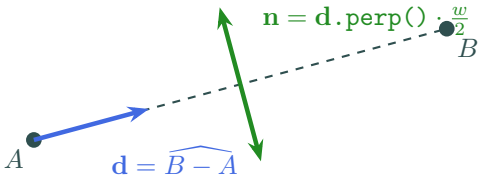
Given: endpoints A, B, width w.

Building the Rectangle: perp Earns Its Keep



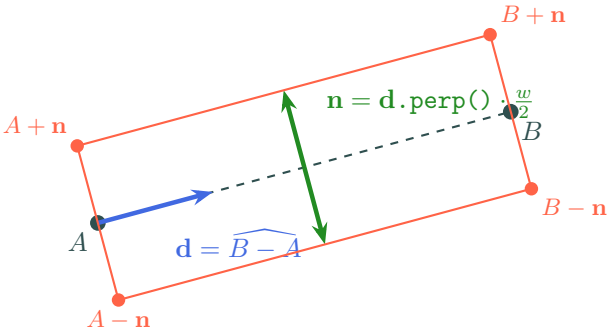
1. which way does the line run? — normalize $B - A$

Building the Rectangle: perp Earns Its Keep



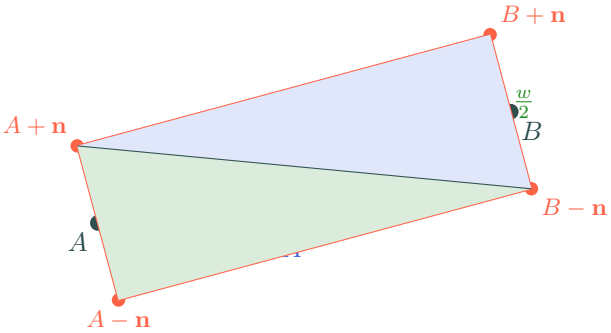
2. which way is *across*? — Monday's quarter-turn, scaled to half the width

Building the Rectangle: perp Earns Its Keep



3. step off both sides of both ends: four corners

Building the Rectangle: perp Earns Its Keep



4. fill as **two triangles** (`painter.fill_triangle`, given) — done.

Guards: $w \leq 0$ or $A = B \Rightarrow$ draw nothing (a zero vector has no `perp` — Monday's trap).

Every Stroked Path You've Ever Seen

This "line $\rightarrow$ rectangle $\rightarrow$ triangles" move is not a classroom hack:

- every vector renderer — browsers, PDF viewers, Figma, font engines — strokes paths *exactly* this way (their extra care goes into **joins and caps**, which we skip)
- GPUs can *only* fill triangles — Week 10 onward, "turn it into triangles" becomes the answer to *everything*
- today: your curves get weight and confidence — thick polylines = `thick_line` per segment

Motion: Time Is Just Another Dial

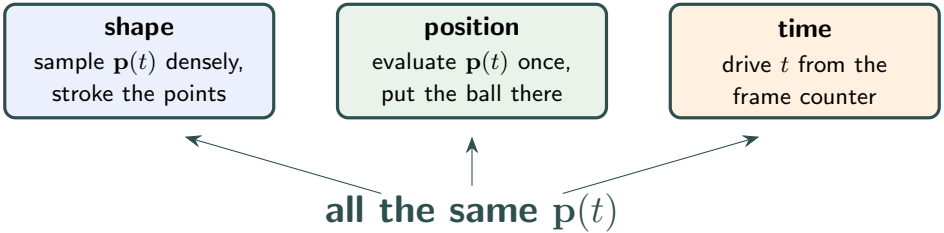
The Fifteen Lines Behind Every Animation

```
FRAMES = 90

def render_frame(i):
    u = i / (FRAMES - 1)    # 0.0 -> 1.0
    img = Canvas(W, H, SKY)
    p = line_point(a, b, u)  # where NOW?
    painter.fill_circle(img,
        round(p.x), round(p.y), 7, GOLD)
    return img

for i in range(FRAMES):
    render_frame(i).save(
        f"frames/frame_{i:04d}.ppm")
```

- the frame index becomes **the dial**: $u = \frac{i}{\text{FRAMES}-1}$
- the same $\mathbf{p}(t)$ that *drew* the path now *moves* a ball along it
- ffmpeg glues the frames into video

`frame_0007.ppm`, *four* digits — unpadded names sort 1, 10, 11, 2 and shuffle your movie.

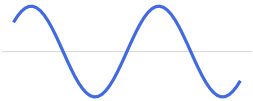
One Abstraction, Three Costumes



Compose them: a moon *orbiting* (`circle_point`, $\theta = 2\pi t$)
a planet that is itself *sliding* (`line_point`) — add the vectors, done.

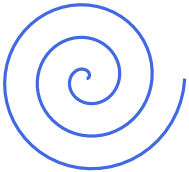
That two-liner is a strong skeleton for any animation. Seed your randomness; craft does the rest.

Anything You Can Type, You Can Draw



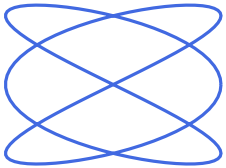
`(t, sin t)`

wave



`t*(cos 6πt, sin 6πt)`

spiral



`(cos 3t, sin 2t)`

Lissajous

Each is **one** `lambda` handed to the pipeline you built today.
Ten minutes of playing with $\mathbf{p}(t)$ will teach you more than an hour of reading — go break things.

Your Turn

Today's Toolkit — and Each One's Trap

function	what	the trap
`sample_parametric`	$\mathbf{p}(t)$ to a list of points	`steps+1` points, ends included
`circle_point`, `ellipse_point`	rim points	θ is a dial, not the rim angle
`draw_polyline`	points to pixels	`round()`, never `int()`
`thick_line`	a filled rectangle	$A = B$ has no `perp`: draw nothing

- `sample_parametric` + `draw_polyline` = `draw_curve`: any $\mathbf{p}(t)$ to pixels
- `thick_line` leans on Monday: `normalized`, `perp`, and the zero-vector guard
- Friday adds the last tool: Bézier curves, the same dial bending

Before Friday

1. Work these **by hand**, no computer:
   - $A = (0, 0)$, $B = (8, 4)$: find $\mathbf{p}(0.25)$
   - `sample_parametric(f, 0, 1, steps=10)` — how many points, and what is the last one?
   - `circle_point`(C, r, θ) for $\theta = 0, \frac{\pi}{2}, \pi$ — *on screen* (y down), which way does the point travel?
2. Sketch a **steep** thick line and a **zero-length** one — what should each draw, and why?
3. Bring a curve idea you would like to see *move* — Friday adds the last tool (Bézier)

Exit Ticket

Today's questions *(2 minutes)*

(a) $\mathbf{p}(t) = A + t\,(B - A)$ with $A = (2, 3)$, $B = (10, 7)$. Where is $\mathbf{p}(0.5)$?

(b) `sample_parametric(f, 0, 1, steps=4)` returns how many points — and why that many?

CS 453 — Exit Ticket #5

Name: _____

Roll no: _____

(a) _____

(b) _____

hand in on your way out

Friday: curves that *bend* where you tell them to — Bézier curves and de Casteljau's beautiful trick.