Assignment 2: Vectors, Curves, Motion

Weightage: ***3.6%*** (1/11 of the 40% assignment component)

Deadline: Friday 18^{th} September 2026, **11:59 pm**

Introduction

In A1 you decided the color of pixels one at a time. That is the bottom of the stack, and you will never have to wonder what is underneath it again. This week you move one level up: you stop asking *"what color is this pixel?"* and start asking *"where is this thing, and which way is it pointing?"* The answers are vectors, and the tool that extracts almost every useful fact from them is the dot product.

Then curves. A circle, an ellipse and a Bézier are all the same kind of object — a function from a number to a point — and once you can sample such a function you can draw all of them with one piece of code. Finally, because a function of a parameter is exactly what an animator needs, you will let that parameter be **time**, render a few hundred frames, and end the week holding a video you computed from arithmetic.

Objectives

- Building a small vector library and using it instead of loose x/y arithmetic
- Reading the dot product geometrically: length, angle, projection, reflection, and the closest point on a segment
- Treating a curve as a function $\mathbf{p}(t)$, and drawing anything you can write down as one
- Evaluating Bézier curves at any degree with de Casteljau's algorithm
- Turning a still renderer into an animation by making one number depend on time

Collaboration and AI Policy

All work is individual. Students are NOT permitted to look at or copy each other's code, code structure, or animations. If caught, both parties receive a 0 in A2 and the case is reported.

Per the course policy: AI assistants may be used to *explain* concepts and to *debug code you have already written*, but not to generate solutions or animations. Whatever you use, disclose it in the header of `vec2.py`, `curves.py` and `anim.py`. You must be able to explain, and modify on request, any line you submit — spot checks happen.

Getting Started

A) File Setup

Download the starter folder and keep **all files in one directory**:

`vec2.py`	your vector library — **T1–T7 go here**
`curves.py`	your curves and stroking — **T8–T12 go here**
`anim.py`	your animation — **all Part B work goes here**
`painter.py`	**given, working** — the A1 toolkit; do not edit
`canvas.py`	given — the `Canvas`/`Layer` classes; do not edit
`encode.py`	given — turns your frames into a video
`check.py`	the autograder; run it as often as you like
`expected_a2.py`	reference data used by `check.py` — do not edit

Note that `painter.py` is given to you, complete and working. It is the reference implementation of A1. Nobody is blocked this week by an A1 bug, and everyone draws through the same rasterizer, so the checker's expected pixels are meaningful. Read it and compare it against what you submitted last week — quietly, for your own benefit. You are not graded on it and you must not edit it.

As in A1, there is **nothing to install**: A2 is standard-library Python throughout, so no `pip`, no virtual environment, and no third-party packages. Any Python 3.9+ interpreter will do, **PyPy** included — and PyPy is worth considering here, because Part B renders a hundred-odd frames and its JIT makes pixel loops several times faster.

The one external tool A2 needs is `ffmpeg`, which is a standalone program rather than a Python package. That is the next section, and it is worth doing first.

B) ffmpeg — do this *before* you write any code

Part B ends with a video file, and the program that assembles video from still images is **ffmpeg**. It is not a Python package: `pip install ffmpeg` will not give you the tool you need. It is a standalone program you download once.

Step 1 — get the binary.

Windows	https://www.gyan.dev/ffmpeg/builds/ download `ffmpeg-release-essentials.zip`, unzip, and find `ffmpeg.exe` inside its `bin/` folder
macOS	https://evermeet.cx/ffmpeg/ (or `brew install ffmpeg` if you have Homebrew)
Linux	`sudo apt install ffmpeg` (or your distro's equivalent)

Step 2 — put it where the script can find it. Either add it to your PATH, or — much simpler — drop the binary into your A2 folder, right next to `encode.py`. The script looks in both places, and also accepts an explicit path via `--ffmpeg`.

Step 3 — prove it works, now:

```
python3 encode.py --check
```

When that prints `ffmpeg is ready`, you are done thinking about ffmpeg for the rest of the semester. **Do not leave this until the night before the deadline** — an ffmpeg problem discovered at 2am is an ffmpeg problem you will not solve by 3am, and "I could not install ffmpeg" is not an extension.

C) The Vector Model

Everything in A2 works with the same tiny data type:

- A **vector** is a `Vec2`: a small class holding two floats, `.x` and `.y`. It also unpacks and indexes like a pair, so `x, y = v` and `v[0]` both work.
- The same type represents a **point** (a location) and a **displacement** (an arrow). The difference lives in your head and in your naming, not in the type.
- A `Vec2` is **immutable**: every operation returns a **new** `Vec2` and nothing is modified in place.
- Screen convention is unchanged from A1: x grows right, y grows **down**, origin top-left.

Part A rule: standard library only. No third-party packages in `vec2.py` or `curves.py`. Coordinates stay *floats* until the very last moment, when a drawing call rounds them to pixels — rounding early is the most common way to make a smooth curve look lumpy. Do not change any function signature.

That y-down convention has one consequence worth stating plainly, because it will confuse you at least once: `Vec2(x, y).perp() = Vec2(-y, x)` is a *counter-clockwise* quarter turn in ordinary math axes, but on screen — where y points down — it appears **clockwise**. It is the same flip Lecture 5 pointed out for `circle_point`, where growing θ travels clockwise on screen. The algebra is right either way. Only your intuition needs adjusting.

The `Vec2` API at a glance. The left column is **given** and already works; the right column is **yours**, T1–T7. Everything in the right column raises `NotImplementedError` until you write it.

Given (plumbing)	Yours (T1–T7, 40 pts)
`Vec2(x, y)`	`a + b` `a - b`
`v.x` `v.y`	`v * k` `k * v` `-v`
`x, y = v` `v[0]`	`a.dot(b)` `a.cross(b)`
`v.rounded()`	`v.length()` `abs(v)`
`Vec2.zero()`	`v.normalized()` `v.perp()`
`a == b` `repr(v)`	`a.angle_between(b)`
	`a.project(b)` `d.reflect(n)`
	`p.closest_point_on_segment(a, b)`
	`p.distance_to_segment(a, b)`

`v.rounded()` is the bridge to the rasterizer: it returns a plain `(int, int)` pair, so `painter.fill_circle(img, *p.rounded(), r, color)` is the idiom for drawing at a computed position. It is exactly the `round(p.x), round(p.y)` you saw in lecture code, in one call. Round there and nowhere earlier.

How `a + b` finds your code. Lecture 4 showed `Vec2` arithmetic being *used*; the handout is where you learn how it is *wired*. Python has no idea what `+` means for a class you wrote, so it looks for a method with a special name and calls that. The right-hand column of the table above is the list of those names:

You write	Python calls	
`a + b`	`a.__add__(b)`	
`a - b`	`a.__sub__(b)`	
`v * 2`	`v.__mul__(2)`	
`2 * v`	`v.__rmul__(2)`	(the *r* is for "right operand")
`-v`	`v.__neg__()`	
`abs(v)`	`v.__abs__()`	

Each of these builds and **returns a new** `Vec2` (or a float, for `abs`) from `self.x`, `self.y` and the argument. None of them assigns to `self.x` — that is what "immutable" means in practice. Once `__add__` works, the rest are the same shape with a different arithmetic line.

Part A — Vectors and Curves (70 pts)

Implement T1–T7 in `vec2.py` and T8–T12 in `curves.py`. Each function's docstring in the starter repeats its exact contract. Check yourself anytime:

```
python3 check.py
```

The checker runs two kinds of case. **Value cases** compare what your functions *return*, within a tolerance of 10^{-6} — floats are never exactly equal, so we never ask them to be. **Pixel cases** compare what your functions *draw*, exactly, as in A1. **Grading runs additional hidden cases**: passing the checker is necessary, not sufficient — hardcoding its expected values earns a 0 for the task.

T1–T7 below are written in `vec2.py`. Open that file now — the stubs are in the same order as the tasks here, and each docstring repeats the contract you are reading. Nothing in this file draws anything or imports anything beyond `math`: it is pure arithmetic.

T1 · `+`, `-`, `*` · 4 pts

Componentwise sum, difference and scaling by a number, written as the operators `__add__`, `__sub__`, `__mul__`, `__rmul__` and `__neg__` so that `a + b`, `a - b`, `v * 2`, `2 * v` and `-v` all read the way the mathematics does. Each is one line, and every remaining task in the assignment is built from them.

Note there is deliberately no `Vec2 * Vec2`: multiplying two vectors has no single sensible meaning, which is why `dot` and `cross` exist as separate, differently-named operations.

Read `a - b` as *"the arrow from `b` to `a`"*. Nearly every geometric question below opens by subtracting two points to get a direction, so it is worth fixing that reading now.

T2 · `dot`, `length`, `normalized` · 6 pts

$$\mathbf{a}\cdot\mathbf{b} = a_x b_x + a_y b_y \qquad |\mathbf{a}| = \sqrt{\mathbf{a}\cdot\mathbf{a}} \qquad \hat{\mathbf{a}} = \frac{\mathbf{a}}{|\mathbf{a}|}$$

The dot product is the most useful single number in graphics, and you can often use it without computing an angle at all — *its sign is already an answer*, as Figure 1 shows.

`normalized` must survive the **zero vector**: it has no direction, so return `Vec2(0.0, 0.0)` rather than dividing by zero. Guard with `EPSILON`, not `== 0.0`. This is not a hypothetical — you will write `(p - q).normalized()` in Part B, and sooner or later `p` will equal `q`.

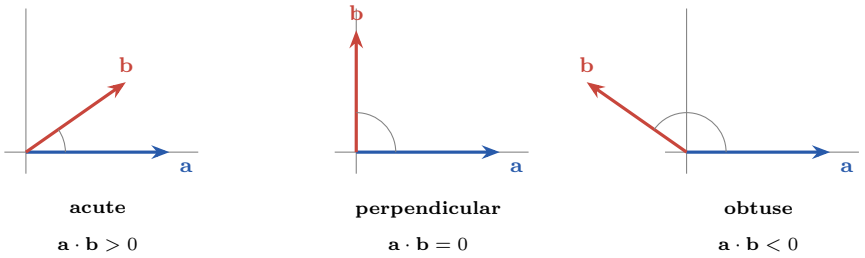

Figure 1: The sign of the dot product is a question already answered: are these two pointing roughly the same way? You rarely need `angle_between` to find out — and a sign test is both faster and free of the rounding traps that `acos` brings.

The name is past tense, like Python's `sorted`, because it returns a new vector rather than changing this one. `abs(v)` is also yours to implement, as a one-liner returning the length.

T3 · `perp` and `cross` · 5 pts

`Vec2(x, y).perp()` $= \texttt{Vec2}(-y, x)$ rotates by a quarter turn. Verify it yourself: `a.dot(a.perp())` must be 0 for every `a`.

In 2D the cross product is a single *number*, not a vector:

$$\mathbf{a} \times \mathbf{b} = a_x b_y - a_y b_x$$

It is the signed area of the parallelogram spanned by the two vectors, which makes it a **side test**: its sign tells you which way **b** turns relative to **a**, and zero means they are parallel. You will meet it again in the ray tracer, and in Lecture 23 it becomes the basis of barycentric coordinates.

T4 · `angle_between` · 5 pts

The unsigned angle in radians, in $[0, \pi]$, from

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}|\,|\mathbf{b}|}$$

Two traps, both tested by the checker. A zero-length vector has no defined angle — return `0.0`. And floating-point error can make that quotient come out as `1.0000000000000002`, which sends `math.acos` straight into a `ValueError`. **Clamp to $[-1, 1]$ before calling `acos`.** This is not defensive paranoia; it happens for ordinary inputs like `Vec2(2,2).angle_between(Vec2(4,4))`.

T5 · `project` · 6 pts

The part of **a** that lies along **b** — **a**'s shadow on **b**:

$$\text{proj}_{\mathbf{b}}(\mathbf{a}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}}\,\mathbf{b}$$

The result is a *vector* parallel to **b**, not a length. If **b** is the zero vector there is nothing to project onto: return `Vec2(0.0, 0.0)`.

Figure 2 is worth studying, because the next two tasks are both this picture with one extra step.

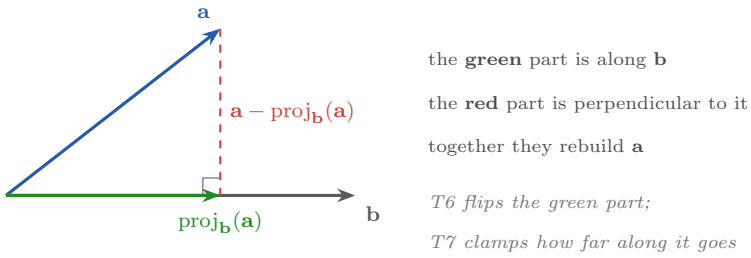

Figure 2: Projection splits **a** into a part along **b** and a part perpendicular to it. Almost every geometry question in this assignment is that split, used slightly differently.

T6 · `reflect` · 7 pts

Reflect a direction **d** off a surface with **unit** normal **n**:

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\,\mathbf{n}$$

You may assume **n** is already normalized — the caller does it. The convention is a ball bouncing off a wall: **d** points *into* the surface, **r** comes back out.

Rather than memorizing the formula, derive it from Figure 2: split **d** into its along-**n** part and its perpendicular part. A mirror keeps the perpendicular part and flips the other one, and subtracting *twice* the along-**n** part does exactly that.

Looking ahead: you will write this identical line, unchanged except for a third component, when metal surfaces start bouncing light in Lecture 17. Understanding it once here saves you a week later.

T7 · `closest_point_on_segment` · 7 pts

This one was not in lecture. It is Lecture 4's "every vector splits in two" picture with one extra step, so read this section slowly and it will be familiar by the end.

Given a point **p** and a **segment** from **a** to **b**, find the point on that segment nearest to **p**, and (in `distance_to_segment`) how far away it is.

Step 1 — move the problem to a. Two arrows, both starting at **a**: the segment itself, $\mathbf{b} - \mathbf{a}$, and the arrow to the point, $\mathbf{p} - \mathbf{a}$.

Step 2 — project. Split $\mathbf{p} - \mathbf{a}$ into its *along* part (along the segment) and its *across* part. The closest point is where the along part ends, and the across part *is* the distance. Rather than the projected vector, keep the **fraction** of the way along the segment:

$$t = \frac{(\mathbf{p} - \mathbf{a}) \cdot (\mathbf{b} - \mathbf{a})}{(\mathbf{b} - \mathbf{a}) \cdot (\mathbf{b} - \mathbf{a})}$$

This is T5's formula with the final "$\cdot\,\mathbf{b}$" left off: $t = 0$ means the foot of the perpendicular lands on **a**, $t = 1$ on **b**, $t = 0.5$ halfway.

Step 3 — clamp. On an infinite *line* you would stop at step 2. On a *segment* you must **clamp** t **to** $[0, 1]$, or your "closest point" sails off past an endpoint — see Figure 3. Same one-liner as T4: `t = max(0.0, min(1.0, t))`.

Step 4 — walk. The answer is $\mathbf{a} + t(\mathbf{b} - \mathbf{a})$ — Monday's lerp. The distance is then $|\mathbf{p} - \text{closest}|$, one subtraction and one `length`.

A degenerate segment ($\mathbf{a} = \mathbf{b}$) has exactly one point: return **a**. Test for it with the `EPSILON` idea from T2 on $(\mathbf{b} - \mathbf{a}) \cdot (\mathbf{b} - \mathbf{a})$, which is the denominator above — that is the division you are guarding.

Worked, by hand. Segment $\mathbf{a} = (2, 1)$ to $\mathbf{b} = (8, 1)$, so $\mathbf{b} - \mathbf{a} = (6, 0)$ and $(\mathbf{b} - \mathbf{a}) \cdot (\mathbf{b} - \mathbf{a}) = 36$.

$\mathbf{p}$	$\mathbf{p} - \mathbf{a}$	t (raw $\to$ clamped)	closest	distance
$(7, 3)$	$(5, 2)$	$30/36 = 0.83 \to 0.83$	$(2, 1) + 0.83\,(6, 0) = (7, 1)$	$\lvert(0, 2)\rvert = 2$
$(12, 4)$	$(10, 3)$	$60/36 = 1.67 \to \mathbf{1}$	$(8, 1)$, the endpoint	$\lvert(4, 3)\rvert = 5$
$(0, 1)$	$(-2, 0)$	$-12/36 = -0.33 \to \mathbf{0}$	$(2, 1)$, the endpoint	$\lvert(-2, 0)\rvert = 2$

In pseudocode, the whole function is five lines — and every one of them is a `Vec2` operation you have already written:

```
function closest_point_on_segment(p, a, b):
    seg = b - a                          # T1
    denom = seg.dot(seg)                 # T2: |seg|^2, no square root
    if denom < EPSILON: return a         # degenerate: one point only
    t = (p - a).dot(seg) / denom         # T5's fraction
    t = clamp(t, 0, 1)                   # segment, not line
    return a + seg * t                   # lerp
```

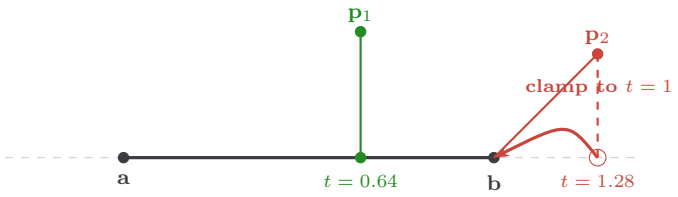

Figure 3: Clamping is the whole difference between a segment and a line. For $\mathbf{p}_1$ the foot of the perpendicular lands on the segment and is the answer. For $\mathbf{p}_2$ it lands past $\mathbf{b}$, so the nearest point on the *segment* is the endpoint $\mathbf{b}$ itself.

Looking ahead: this is a distance function, and in Lecture 33 you will build entire images out of nothing but distance functions (SDFs). It is also how you would hit-test a click against a line in any drawing program.

`vec2.py` **is done — T8–T12 below are written in** `curves.py`**.** That file already imports your `vec2` and the given `painter`, and this is where the two meet: `curves.py` uses vectors to decide *which points*, then hands them to `painter` to decide *which pixels*. Keep using your own `Vec2` operators here — if you are writing `Vec2(a.x + b.x, a.y + b.y)` by hand, you are re-implementing T1.

T8 · `sample_parametric` · 5 pts

Here is the idea the rest of the assignment rests on. A curve is a **function from a parameter to a point**, $\mathbf{p}(t)$. To draw one, ask it for points at a series of t values and connect them with straight lines. If the points are close enough together, the eye sees a curve.

The call `sample_parametric(f, t0, t1, steps)` returns the list $[f(t_0), \ldots, f(t_1)]$, **inclusive of both ends**. So `steps` counts the *gaps* between samples, and you return `steps + 1` points — the same inclusive convention as A1's `vertical_gradient`. When `steps` is less than 1, return just the single point `[f(t0)]`.

Note what this function does *not* know: what kind of curve it is sampling. Circles, ellipses and Béziers all reduce to a different `f` handed to this one function. That is the whole payoff of thinking parametrically.

T9 · `circle_point` and `ellipse_point` · 5 pts

$$\text{circle: } (c_x + r\cos\theta,\ c_y + r\sin\theta) \qquad \text{ellipse: } (c_x + r_x\cos\theta,\ c_y + r_y\sin\theta)$$

In A1 you drew a filled circle with an inside test over a bounding box. This is the other way to think about circles, and it gives you the *outline* almost for free — hand `circle_point` to `sample_parametric` over $[0, 2\pi]$ and stroke the result.

One honest subtlety: for an ellipse, θ is the **parameter**, not the geometric angle from the center to the point (they agree only when $r_x = r_y$). Equal steps in θ therefore do *not* give equal steps along the rim — samples bunch up near the ends of the long axis. That is fine for drawing, and it is exactly the sort of thing that quietly breaks naive attempts to move an object at constant speed around an ellipse.

T10 · `de_casteljau` · 10 pts

The heart of the assignment. A Bézier curve is defined by control points, and de Casteljau's algorithm evaluates it using nothing but repeated linear interpolation:

1. Between each consecutive *pair* of control points, take the point a fraction t of the way along: $(1 - t)\mathbf{P}_i + t\mathbf{P}_{i+1}$. So n points go in and $n - 1$ come out.
2. Repeat on the new, shorter list.
3. When a single point remains, that point is on the curve.

Figure 4 traces one full evaluation of a cubic at $t = 0.35$.

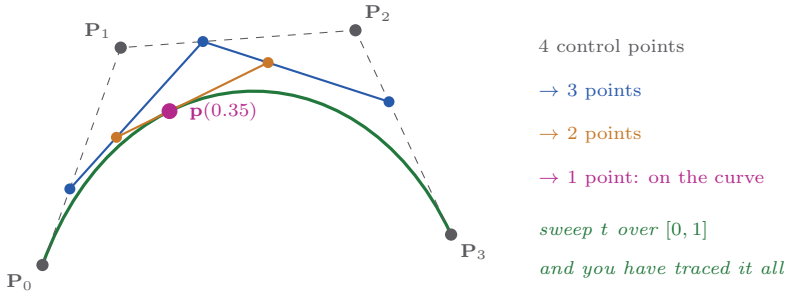

Figure 4: De Casteljau at $t = 0.35$ on a cubic. Each round replaces the list of points by the interpolated points between consecutive pairs. The lone survivor is $\mathbf{p}(0.35)$. Nothing here is specific to cubics — the same loop handles any number of control points.

Implement it as a loop over lists, not as a hardcoded cubic formula. Done properly, one short function handles 2, 4 or 40 control points with no special cases — and the checker tests degrees 0 through 4 precisely to see whether you did. `quadratic_bezier` and `cubic_bezier` are then one line each.

Two properties worth knowing, both checked: the curve always passes *exactly* through its first and last control points ($t = 0$ and $t = 1$), and it generally does *not* pass through the interior ones — they pull on it without being touched.

T11 · `draw_polyline` · 5 pts

Connect consecutive points with `painter.bresenham_line`; with `closed=True`, also join the last point back to the first. Fewer than two points draws nothing.

Your points are floats and Bresenham needs ints, so this is where rounding finally happens. Use `round`, **not** `int()`: `int()` truncates toward zero, which biases every coordinate the same direction and visibly warps the curve. Round *here*, at the last possible moment, so everything upstream keeps full precision.

Combine T8 and T11 and you can draw any curve you can write down. That combination is common enough that the starter gives it to you as `draw_curve`.

T12 · `thick_line` · 5 pts

Bresenham lines are exactly one pixel wide. To draw a thick one, stop thinking "line" and start thinking "rectangle":

$$\begin{aligned} \mathbf{d} &= \text{normalize}(\mathbf{b}-\mathbf{a}) \\ \mathbf{n} &= \text{perp}(\mathbf{d}) \cdot \frac{\text{width}}{2} \\ \text{corners} &\quad \mathbf{a}+\mathbf{n},\ \mathbf{b}+\mathbf{n},\ \mathbf{b}-\mathbf{n},\ \mathbf{a}-\mathbf{n} \end{aligned}$$

Fill that quadrilateral as **two triangles** using `painter.fill_triangle` (given): $(\mathbf{a}{+}\mathbf{n}, \mathbf{b}{+}\mathbf{n}, \mathbf{b}{-}\mathbf{n})$ and $(\mathbf{a}{+}\mathbf{n}, \mathbf{b}{-}\mathbf{n}, \mathbf{a}{-}\mathbf{n})$. Draw nothing if `width` ≤ 0 or if $\mathbf{a} = \mathbf{b}$ — a zero direction has no perpendicular.

This is the first time you turn a mathematical object into geometry a rasterizer can eat, and it is a real technique, not a classroom exercise: every stroked path in every vector renderer is a more careful version of exactly this (the care goes into joins and caps, which we are skipping).

Part B — Your Animation (30 pts)

Using your toolkit, design and render an animation in `anim.py` as a sequence of frames, then encode it to `anim.mp4`.

The big idea is smaller than it sounds. An animation is the still image you already know how to make, rendered once per frame, with one number changed. That number is time. Your `render_frame(i)` computes

$$u = \frac{i}{\text{FRAMES} - 1} \in [0, 1]$$

and every moving quantity is derived from u. Derive from u, never from `i` directly — then changing `FRAMES` changes the *smoothness* of your animation, not its choreography.

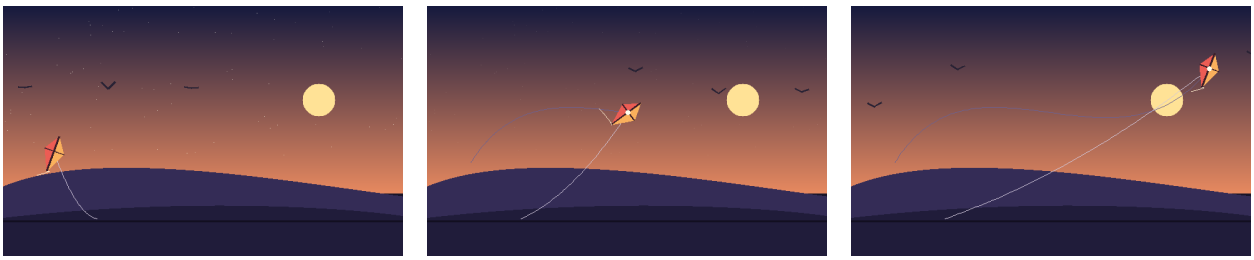

Figure 5: Three frames from the instructor exemplar, "Kite over the Hills" (frames 12, 58 and 104 of 120). The kite travels a cubic Bézier and *banks into its heading* — the lean comes from `angle_between` the heading and vertical, with `cross` supplying the sign. The hills are Bézier silhouettes, the string is a quadratic that sags, and the birds keep clear of the string using `distance_to_segment`. Your animation should be *yours*, not this one.

Workflow — follow this order

1. `python3 encode.py --check` — prove ffmpeg works *before* you animate anything.
2. Set `FRAMES = 1` and get a **single frame** looking right. Iterating on one frame is seconds; iterating on 120 is minutes.
3. Raise `FRAMES`, render, encode, watch it. Then adjust.

```
python3 anim.py        # writes frames/frame_0000.ppm, frame_0001.ppm, ...
python3 encode.py      # combines them into anim.mp4
```

`anim.py` clears stale frames each run — otherwise a shorter run leaves the tail of a longer one behind and ffmpeg cheerfully encodes frames you no longer want.

If your render is slow

It should not be. A 90-frame render finishes in well under a second; if yours takes a minute, you are almost certainly **repainting your background every frame**.

Remember what `painter.set_pixel` really is: one pixel, one Python function call. A 640×400 gradient is 256,000 of them, and doing that once per frame across 90 frames is 23 million calls spent redrawing an image that is *identical every time*.

The starter shows the fix. Anything that does not depend on `u` is painted once into a module-level `BACKGROUND`, and each frame starts from a copy:

```
BACKGROUND = make_background()      # painted ONCE, at import

def render_frame(i):
    img = BACKGROUND.copy()         # not a repaint - a copy
    ...
```

Note the `.copy()`. Draw onto `BACKGROUND` directly and every frame accumulates the previous one on top of it.

The test is not "is it background?" but "does it depend on `u`?" A gradient sky and a static path are cacheable; stars that fade as the animation advances are not — those still belong in `render_frame`. Put your skyline, starfield or texture wherever that question puts it, and your render stays fast enough to iterate on.

B1 · Compliance Checklist · 16 pts

Binary items, checked by the grader exactly as written. The checklist also appears as comments in the starter `anim.py`.

Item	Pts
`python3 anim.py` runs from a clean folder and writes `frames/frame_0000.ppm` ...	2
At least 60 frames; canvas at least 480×320	1
`python3 encode.py` produces a playable `anim.mp4` from your frames	2
Something **moves along a curve** you evaluate (Bézier or parametric)	3
At least one use of `de_casteljau` / `quadratic_bezier` / `cubic_bezier`	2
At least one `thick_line` stroke	2
A dot-product function (`angle_between` / `project` / `reflect` / `closest_point_on_segment`) that **affects what is drawn**, marked with a comment	3
Any randomness is seeded — the same movie on every run	1
Total	**16**

That dot-product item is the one students try to fake. A call whose result is computed and then ignored scores 0. It must change something visible — an orientation, a distance, a color, a position.

B2 · Craft Rubric · 14 pts

Criterion	0 — absent	1 — attempted	2 — competent	3 — excellent
Motion quality (0–4, doubled at 2)	Objects teleport or jitter; motion unreadable	Movement happens but is uniformly linear and abrupt	Smooth, continuous motion; easing or varied pacing used deliberately	Motion has weight and intent — anticipation, follow-through, secondary movement
Composition & design	Elements scattered; no framing	Some structure; foreground and background undifferentiated	Clear staging; the moving element is easy to follow	Deliberate composition that leads the eye; the frame stays balanced *throughout* the motion
Technical ambition	Bare-minimum checklist calls	Primitives combined into one compound object	Several compound objects, or a custom helper built atop the toolkit	Beyond the spec: layered depth, procedural variation, curve-driven orientation, trails
Originality & polish	Reproduces the starter example or the handout exemplar	Derivative with small changes	Clearly original; minor artifacts (popping, gaps, stray pixels)	Original and artifact-free; the loop or ending is considered

Motion quality is scored 0–4 (it is what separates an animation from a slideshow); the other three are 0–3. Total 14.

Calibration: *competent* on every axis earns 9/14 — a solid outcome. Top marks are reserved for work that would headline the class gallery. Submitting the starter's travelling dot, lightly recolored, caps *Originality* at 1.

Two cheap upgrades, both one line. First, easing: pass your parameter through the given `ease_in_out` so motion starts and stops gently instead of snapping. Second, orientation: sample your curve slightly ahead of the current t, subtract to get a heading, and rotate your object to face it. Objects that turn to face where they are going stop looking like stickers sliding across glass.

Submission

Submit **one zip file** named `RollNo_A2.zip` (e.g. `231234567_A2.zip`) on the course website, containing exactly:

`vec2.py` `curves.py` `anim.py` `anim.mp4`

Do **not** include any of the following: the `frames/` folder (it is hundreds of PNGs), `painter.py`, `encode.py`, `check.py`, `expected_a2.py`, `__pycache__`, the ffmpeg binary, or the given files (`canvas.py`, `painter.py`, `check.py`, `expected_a2.py`, `encode.py`) — we run your submission against our own copies. All three `.py` files must have the header block (name, roll number, AI disclosure) filled in.

If `anim.mp4` exceeds the upload limit, reduce your canvas size or frame count and re-encode — do not submit a partial video.

The deadline is hard: no late submissions, per the course outline. A missed deadline scores zero. Documented medical or university grounds are the only exception.

Appendix A: The Dot Product, Five Ways

Every one of these is the same operation. Recognizing which reading applies is most of the skill.

You want to know	Use
Are these pointing the same way?	the **sign** of $\mathbf{a} \cdot \mathbf{b}$ — no angle needed
How long is this vector?	$\lvert\mathbf{a}\rvert = \sqrt{\mathbf{a} \cdot \mathbf{a}}$
What is the angle between them?	$\theta = \arccos\left(\dfrac{\mathbf{a} \cdot \mathbf{b}}{\lvert\mathbf{a}\rvert\lvert\mathbf{b}\rvert}\right)$, *clamped*
How much of $\mathbf{a}$ lies along $\mathbf{b}$?	$\text{proj}_{\mathbf{b}}(\mathbf{a}) = \dfrac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}}\mathbf{b}$
How do I bounce off this surface?	$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$

When a graphics problem looks geometric and you do not know where to start, ask what you would do with a dot product. It is right more often than it has any business being.

Appendix B: De Casteljau in Pseudocode

The algorithm from Lecture 6. Note there is no mention of degree anywhere — the loop simply runs until one point is left.

```
function de_casteljau(control_points, t):
    points = copy of control_points
    while length(points) > 1:
        next = empty list
        for i from 0 to length(points) - 2:
            # the point a fraction t of the way from points[i]
            # to points[i+1]
            q = (1 - t) * points[i]  +  t * points[i+1]
            append q to next
        points = next
    return points[0]
```

The multiplication and addition there are *vector* operations — use your `Vec2` operators rather than writing the components out. In Python that line is simply `points[i] * (1 - t) + points[i+1] * t`. If you find yourself touching `.x` and `.y` inside this function, you are re-implementing T1.

Appendix C: An Animation in Fifteen Lines

The skeleton every animation in this assignment shares. Everything interesting lives in `render_frame`; the loop around it never changes.

```
FRAMES = 90

def render_frame(i):
    u = i / max(FRAMES - 1, 1)      # 0.0 -> 1.0 over the whole animation
    img = Canvas(WIDTH, HEIGHT, SKY)
```

```
        t = curves.ease_in_out(u)              # gentle start and stop
        p = curves.cubic_bezier(a, b, c, d, t) # where the object is NOW

        painter.fill_circle(img, *p.rounded(), 7, GOLD)
        return img

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

The zero-padded filename (`frame_0007.ppm`, not `frame_7.ppm`) is not cosmetic: ffmpeg reads the sequence in the order the pattern `frame_%04d.ppm` implies, and unpadded names sort as `1, 10, 11, 2`, which would shuffle your animation. `encode.py` expects exactly four digits.