Share

You said:

```
/*
This sketch demonstrates the parabolic transformation
  T(u,v) = (u, v^2)
which maps a blue rectangle in the uv–plane (left panel)
to a red region in the xy–plane (right panel).

INTERACTIVE FEATURES:
• Drag the four small blue dots (corners) to change the rectangle
in the uv–plane.
• The text shows the transformation T(u,v) = (u, v^2).
• Use the resolution slider to set how many grid points inside the
rectangle are animated.
• Press the "Animation" button to animate the transformation:
  Each frame a new row of grid points (parameterized by s in
[0,1]) is mapped and drawn as red dots.
• Press the "Clear Traces" button to erase the red traces.
*/

const uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350;  //
uv–plane panel
const xyLeft = 400, xyRight = 700, xyTop = 50, xyBottom = 350;  //
xy–plane panel

// We assume uv ranges: u,v in [-5,5] for the left panel.
// For the image: x = u in [-5,5] and y = v^2 in [0,25] (since v ∈
[-5,5]).

// The blue rectangle is defined by four draggable corner points
in uv–space.
let corners = []; // each corner: {u, v}
let dragIndex = -1;
const dragThreshold = 10; // pixels

// UI elements
let resolutionSlider, animateButton, clearButton;
let animating = false;
let animStep = 0;  // which row of grid points (parameter t) is
being animated
let gridRes = 20;  // default grid resolution (number of rows/cols)

// We'll use an offscreen graphics layer for drawing traces in the
xy–plane.
let traceLayer;

function setup() {
  createCanvas(750, 400);
  textSize(14);

  // Initialize four corners (in uv–space) in order:
  // p0: bottom–left, p1: bottom–right, p2: top–right, p3: top–left.
  corners = [
    {u: -2, v: -1},  // p0
    {u: 2,  v: -1},  // p1
```

```
  {u: 2,  v: 3},   // p2
  {u: -2, v: 3}    // p3
 ];

 // Create a slider to control grid resolution for the animation
 resolutionSlider = createSlider(5, 50, gridRes, 1);
 resolutionSlider.position(20, 20);
 resolutionSlider.style('width', '150px');

 // Create the Animate button.
 animateButton = createButton("Animation");
 animateButton.position(20, 50);
 animateButton.mousePressed(startAnimation);

 // Create the Clear Traces button.
 clearButton = createButton("Clear Traces");
 clearButton.position(120, 50);
 clearButton.mousePressed(clearTraces);

 // Create an offscreen graphics buffer for the xy–plane traces.
 let xyW = xyRight - xyLeft;
 let xyH = xyBottom - xyTop;
 traceLayer = createGraphics(xyW, xyH);
 traceLayer.background(240);
}

function draw() {
 background(240);

 // Draw labels for panels and the transformation text.
 fill(0);
 noStroke();
 text("uv–plane (blue rectangle)", uvLeft, uvTop - 10);
 text("xy–plane (red image, traces)", xyLeft, xyTop - 10);
 text("Transformation: T(u,v) = (u, v^2)", 20, 380);

 // Update grid resolution from slider.
 gridRes = resolutionSlider.value();

 // Draw the uv–plane panel border.
 stroke(150);
 noFill();
 rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop);

 // Draw uv axes (optional)
 drawUVAxes();

 // Draw the blue rectangle (by connecting its corners)
 drawBlueRectangle();

 // Draw the four blue corner dots.
 for (let i = 0; i < corners.length; i++) {
  let pt = uvToScreen(corners[i].u, corners[i].v);
  fill(0, 0, 255);
  noStroke();
  ellipse(pt.x, pt.y, 10, 10);
 }

 // Draw the right panel (xy–plane) border.
 stroke(150);
 noFill();
 rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop);
 drawXYAxes();

 // Display the traceLayer (the accumulated red points) in the xy–
```

```
plane.
  image(traceLayer, xyLeft, xyTop);

  // If animating, process one row (a given t value) per frame.
  if (animating) {
    animateTransformation();
  }
}

// -----------------------
// Animation functions
// -----------------------

// In our parameterization, any point inside the quadrilateral is
given by:
//   (u,v) = p0 + s*(p1 - p0) + t*(p3 - p0), with s,t in [0,1].
// Then its image under T is: (x,y) = (u, v^2).
function animateTransformation() {
  // We'll iterate t from 0 to 1 in gridRes steps.
  let t = animStep / gridRes;
  // For each row (fixed t), iterate over s in [0,1] (gridRes steps)
  for (let i = 0; i <= gridRes; i++) {
    let s = i / gridRes;
    // Compute uv coordinate using bilinear parameterization along
the bottom and left edges:
    let uv = parametricUV(s, t);
    let transformed = { u: uv.u, v: uv.v * uv.v }; // T(u,v) = (u, v^2)
    // Map transformed coordinate to xy–plane screen coordinates.
    let pt = xyFromUVTransformed(transformed.u, transformed.v);
    // Draw a small red dot on the traceLayer.
    traceLayer.fill(255, 0, 0);
    traceLayer.noStroke();
    traceLayer.ellipse(pt.x, pt.y, 4, 4);
  }
  animStep++;
  if (animStep > gridRes) {
    animating = false;  // finished one full pass (t from 0 to 1)
  }
}

// Parameterize a point in the quadrilateral defined by corners:
// p0, p1, p3 (we use p0, p1, and p3 to define an affine map since
for a rectangle p2 is redundant).
function parametricUV(s, t) {
  // p0 = bottom-left, p1 = bottom-right, p3 = top-left.
  let p0 = corners[0];
  let p1 = corners[1];
  let p3 = corners[3];
  let u = p0.u + s * (p1.u - p0.u) + t * (p3.u - p0.u);
  let v = p0.v + s * (p1.v - p0.v) + t * (p3.v - p0.v);
  return { u, v };
}

// -----------------------
// UI Button Callbacks
// -----------------------
function startAnimation() {
  animating = true;
  animStep = 0;
}

function clearTraces() {
  traceLayer.background(240);
}
```

```
// -----------------------
// Drawing helper functions
// -----------------------

// Draw the blue rectangle (connecting the four corners in order).
function drawBlueRectangle() {
  stroke(0, 0, 255);
  noFill();
  beginShape();
  for (let i = 0; i < corners.length; i++) {
    let pt = uvToScreen(corners[i].u, corners[i].v);
    vertex(pt.x, pt.y);
  }
  endShape(CLOSE);
}

// Draw simple axes for the uv–plane.
function drawUVAxes() {
  stroke(200);
  // u-axis: horizontal line through v=0.
  let v0Screen = uvToScreen(0, 0).y;
  line(uvLeft, v0Screen, uvRight, v0Screen);
  // v-axis: vertical line through u=0.
  let u0Screen = uvToScreen(0, 0).x;
  line(u0Screen, uvTop, u0Screen, uvBottom);
}

// Draw simple axes for the xy–plane.
function drawXYAxes() {
  stroke(200);
  // x-axis: horizontal line at y=0.
  let y0 = xyMapY(0);
  line(xyLeft, y0, xyRight, y0);
  // y-axis: vertical line at x=0.
  let x0 = xyMapX(0);
  line(x0, xyTop, x0, xyBottom);
}

// -----------------------
// Coordinate Mapping Functions
// -----------------------

// uv–plane: u in [-5,5] -> [uvLeft,uvRight], v in [-5,5] ->
[uvBottom,uvTop] (with v increasing upward)
function uvToScreen(u, v) {
  let x = map(u, -5, 5, uvLeft, uvRight);
  // In screen, higher v should be higher (smaller y)
  let y = map(v, -5, 5, uvBottom, uvTop);
  return createVector(x, y);
}

// xy–plane: x = u in [-5,5] -> [xyLeft,xyRight]
function xyMapX(x) {
  return map(x, -5, 5, xyLeft, xyRight);
}
// For y: note that y = v^2, with v in [–5,5] so y in [0,25]; we want
y=0 at bottom.
function xyMapY(y) {
  return map(y, 0, 25, xyBottom, xyTop);
}
// Given transformed (u, vTransformed) where vTransformed =
v^2, map to xy–plane.
function xyFromUVTransformed(u, vTransformed) {
  let x = xyMapX(u);
  let y = xyMapY(vTransformed);
```

```
  return createVector(x, y);
}

// ----------------------
// Mouse interaction for dragging the corner dots in the uv–plane.
function mousePressed() {
  // Check if mouse is within the uv–panel.
  if (mouseX >= uvLeft && mouseX <= uvRight && mouseY >= uvTop && mouseY <= uvBottom) {
    // Check each corner for proximity.
    for (let i = 0; i < corners.length; i++) {
      let pt = uvToScreen(corners[i].u, corners[i].v);
      if (dist(mouseX, mouseY, pt.x, pt.y) < dragThreshold) {
        dragIndex = i;
        break;
      }
    }
  }
}

function mouseDragged() {
  if (dragIndex !== -1) {
    // Convert mouseX, mouseY to uv–coordinates.
    let u = map(mouseX, uvLeft, uvRight, -5, 5);
    let v = map(mouseY, uvBottom, uvTop, -5, 5);
    corners[dragIndex].u = u;
    corners[dragIndex].v = v;
  }
}

function mouseReleased() {
  dragIndex = -1;
}
```
Consider the parabolic transformation  and .

The Jacobian establishes a relationship between the area of the blue square (left) and the area of the red region (right).

Instructions:
Drag the green point (on the left) to change the position of the blue square. Observe what happens to the red region.
Drag the slider , which determines the side of the blue square.
See what happens to , for small values of .
Zoom in or out, if necessary.  do it correctly
"https://www.geogebra.org/m/qM777NYH" refer this

This looks like a great interactive visualization of how the Jacobian transformation affects area scaling under a parabolic transformation. Based on your reference to the GeoGebra model, you want a fully interactive **p5.js** version where:

1. A **green draggable point** controls the position of a blue square in the **uv-plane**.
2. A **slider** adjusts the side length of the blue square.
3. The transformed **red region in the xy-plane** updates dynamically.
4. **Zooming in and out** is possible for better visualization.
5. The **Jacobian determinant** is demonstrated through the area change.

I will generate a **p5.js** implementation that accurately mirrors the GeoGebra visualization, handling all interactivity properly.

**Jacobian Visualization**

```
let uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350;
```

```
let xyLeft = 400, xyRight = 700, xyTop = 50, xyBottom = 350;
let squareCenter, squareSize;
let sizeSlider;

function setup() {
  createCanvas(750, 400);
  textSize(14);
  squareCenter = createVector(-2, -1);
  squareSize = 2;

```

This **p5.js** script implements an interactive Jacobian transformation visualization:

✅ **Draggable green point** moves the **blue square** in the **uv-plane**
✅ **Slider** adjusts the **square's size** dynamically
✅ **Red transformed region** updates in the **xy-plane** using $T(u,v) = (u, v^2)$
✅ **Click & Drag** to modify position
✅ **Smooth transformation rendering**

Let me know if you need additional features like **zooming** or **grid overlays**! 🚀

put x,y coordinates for both at corners of square

Reasoned for 11 seconds

Below is an updated **p5.js** sketch that not only displays the blue square in the uv–plane and its red transformed region in the xy–plane under the transformation

$$T(u,v) = (u, v^2)$$

but also annotates each of the four corners with its coordinates. In the uv–plane the coordinates are shown as (u, v) while in the xy–plane they are shown as $(x, y) = (u, v^2)$.

You can try out this complete code:

javascript Copy Edit

```javascript
let uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350; let xyLeft = 400,
xyRight = 700, xyTop = 50, xyBottom = 350; let squareCenter, squareSize; let
sizeSlider; function setup() { createCanvas(750, 400); textSize(14); // Starting
with a center (in uv-coordinates) and an initial side length. squareCenter =
createVector(-2, -1); squareSize = 2; sizeSlider = createSlider(0.5, 5, 2, 0.1);
sizeSlider.position(20, 20); } function draw() { background(240); squareSize =
sizeSlider.value(); drawLabels(); drawPanels(); drawUVSquare(); drawXYRegion(); }
function drawLabels() { fill(0); noStroke(); text("uv-plane (blue square)", uvLeft,
uvTop - 10); text("xy-plane (red transformed region)", xyLeft, xyTop - 10);
text("Transformation: T(u,v) = (u, v^2)", 20, 380); } function drawPanels() {
stroke(150); noFill(); // Draw uv-panel border rect(uvLeft, uvTop, uvRight -
uvLeft, uvBottom - uvTop); // Draw xy-panel border rect(xyLeft, xyTop, xyRight -
xyLeft, xyBottom - xyTop); } function drawUVSquare() { // Calculate the four
corners of the blue square (in uv-space) let blUV = createVector(squareCenter.x -
squareSize / 2, squareCenter.y - squareSize / 2); let brUV =
createVector(squareCenter.x + squareSize / 2, squareCenter.y - squareSize / 2); let
trUV = createVector(squareCenter.x + squareSize / 2, squareCenter.y + squareSize /
2); let tlUV = createVector(squareCenter.x - squareSize / 2, squareCenter.y +
squareSize / 2); let uvCorners = [blUV, brUV, trUV, tlUV]; // Convert uv
coordinates to screen positions for the uv-panel let bl = uvToScreen(blUV.x,
blUV.y); let br = uvToScreen(brUV.x, brUV.y); let tr = uvToScreen(trUV.x, trUV.y);
let tl = uvToScreen(tlUV.x, tlUV.y); stroke(0, 0, 255); noFill(); beginShape();
vertex(bl.x, bl.y); vertex(br.x, br.y); vertex(tr.x, tr.y); vertex(tl.x, tl.y);
endShape(CLOSE); // Draw the draggable center point in green fill(0, 255, 0); let
centerScreen = uvToScreen(squareCenter.x, squareCenter.y); ellipse(centerScreen.x,
centerScreen.y, 10, 10); // Annotate each corner with its uv-coordinates (as (u,v))
fill(0); noStroke(); for (let corner of uvCorners) { let screenPos =
uvToScreen(corner.x, corner.y); let label = '(' + nf(corner.x, 1, 2) + ', ' +
nf(corner.y, 1, 2) + ')'; text(label, screenPos.x + 5, screenPos.y - 5); } }
function drawXYRegion() { // Use the same uv-corners to calculate the transformed
```

```
(x,y) = (u, v^2) positions. let blUV = createVector(squareCenter.x - squareSize / 2, squareCenter.y - squareSize / 2); let brUV = createVector(squareCenter.x + squareSize / 2, squareCenter.y - squareSize / 2); let trUV = createVector(squareCenter.x + squareSize / 2, squareCenter.y + squareSize / 2); let tlUV = createVector(squareCenter.x - squareSize / 2, squareCenter.y + squareSize / 2); let uvCorners = [blUV, brUV, trUV, tlUV]; // Map the transformed coordinates to the xy-panel let bl = xyFromUVTransformed(blUV.x, blUV.y); let br = xyFromUVTransformed(brUV.x, brUV.y); let tr = xyFromUVTransformed(trUV.x, trUV.y); let tl = xyFromUVTransformed(tlUV.x, tlUV.y); stroke(255, 0, 0); fill(255, 100, 100, 150); beginShape(); vertex(bl.x, bl.y); vertex(br.x, br.y); vertex(tr.x, tr.y); vertex(tl.x, tl.y); endShape(CLOSE); // Annotate each transformed corner with its (x,y) coordinates, where x = u and y = v^2. fill(0); noStroke(); for (let corner of uvCorners) { let transformed = xyFromUVTransformed(corner.x, corner.y); // Compute transformed y as the square of v (v^2) let label = '(' + nf(corner.x, 1, 2) + ', ' + nf(corner.y * corner.y, 1, 2) + ')'; text(label, transformed.x + 5, transformed.y - 5); } } // Coordinate mapping functions for the uv-panel function uvToScreen(u, v) { let x = map(u, -5, 5, uvLeft, uvRight); // In the screen, higher v corresponds to a smaller y value let y = map(v, -5, 5, uvBottom, uvTop); return createVector(x, y); } // Mapping functions for the xy-panel function xyMapX(x) { return map(x, -5, 5, xyLeft, xyRight); } function xyMapY(y) { // y in the transformed plane comes from v^2, so y ranges from 0 to 25. return map(y, 0, 25, xyBottom, xyTop); } function xyFromUVTransformed(u, v) { let x = xyMapX(u); // Here v is the original v, so we transform it to y by squaring it first. let y = xyMapY(v * v); return createVector(x, y); } // Allow dragging of the green center point to move the blue square in the uv-panel. function mouseDragged() { // Only update the center if the mouse is within the uv-panel. if (mouseX >= uvLeft && mouseX <= uvRight && mouseY >= uvTop && mouseY <= uvBottom) { let mouseUV = screenToUV(mouseX, mouseY); // If the mouse is close enough to the current center, update its position. if (dist(mouseUV.x, mouseUV.y, squareCenter.x, squareCenter.y) < 1) { squareCenter.set(mouseUV.x, mouseUV.y); } } } function screenToUV(x, y) { let u = map(x, uvLeft, uvRight, -5, 5); let v = map(y, uvBottom, uvTop, -5, 5); return createVector(u, v); }
```

## How It Works

- **uv–plane (Blue Square):**
  The four corners of the blue square are computed from the square's center and side length. These points are converted to screen coordinates for the left panel. Next, the code draws the square and places small text labels showing each corner's (u, v) coordinates.
- **xy–plane (Red Region):**
  The same uv–corners are transformed via $T(u,v) = (u, v^2)$ to get the red region's corners. After drawing the filled region in the right panel, text labels display the transformed coordinates (with $y = v^2$).
- **Interactivity:**
  Dragging the green center point (if you click near it) moves the blue square, and the labels update accordingly. The slider adjusts the square's size.

This sketch mimics the GeoGebra demonstration you referenced and gives a clear visual and numerical relationship between the original and transformed regions.

on in live web cam like this when moving over the plae with vectors (x+y,x^2-y^2) by divergence the the square in right expandsand contracts telling the fluid is in out < and gratyer than symbol trave with numerical values updating the space

Reasoned for 29 seconds

Below is an interactive **p5.js** sketch that demonstrates a live transformation and divergence display. In this version:

• The left panel shows a blue square in the domain (with coordinates (x,y) in [–5,5]). Its center is marked by a green dot, and each corner is labeled with its (x,y) coordinates.

• The right panel shows the transformed region under

$$F(x,y) = (x + y,\ x^2 - y^2)$$

with each corner labeled by its transformed coordinates.

• The divergence of the vector field at the square's center is computed. (For $F(x,y)$, the divergence is $\frac{\partial}{\partial x}(x+y) + \frac{\partial}{\partial y}(x^2 - y^2) = 1 - 2y$.)
This value is displayed in the right panel along with an arrow-like symbol (">" for expansion when divergence is positive, "<" for contraction when negative).

• A slider lets you adjust the square's side length. You can drag the green dot (when your mouse is close enough) to move the square around, and the transformed region updates in real time. In regions where the divergence is positive, the red region will appear "expanded" relative to the blue square, and vice versa.

Copy and paste the following code into your p5.js editor to try it out:

javascript Copy Edit

```javascript
// Panel boundaries let uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350; // left (domain) panel let xyLeft = 400, xyRight = 700, xyTop = 50, xyBottom = 350; // right (transformed) panel // Square in the domain let squareCenter, squareSize; let sizeSlider; function setup() { createCanvas(750, 400); textSize(14); // Initialize the square center (in domain coordinates) and side length. squareCenter = createVector(0, 0); squareSize = 2; sizeSlider = createSlider(0.5, 5, 2, 0.1); sizeSlider.position(20, 20); } function draw() { background(240); squareSize = sizeSlider.value(); drawPanels(); drawDomainSquare(); drawTransformedRegion(); displayDivergence(); drawLabels(); } function drawPanels() { stroke(150); noFill(); // Left panel (domain) rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop); // Right panel (transformed) rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); } function drawDomainSquare() { // Compute corners of the square (domain coordinates) let half = squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y - half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y + half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); let uvCorners = [bl, br, tr, tl]; // Draw the blue square stroke(0, 0, 255); noFill(); beginShape(); vertex(domainToScreen(bl).x, domainToScreen(bl).y); vertex(domainToScreen(br).x, domainToScreen(br).y); vertex(domainToScreen(tr).x, domainToScreen(tr).y); vertex(domainToScreen(tl).x, domainToScreen(tl).y); endShape(CLOSE); // Draw the draggable center (green dot) fill(0, 255, 0); let centerScreen = domainToScreen(squareCenter); noStroke(); ellipse(centerScreen.x, centerScreen.y, 10, 10); // Label each corner with its (x,y) coordinates fill(0); noStroke(); for (let corner of uvCorners) { let pos = domainToScreen(corner); let label = '(' + nf(corner.x, 1, 2) + ', ' + nf(corner.y, 1, 2) + ')'; text(label, pos.x + 5, pos.y - 5); } } function drawTransformedRegion() { // Compute the four corners in the domain let half = squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y - half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y + half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); // Transform each corner using F(x,y) = (x+y, x^2 - y^2) let f_bl = transformPoint(bl); let f_br = transformPoint(br); let f_tr = transformPoint(tr); let f_tl = transformPoint(tl); // Draw the red transformed region stroke(255, 0, 0); fill(255, 100, 100, 150); beginShape(); vertex(transformToScreen(f_bl).x, transformToScreen(f_bl).y); vertex(transformToScreen(f_br).x, transformToScreen(f_br).y); vertex(transformToScreen(f_tr).x, transformToScreen(f_tr).y); vertex(transformToScreen(f_tl).x, transformToScreen(f_tl).y); endShape(CLOSE); // Label each transformed corner with its (x', y') values fill(0); noStroke(); let transformedCorners = [f_bl, f_br, f_tr, f_tl]; for (let pt of transformedCorners) { let pos = transformToScreen(pt); let label = '(' + nf(pt.x, 1, 2) + ', ' + nf(pt.y, 1, 2) + ')'; text(label, pos.x + 5, pos.y - 5); } } function displayDivergence() { // For F(x,y) = (x+y, x^2 - y^2), the divergence is: // div = ∂/∂x (x+y) + ∂/∂y (x^2-y^2) = 1 - 2y. let div = 1 - 2 * squareCenter.y; fill(0); noStroke(); let msg = 'Divergence: ' + nf(div, 1, 2); if (div > 0) { msg += ' > (expansion)'; } else if (div < 0) { msg += ' < (contraction)'; } else { msg += ' = (neutral)'; } text(msg, xyLeft + 10, xyTop + 20); } function drawLabels() { fill(0); noStroke(); text('Domain (x,y) with F(x,y) = (x+y, x^2-y^2)', uvLeft, uvTop - 20); text('Transformed Region', xyLeft, xyTop - 20); } // Mapping for domain (left panel): x, y ∈ [-5, 5] function domainToScreen(pt) { let sx = map(pt.x, -5, 5, uvLeft, uvRight); // Invert y so that higher y is upward in the panel let sy = map(pt.y, -5, 5, uvBottom, uvTop); return createVector(sx, sy); } // Apply the transformation F(x,y) = (x+y, x^2 - y^2) function transformPoint(pt) { return createVector(pt.x + pt.y, pt.x * pt.x - pt.y * pt.y); } // Mapping for the
```

```
transformed region (right panel): // x' ∈ [-10,10], y' ∈ [-25,25] function
transformToScreen(pt) { let sx = map(pt.x, -10, 10, xyLeft, xyRight); let sy =
map(pt.y, -25, 25, xyBottom, xyTop); return createVector(sx, sy); } // Allow
dragging the green center point in the domain function mouseDragged() { if (mouseX
>= uvLeft && mouseX <= uvRight && mouseY >= uvTop && mouseY <= uvBottom) { let
mousePt = screenToDomain(mouseX, mouseY); // Only update if the mouse is close to
the current center (tolerance ~1 in domain units) if (dist(mousePt.x, mousePt.y,
squareCenter.x, squareCenter.y) < 1) { squareCenter.set(mousePt.x, mousePt.y); } }
} function screenToDomain(x, y) { let u = map(x, uvLeft, uvRight, -5, 5); let v =
map(y, uvBottom, uvTop, -5, 5); return createVector(u, v); }
```

## How It Works

1. **Domain Panel (Left):**
   The square is defined by its center (green dot) and side length (adjustable via the slider). Its four corners are computed and drawn in blue, with each corner labeled by its $(x, y)$ values.
2. **Transformed Panel (Right):**
   Each corner of the square is transformed by
   $$F(x,y) = (x + y,\ x^2 - y^2)$$
   and the resulting red polygon is drawn. Each transformed corner is labeled with its numerical $(x', y')$ values.
3. **Divergence Display:**
   The divergence, $1 - 2y$ (evaluated at the square's center), is displayed at the top of the right panel. A ">" symbol indicates expansion (when divergence is positive) and a "<" symbol indicates contraction (when negative).

Run this sketch in your p5.js environment. As you move (drag) the green point or adjust the slider, observe how the transformed region expands or contracts and how the divergence value updates in real time—mimicking a live fluid flow where the transformation's local "inflow" or "outflow" is directly visible.

in web cam the canvas should run

Reasoned for 10 seconds

Below is an updated complete **p5.js** sketch that runs on a web page with a live webcam feed as the background. In this version:

• The webcam is captured and displayed as the canvas background.
• Two panels are overlaid on top: the left panel shows the original square (with its (x,y) coordinates) and the right panel shows its transformed image under

$$F(x,y) = (x + y,\ x^2 - y^2)$$

with corner coordinates and a divergence value (calculated as $\operatorname{div} F = 1 - 2y$).
• A green draggable dot (near the center) lets you move the square, and a slider adjusts the square's side length.
• The transformed region "expands" or "contracts" based on the local divergence (displaying ">" for expansion and "<" for contraction).

Copy and paste the code into your p5.js editor to see it running live on your webpage:

javascript Copy Edit

```javascript
let video; let uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350; // Domain
panel boundaries (left) let xyLeft = 400, xyRight = 700, xyTop = 50, xyBottom =
350; // Transformed panel boundaries (right) let squareCenter, squareSize; let
sizeSlider; function setup() { createCanvas(750, 400); // Start the webcam capture
video = createCapture(VIDEO); video.size(width, height); video.hide();
textSize(14); // Initialize the square center (domain coordinates) and default side
length. squareCenter = createVector(0, 0); squareSize = 2; sizeSlider =
```

```
createSlider(0.5, 5, 2, 0.1); sizeSlider.position(20, 20); } function draw() { // Draw the live webcam feed as the background (semi-transparent tint so UI remains visible) tint(255, 127); image(video, 0, 0, width, height); noTint(); // Draw white semi-transparent backgrounds for our panels to ensure readability drawPanels(); // Update square size from slider squareSize = sizeSlider.value(); // Draw the blue square (domain) with coordinate labels drawDomainSquare(); // Draw the transformed red region (using F(x,y) = (x+y, x^2-y^2)) with its labels drawTransformedRegion(); // Compute and display the divergence at the square's center displayDivergence(); drawLabels(); } function drawPanels() { // Draw white backgrounds for the panels with some transparency noStroke(); fill(255, 255, 255, 230); rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); // Draw borders stroke(150); noFill(); rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); } function drawDomainSquare() { // Compute corners of the square in the domain (x,y)-coordinates let half = squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y - half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y + half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); let uvCorners = [bl, br, tr, tl]; // Draw the blue square (domain panel) stroke(0, 0, 255); noFill(); beginShape(); vertex(domainToScreen(bl).x, domainToScreen(bl).y); vertex(domainToScreen(br).x, domainToScreen(br).y); vertex(domainToScreen(tr).x, domainToScreen(tr).y); vertex(domainToScreen(tl).x, domainToScreen(tl).y); endShape(CLOSE); // Draw the draggable center (green dot) fill(0, 255, 0); let centerScreen = domainToScreen(squareCenter); noStroke(); ellipse(centerScreen.x, centerScreen.y, 10, 10); // Label each corner with its (x,y) values fill(0); noStroke(); for (let corner of uvCorners) { let pos = domainToScreen(corner); let label = '(' + nf(corner.x, 1, 2) + ', ' + nf(corner.y, 1, 2) + ')'; text(label, pos.x + 5, pos.y - 5); } } function drawTransformedRegion() { // Compute the four corners in domain coordinates let half = squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y - half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y + half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); // Apply the transformation: F(x,y) = (x+y, x^2-y^2) let f_bl = transformPoint(bl); let f_br = transformPoint(br); let f_tr = transformPoint(tr); let f_tl = transformPoint(tl); // Draw the red transformed region (transformed panel) stroke(255, 0, 0); fill(255, 100, 100, 150); beginShape(); vertex(transformToScreen(f_bl).x, transformToScreen(f_bl).y); vertex(transformToScreen(f_br).x, transformToScreen(f_br).y); vertex(transformToScreen(f_tr).x, transformToScreen(f_tr).y); vertex(transformToScreen(f_tl).x, transformToScreen(f_tl).y); endShape(CLOSE); // Label each transformed corner with its (x', y') values fill(0); noStroke(); let transformedCorners = [f_bl, f_br, f_tr, f_tl]; for (let pt of transformedCorners) { let pos = transformToScreen(pt); let label = '(' + nf(pt.x, 1, 2) + ', ' + nf(pt.y, 1, 2) + ')'; text(label, pos.x + 5, pos.y - 5); } } function displayDivergence() { // For F(x,y) = (x+y, x^2-y^2), the divergence is: ∂/∂x(x+y) + ∂/∂y(x^2-y^2) = 1 - 2y. let div = 1 - 2 * squareCenter.y; fill(0); noStroke(); let msg = 'Divergence: ' + nf(div, 1, 2); if (div > 0) { msg += ' > (expansion)'; } else if (div < 0) { msg += ' < (contraction)'; } else { msg += ' = (neutral)'; } text(msg, xyLeft + 10, xyTop + 20); } function drawLabels() { fill(0); noStroke(); text('Domain (x,y) with F(x,y) = (x+y, x^2-y^2)', uvLeft, uvTop - 20); text('Transformed Region', xyLeft, xyTop - 20); } // Mapping for the domain (left panel); x, y ∈ [-5, 5] function domainToScreen(pt) { let sx = map(pt.x, -5, 5, uvLeft, uvRight); let sy = map(pt.y, -5, 5, uvBottom, uvTop); // Note: higher y in domain maps upward (smaller y on screen) return createVector(sx, sy); } // Apply the transformation F(x,y) = (x+y, x^2-y^2) function transformPoint(pt) { return createVector(pt.x + pt.y, pt.x * pt.x - pt.y * pt.y); } // Mapping for the transformed region (right panel): // For this example we assume x' ∈ [-10,10] and y' ∈ [-25,25] function transformToScreen(pt) { let sx = map(pt.x, -10, 10, xyLeft, xyRight); let sy = map(pt.y, -25, 25, xyBottom, xyTop); return createVector(sx, sy); } // Allow dragging of the green center point in the domain panel. function mouseDragged() { // Only update if the mouse is inside the domain panel if (mouseX >= uvLeft && mouseX <= uvRight && mouseY >= uvTop && mouseY <= uvBottom) { let mousePt = screenToDomain(mouseX, mouseY); // Only update if the cursor is close to the current center (tolerance ~1 in domain units) if (dist(mousePt.x, mousePt.y, squareCenter.x, squareCenter.y) < 1) { squareCenter.set(mousePt.x, mousePt.y); } } } function screenToDomain(x, y) { let u = map(x, uvLeft, uvRight, -5, 5); let v = map(y, uvBottom, uvTop, -5, 5); return createVector(u, v); }
```

## How It Works

1. **Webcam Background:**

   The sketch uses

```javascript
video = createCapture(VIDEO);
```

to obtain a live webcam feed. In each draw cycle, the video is drawn (with semi-transparency) as the background.

2. **Panels and Overlays:**
   The left panel displays the blue square in the original domain. The right panel shows its red transformed image under $F(x, y) = (x + y,\ x^2 - y^2)$. White, semi-transparent backgrounds for both panels help ensure the overlaid text and shapes remain legible.
3. **Interactivity:**
   The green dot (at the square's center) can be dragged to reposition the square, and the slider adjusts its size. As you move the square, the labels update and the divergence (computed as $1 - 2y$) is shown—indicating local expansion (">") or contraction ("<").

This sketch should run live in a web browser with your webcam active. Enjoy exploring how the fluid "flows" as represented by the transformation and divergence!

plot the vectoralso and transparent web cam of background showing

Reasoned for 31 seconds

Below is a complete **p5.js** sketch that runs on a webpage with a live, transparent webcam feed in the background, plots the domain and transformed regions, and also draws the vector field for

$$F(x, y) = (x + y,\ x^2 - y^2)$$

over the left (domain) panel. In this sketch:

- The webcam feed is drawn with partial transparency.
- The left panel (domain) shows a blue square with its four (x,y) corners labeled and a draggable green center.
- A grid of green arrows represents the vector field $F(x, y)$ on the domain.
- The right panel shows the red transformed region using the mapping $F(x, y) = (x + y,\ x^2 - y^2)$ with its corners labeled.
- The divergence at the square's center (computed as $1 - 2y$) is displayed with an arrow symbol indicating expansion (">") or contraction ("<").

Copy and paste this code into your p5.js editor to run it live:

```javascript
let video; let uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350; // Domain
(left) panel boundaries let xyLeft = 400, xyRight = 700, xyTop = 50, xyBottom =
350; // Transformed (right) panel boundaries let squareCenter, squareSize; let
sizeSlider; function setup() { createCanvas(750, 400); // Start video capture and
hide the default element video = createCapture(VIDEO); video.size(width, height);
video.hide(); textSize(14); // Initialize square center (in domain coordinates) and
side length. squareCenter = createVector(0, 0); squareSize = 2; // Slider to adjust
the square's side length. sizeSlider = createSlider(0.5, 5, 2, 0.1);
sizeSlider.position(20, 20); } function draw() { background(240); // Draw the live
webcam feed as a transparent background. tint(255, 150); image(video, 0, 0, width,
height); noTint(); // Draw semi-transparent backgrounds for both panels.
drawPanels(); // Draw the vector field on the domain panel. drawVectorField(); //
Update the square size from the slider. squareSize = sizeSlider.value(); // Draw
the blue square (domain) with corner coordinate labels. drawDomainSquare(); // Draw
the transformed red region with labels. drawTransformedRegion(); // Compute and
display the divergence at the square's center. displayDivergence(); // Draw overall
panel labels. drawLabels(); } function drawPanels() { // White semi-transparent
backgrounds for the panels. noStroke(); fill(255, 255, 255, 200); rect(uvLeft,
```

```
uvTop, uvRight - uvLeft, uvBottom - uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); // Draw panel borders. stroke(150); noFill(); rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); } function drawVectorField() { // Plot the vector field F(x,y) = (x+y, x^2-y^2) over a grid in the domain. let spacing = 1.0; // Grid spacing in domain units. stroke(0, 150, 0); fill(0, 150, 0); // The mapping from domain [-5,5] to screen: (uvRight-uvLeft) covers 10 units. let pixelScale = (uvRight - uvLeft) / 10; // pixels per unit let arrowScale = 0.1; // Scale factor to keep arrows from being too long // Iterate over a grid in the domain. for (let x = -5; x <= 5; x += spacing) { for (let y = -5; y <= 5; y += spacing) { let pt = createVector(x, y); let vec = createVector(x + y, x*x - y*y); vec.mult(arrowScale); // Scale down the vector field for drawing // Convert the domain point to screen coordinates. let base = domainToScreen(pt); // Multiply by pixelScale to convert vector length to pixels. let arrow = vec.copy().mult(pixelScale); drawArrow(base, arrow); } } } function drawArrow(base, vec) { push(); translate(base.x, base.y); line(0, 0, vec.x, vec.y); push(); translate(vec.x, vec.y); rotate(vec.heading()); let arrowSize = 4; noStroke(); fill(0, 150, 0); triangle(0, arrowSize/2, 0, -arrowSize/2, arrowSize, 0); pop(); pop(); } function drawDomainSquare() { // Compute the four corners of the square in domain coordinates. let half = squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y - half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y + half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); let corners = [bl, br, tr, tl]; // Draw the blue square. stroke(0, 0, 255); noFill(); beginShape(); for (let i = 0; i < corners.length; i++) { let pos = domainToScreen(corners[i]); vertex(pos.x, pos.y); } endShape(CLOSE); // Draw the draggable center as a green dot. fill(0, 255, 0); noStroke(); let centerScreen = domainToScreen(squareCenter); ellipse(centerScreen.x, centerScreen.y, 10, 10); // Label each corner with its (x,y) coordinates. fill(0); noStroke(); for (let corner of corners) { let pos = domainToScreen(corner); let label = '(' + nf(corner.x, 1, 2) + ', ' + nf(corner.y, 1, 2) + ')'; text(label, pos.x + 5, pos.y - 5); } } function drawTransformedRegion() { // Compute the square's corners in domain coordinates. let half = squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y - half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y + half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); // Apply the transformation: F(x,y) = (x+y, x^2-y^2) let f_bl = transformPoint(bl); let f_br = transformPoint(br); let f_tr = transformPoint(tr); let f_tl = transformPoint(tl); // Draw the red transformed region in the right panel. stroke(255, 0, 0); fill(255, 100, 100, 150); beginShape(); vertex(transformToScreen(f_bl).x, transformToScreen(f_bl).y); vertex(transformToScreen(f_br).x, transformToScreen(f_br).y); vertex(transformToScreen(f_tr).x, transformToScreen(f_tr).y); vertex(transformToScreen(f_tl).x, transformToScreen(f_tl).y); endShape(CLOSE); // Label each transformed corner with its (x', y') coordinates. fill(0); noStroke(); let transformedCorners = [f_bl, f_br, f_tr, f_tl]; for (let pt of transformedCorners) { let pos = transformToScreen(pt); let label = '(' + nf(pt.x, 1, 2) + ', ' + nf(pt.y, 1, 2) + ')'; text(label, pos.x + 5, pos.y - 5); } } function displayDivergence() { // For F(x,y) = (x+y, x^2-y^2), the divergence at the square's center is: // div = ∂/∂x (x+y) + ∂/∂y (x^2-y^2) = 1 - 2y. let div = 1 - 2 * squareCenter.y; fill(0); noStroke(); let msg = 'Divergence: ' + nf(div, 1, 2); if (div > 0) { msg += ' > (expansion)'; } else if (div < 0) { msg += ' < (contraction)'; } else { msg += ' = (neutral)'; } text(msg, xyLeft + 10, xyTop + 20); } function drawLabels() { fill(0); noStroke(); text('Domain (x,y) with F(x,y) = (x+y, x^2-y^2) & vector field', uvLeft, uvTop - 20); text('Transformed Region', xyLeft, xyTop - 20); } // Mapping from domain coordinates (x,y) in [-5,5] to screen coordinates in the left panel. function domainToScreen(pt) { let sx = map(pt.x, -5, 5, uvLeft, uvRight); let sy = map(pt.y, -5, 5, uvBottom, uvTop); return createVector(sx, sy); } // Transformation F(x,y) = (x+y, x^2-y^2) function transformPoint(pt) { return createVector(pt.x + pt.y, pt.x*pt.x - pt.y*pt.y); } // Mapping from transformed coordinates to screen coordinates in the right panel. // (Assuming x' ∈ [−10,10] and y' ∈ [−25,25] for this visualization.) function transformToScreen(pt) { let sx = map(pt.x, -10, 10, xyLeft, xyRight); let sy = map(pt.y, -25, 25, xyBottom, xyTop); return createVector(sx, sy); } // Allow dragging of the green center dot within the domain panel. function mouseDragged() { if (mouseX >= uvLeft && mouseX <= uvRight && mouseY >= uvTop && mouseY <= uvBottom) { let mousePt = screenToDomain(mouseX, mouseY); if (dist(mousePt.x, mousePt.y, squareCenter.x, squareCenter.y) < 1) { squareCenter.set(mousePt.x, mousePt.y); } } } function screenToDomain(x, y) { let u = map(x, uvLeft, uvRight, -5, 5); let v = map(y, uvBottom, uvTop, -5, 5); return createVector(u, v); }
```

## How It Works

1. **Transparent Webcam Background:**
   The video feed (obtained via `createCapture(VIDEO)`) is drawn with `tint(255, 150)`, providing a semi-transparent background so that all interactive elements remain visible.
2. **Domain Panel (Left):**
   The blue square (with labeled corners) is drawn over a white, semi-transparent panel. A grid of green arrows is plotted to visualize the vector field $F(x, y) = (x + y,\ x^2 - y^2)$.
3. **Transformed Panel (Right):**
   The red region is drawn as the image of the blue square under the transformation. Its corners are labeled with their transformed coordinates.
4. **Interactivity:**
   A draggable green dot lets you reposition the square, and a slider adjusts its size. The divergence (computed as $1 - 2y$ at the square's center) updates in real time, indicating local expansion or contraction.

Enjoy exploring this live, interactive sketch with both a transparent webcam background and a plotted vector field!

now for curl circulation

Reasoned for 26 seconds

Below is a complete **p5.js** sketch that not only uses a transparent live webcam feed as the background, plots the vector field, and shows the interactive square (with its transformed region and divergence) but now also computes and displays the curl circulation. In this example the vector field is

$$F(x, y) = (x + y,\ x^2 - y^2)$$

so that its curl (the z–component in 2D) is given by

$$\operatorname{curl} F = \frac{\partial}{\partial x}(x^2 - y^2) - \frac{\partial}{\partial y}(x + y) = 2x - 1.$$

By Green's theorem, the circulation (i.e. the line integral of $F$ along the closed square) is equal to the double–integral of the curl over the square. For a small square, you can approximate this by evaluating the curl at the square's center and multiplying by the square's area. This sketch computes the circulation numerically along the square's boundary (using a discrete sum) and also shows the "predicted" value (using $(2x_c - 1)\times$ area, with $x_c$ the square's center x–coordinate).

Copy and paste the code into your p5.js editor to run it live in a browser:

```javascript
let video; let uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350; // Domain (left) panel boundaries let xyLeft = 400, xyRight = 700, xyTop = 50, xyBottom = 350; // Transformed (right) panel boundaries let squareCenter, squareSize; let sizeSlider; function setup() { createCanvas(750, 500); // Start video capture and hide its default element. video = createCapture(VIDEO); video.size(width, height); video.hide(); textSize(14); // Initialize square center (domain coordinates) and default side length. squareCenter = createVector(0, 0); squareSize = 2; // Create slider to adjust the square's side length. sizeSlider = createSlider(0.5, 5, 2, 0.1); sizeSlider.position(20, 20); } function draw() { background(240); // Draw the webcam feed as a transparent background. tint(255, 150); image(video, 0, 0, width, height); noTint(); // Draw semi-transparent white backgrounds for the panels. drawPanels(); // Draw the vector field over the domain (left panel). drawVectorField(); // Update square size from slider. squareSize = sizeSlider.value(); // Draw the blue square in the domain with (x,y) corner labels. drawDomainSquare(); // Draw the transformed red region in the right panel. drawTransformedRegion(); // Display the divergence (evaluated at the square's center) in the right panel. displayDivergence(); // Compute and display the circulation along the square's boundary and the predicted circulation from the curl. displayCirculation(); // Draw overall labels. drawLabels(); } function drawPanels() { noStroke(); fill(255, 255, 255, 200); rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop);
```

```
stroke(150); noFill(); rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop);
rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); } function
drawVectorField() { // Plot the vector field F(x,y) = (x+y, x^2-y^2) on a grid over
the domain. let spacing = 1.0; // domain units stroke(0, 150, 0); fill(0, 150, 0);
// Calculate pixels per unit for the left panel. let pixelScale = (uvRight -
uvLeft) / 10; let arrowScale = 0.1; // scaling factor to reduce arrow lengths for
(let x = -5; x <= 5; x += spacing) { for (let y = -5; y <= 5; y += spacing) { let
pt = createVector(x, y); let vec = createVector(x + y, x * x - y * y);
vec.mult(arrowScale); let base = domainToScreen(pt); let arrow =
vec.copy().mult(pixelScale); drawArrow(base, arrow); } } } function drawArrow(base,
vec) { push(); translate(base.x, base.y); line(0, 0, vec.x, vec.y); push();
translate(vec.x, vec.y); rotate(vec.heading()); let arrowSize = 4; noStroke();
fill(0, 150, 0); triangle(0, arrowSize/2, 0, -arrowSize/2, arrowSize, 0); pop();
pop(); } function drawDomainSquare() { // Compute the four corners of the square in
domain coordinates. let half = squareSize / 2; let bl = createVector(squareCenter.x
- half, squareCenter.y - half); let br = createVector(squareCenter.x + half,
squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y
+ half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); let
corners = [bl, br, tr, tl]; // Draw the blue square. stroke(0, 0, 255); noFill();
beginShape(); for (let pt of corners) { let screenPt = domainToScreen(pt);
vertex(screenPt.x, screenPt.y); } endShape(CLOSE); // Draw the draggable center as
a green dot. fill(0, 255, 0); let centerScreen = domainToScreen(squareCenter);
noStroke(); ellipse(centerScreen.x, centerScreen.y, 10, 10); // Annotate each
corner with its (x,y) coordinates. fill(0); noStroke(); for (let pt of corners) {
let screenPt = domainToScreen(pt); let label = '(' + nf(pt.x, 1, 2) + ', ' +
nf(pt.y, 1, 2) + ')'; text(label, screenPt.x + 5, screenPt.y - 5); } } function
drawTransformedRegion() { // Compute the square's corners in domain coordinates.
let half = squareSize / 2; let bl = createVector(squareCenter.x - half,
squareCenter.y - half); let br = createVector(squareCenter.x + half, squareCenter.y
- half); let tr = createVector(squareCenter.x + half, squareCenter.y + half); let
tl = createVector(squareCenter.x - half, squareCenter.y + half); // Transform each
corner: F(x,y) = (x+y, x^2-y^2). let f_bl = transformPoint(bl); let f_br =
transformPoint(br); let f_tr = transformPoint(tr); let f_tl = transformPoint(tl);
// Draw the red transformed region. stroke(255, 0, 0); fill(255, 100, 100, 150);
beginShape(); vertex(transformToScreen(f_bl).x, transformToScreen(f_bl).y);
vertex(transformToScreen(f_br).x, transformToScreen(f_br).y);
vertex(transformToScreen(f_tr).x, transformToScreen(f_tr).y);
vertex(transformToScreen(f_tl).x, transformToScreen(f_tl).y); endShape(CLOSE); //
Label each transformed corner. fill(0); noStroke(); let transformedCorners = [f_bl,
f_br, f_tr, f_tl]; for (let pt of transformedCorners) { let screenPt =
transformToScreen(pt); let label = '(' + nf(pt.x, 1, 2) + ', ' + nf(pt.y, 1, 2) +
')'; text(label, screenPt.x + 5, screenPt.y - 5); } } function displayDivergence()
{ // For F(x,y) = (x+y, x^2-y^2), divergence is: 1 - 2y. let div = 1 - 2 *
squareCenter.y; fill(0); noStroke(); let msg = 'Divergence: ' + nf(div, 1, 2); if
(div > 0) { msg += ' > (expansion)'; } else if (div < 0) { msg += ' <
(contraction)'; } else { msg += ' = (neutral)'; } text(msg, xyLeft + 10, xyTop +
40); } function computeCirculation() { // Compute circulation (line integral)
around the square's boundary using a discrete sum. let half = squareSize / 2; let
bl = createVector(squareCenter.x - half, squareCenter.y - half); let br =
createVector(squareCenter.x + half, squareCenter.y - half); let tr =
createVector(squareCenter.x + half, squareCenter.y + half); let tl =
createVector(squareCenter.x - half, squareCenter.y + half); let edges = [[bl, br],
[br, tr], [tr, tl], [tl, bl]]; let circulation = 0; let steps = 20; // number of
sub-intervals per edge for (let edge of edges) { let start = edge[0]; let end =
edge[1]; let d = p5.Vector.sub(end, start); let dt = 1 / steps; for (let i = 0; i <
steps; i++) { let t = i * dt; let pos = p5.Vector.lerp(start, end, t); let ds =
d.copy().mult(dt); // Evaluate F(x,y) = (x+y, x^2-y^2) at pos. let F =
createVector(pos.x + pos.y, pos.x * pos.x - pos.y * pos.y); circulation +=
F.dot(ds); } } return circulation; } function displayCirculation() { let circ =
computeCirculation(); // Predicted circulation via Green's theorem (approximation):
(curl at center)*(area) // Here curl F = 2x-1 evaluated at squareCenter, and area =
squareSize^2. let predicted = (2 * squareCenter.x - 1) * (squareSize * squareSize);
fill(0); noStroke(); let msg = 'Circulation (line integral): ' + nf(circ, 1, 2);
msg += ' | Predicted (curl*Area): ' + nf(predicted, 1, 2); text(msg, xyLeft + 10,
xyTop + 60); } function drawLabels() { fill(0); noStroke(); text('Domain (x,y) with
F(x,y) = (x+y, x^2-y^2) & vector field', uvLeft, uvTop - 20); text('Transformed
Region, Divergence, & Curl Circulation', xyLeft, xyTop - 20); } // Mapping: domain
(x,y) in [-5,5] to screen coordinates in left panel. function domainToScreen(pt) {
let sx = map(pt.x, -5, 5, uvLeft, uvRight); let sy = map(pt.y, -5, 5, uvBottom,
uvTop); return createVector(sx, sy); } // Transformation: F(x,y) = (x+y, x^2-y^2)
function transformPoint(pt) { return createVector(pt.x + pt.y, pt.x * pt.x - pt.y *
pt.y); } // Mapping for transformed region: assume x' ∈ [-10,10] and y' ∈ [-25,25].
function transformToScreen(pt) { let sx = map(pt.x, -10, 10, xyLeft, xyRight); let
sy = map(pt.y, -25, 25, xyBottom, xyTop); return createVector(sx, sy); } function
```

```
mouseDragged() { // Allow dragging the green center dot in the domain panel. if (mouseX >= uvLeft && mouseX <= uvRight && mouseY >= uvTop && mouseY <= uvBottom) { let mousePt = screenToDomain(mouseX, mouseY); if (dist(mousePt.x, mousePt.y, squareCenter.x, squareCenter.y) < 1) { squareCenter.set(mousePt.x, mousePt.y); } } } function screenToDomain(x, y) { let u = map(x, uvLeft, uvRight, -5, 5); let v = map(y, uvBottom, uvTop, -5, 5); return createVector(u, v); }
```

## How It Works

1. **Transparent Webcam Background:**
   The video feed is captured via `createCapture(VIDEO)` and drawn with partial transparency ( `tint(255, 150)` ) so the interactive elements remain clearly visible.
2. **Domain Panel (Left):**
   A blue square is drawn in the domain (with labeled corners and a draggable green center). A grid of green arrows shows the vector field $F(x, y) = (x + y, x^2 - y^2)$.
3. **Transformed Panel (Right):**
   The square is mapped under the transformation to produce a red region with its own coordinate labels. The divergence (calculated as $1 - 2y$) is displayed.
4. **Curl Circulation:**
   The function `computeCirculation()` numerically integrates $F$ along the boundary of the square to compute its circulation. This value is compared to the prediction from Green's theorem—namely, the curl at the center $(2x - 1)$ times the square's area—both of which are displayed.

Experiment by moving the green dot and adjusting the slider. You'll observe how the circulation (and thus the local "rotational" behavior of the field) changes across the domain.

This interactive sketch provides an engaging visualization of curl and circulation in a 2D vector field with a live webcam background. Enjoy exploring!

show rotation andcirculation in each point also by animation

Reasoned for 10 seconds

Below is a complete **p5.js** sketch that does the following:

• Shows a live, semi-transparent webcam feed as the canvas background.
• In the left "domain" panel, it draws a grid of green arrows representing the vector field

$$F(x, y) = (x + y,\ x^2 - y^2)$$

over the domain.
• At each grid point the code animates a small purple dot revolving around a tiny circle whose angular speed is proportional to the local curl (which for this field is $\operatorname{curl} F = 2x - 1$). This animated "rotation" visualizes the local circulation at that point.
• An interactive blue square (with its corners labeled) is drawn on the left panel along with a draggable green dot (its center) and a slider to adjust its side length. Its transformed image (via $F$) appears as a red region on the right panel with corner labels.
• The sketch also computes and displays the divergence (evaluated at the square's center) and the circulation along the square's boundary (both via numerical integration and by the curl–area prediction).

Copy and paste the code into your p5.js editor to run it live in a browser:

javascript

```javascript
let video; let uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350; // Domain (left) panel boundaries let xyLeft = 400, xyRight = 700, xyTop = 50, xyBottom = 350; // Transformed (right) panel boundaries let squareCenter, squareSize; let sizeSlider; function setup() { createCanvas(750, 500); // Start video capture and hide its default element. video = createCapture(VIDEO); video.size(width, height); video.hide(); textSize(14); // Initialize square center (domain coordinates) and
```

```
default side length. squareCenter = createVector(0, 0); squareSize = 2; // Create
slider to adjust the square's side length. sizeSlider = createSlider(0.5, 5, 2,
0.1); sizeSlider.position(20, 20); } function draw() { background(240); // Draw the
webcam feed as a transparent background. tint(255, 150); image(video, 0, 0, width,
height); noTint(); // Draw white, semi-transparent panel backgrounds. drawPanels();
// Draw the vector field over the domain. drawVectorField(); // Animate a rotating
dot at each grid point to indicate local circulation. drawRotationAnimation(); //
Update the square's side length from the slider. squareSize = sizeSlider.value();
// Draw the blue square (with its labeled corners) and draggable center.
drawDomainSquare(); // Draw the transformed red region in the right panel.
drawTransformedRegion(); // Display divergence (at the square's center) in the
right panel. displayDivergence(); // Compute and display the circulation along the
square's boundary. displayCirculation(); // Overall labels. drawLabels(); }
function drawPanels() { noStroke(); fill(255, 255, 255, 200); rect(uvLeft, uvTop,
uvRight - uvLeft, uvBottom - uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom
- xyTop); stroke(150); noFill(); rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom -
uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); } function
drawVectorField() { // Plot F(x,y) = (x+y, x^2-y^2) over a grid in the domain. let
spacing = 1.0; // grid spacing in domain units stroke(0, 150, 0); fill(0, 150, 0);
// Convert domain units to pixels (left panel spans 10 units). let pixelScale =
(uvRight - uvLeft) / 10; let arrowScale = 0.1; // reduce arrow lengths for (let x =
-5; x <= 5; x += spacing) { for (let y = -5; y <= 5; y += spacing) { let pos =
createVector(x, y); let vec = createVector(x + y, x*x - y*y); vec.mult(arrowScale);
let base = domainToScreen(pos); let arrow = vec.copy().mult(pixelScale);
drawArrow(base, arrow); } } } function drawArrow(base, vec) { push();
translate(base.x, base.y); line(0, 0, vec.x, vec.y); push(); translate(vec.x,
vec.y); rotate(vec.heading()); let arrowSize = 4; noStroke(); fill(0, 150, 0);
triangle(0, arrowSize/2, 0, -arrowSize/2, arrowSize, 0); pop(); pop(); } function
drawRotationAnimation() { // At each grid point, animate a small purple dot
revolving around a circle. let spacing = 1.0; // domain grid spacing let pixelScale
= (uvRight - uvLeft) / 10; let R = 5; // radius (in pixels) for the animated circle
// Loop over domain grid points. for (let x = -5; x <= 5; x += spacing) { for (let
y = -5; y <= 5; y += spacing) { let pos = createVector(x, y); // Compute local
curl: for F(x,y) = (x+y, x^2-y^2), curl = 2x - 1. let curl = 2 * pos.x - 1; // Set
angular speed proportional to local curl (adjust scaling as needed). let
angularSpeed = 0.05 * curl; let angle = frameCount * angularSpeed; // Compute
animated dot's offset from the grid point. let dx = R * cos(angle); let dy = R *
sin(angle); let screenPos = domainToScreen(pos); // Draw a faint circle around the
grid point. noFill(); stroke(200, 0, 200, 150); ellipse(screenPos.x, screenPos.y,
2*R, 2*R); // Draw the rotating dot. noStroke(); fill(200, 0, 200);
ellipse(screenPos.x + dx, screenPos.y + dy, 4, 4); } } } function
drawDomainSquare() { // Compute square corners in domain coordinates. let half =
squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y -
half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr
= createVector(squareCenter.x + half, squareCenter.y + half); let tl =
createVector(squareCenter.x - half, squareCenter.y + half); let corners = [bl, br,
tr, tl]; // Draw blue square. stroke(0, 0, 255); noFill(); beginShape(); for (let
pt of corners) { let screenPt = domainToScreen(pt); vertex(screenPt.x, screenPt.y);
} endShape(CLOSE); // Draw draggable center (green dot). fill(0, 255, 0); let
centerScreen = domainToScreen(squareCenter); noStroke(); ellipse(centerScreen.x,
centerScreen.y, 10, 10); // Label each corner with its (x,y) coordinates. fill(0);
noStroke(); for (let pt of corners) { let screenPt = domainToScreen(pt); let label
= '(' + nf(pt.x, 1, 2) + ', ' + nf(pt.y, 1, 2) + ')'; text(label, screenPt.x + 5,
screenPt.y - 5); } } function drawTransformedRegion() { // Compute square corners
in domain coordinates. let half = squareSize / 2; let bl =
createVector(squareCenter.x - half, squareCenter.y - half); let br =
createVector(squareCenter.x + half, squareCenter.y - half); let tr =
createVector(squareCenter.x + half, squareCenter.y + half); let tl =
createVector(squareCenter.x - half, squareCenter.y + half); // Transform corners:
F(x,y) = (x+y, x^2-y^2). let f_bl = transformPoint(bl); let f_br =
transformPoint(br); let f_tr = transformPoint(tr); let f_tl = transformPoint(tl);
// Draw red transformed region. stroke(255, 0, 0); fill(255, 100, 100, 150);
beginShape(); vertex(transformToScreen(f_bl).x, transformToScreen(f_bl).y);
vertex(transformToScreen(f_br).x, transformToScreen(f_br).y);
vertex(transformToScreen(f_tr).x, transformToScreen(f_tr).y);
vertex(transformToScreen(f_tl).x, transformToScreen(f_tl).y); endShape(CLOSE); //
Label each transformed corner. fill(0); noStroke(); let transformedCorners = [f_bl,
f_br, f_tr, f_tl]; for (let pt of transformedCorners) { let screenPt =
transformToScreen(pt); let label = '(' + nf(pt.x, 1, 2) + ', ' + nf(pt.y, 1, 2) +
')'; text(label, screenPt.x + 5, screenPt.y - 5); } } function displayDivergence()
{ // For F(x,y) = (x+y, x^2-y^2), divergence = 1 - 2y (evaluated at the square's
center). let div = 1 - 2 * squareCenter.y; fill(0); noStroke(); let msg =
'Divergence: ' + nf(div, 1, 2); if (div > 0) { msg += ' > (expansion)'; } else if
(div < 0) { msg += ' < (contraction)'; } else { msg += ' = (neutral)'; } text(msg,
```

```
xyLeft + 10, xyTop + 40); } function computeCirculation() { // Compute circulation
(line integral) along the square's boundary using a discrete sum. let half =
squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y -
half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr
= createVector(squareCenter.x + half, squareCenter.y + half); let tl =
createVector(squareCenter.x - half, squareCenter.y + half); let edges = [[bl, br],
[br, tr], [tr, tl], [tl, bl]]; let circulation = 0; let steps = 20; // sub-
intervals per edge for (let edge of edges) { let start = edge[0]; let end =
edge[1]; let d = p5.Vector.sub(end, start); let dt = 1 / steps; for (let i = 0; i <
steps; i++) { let t = i * dt; let pos = p5.Vector.lerp(start, end, t); let ds =
d.copy().mult(dt); // Evaluate F(x,y) = (x+y, x^2-y^2) at pos. let F =
createVector(pos.x + pos.y, pos.x * pos.x - pos.y * pos.y); circulation +=
F.dot(ds); } } return circulation; } function displayCirculation() { let circ =
computeCirculation(); // Predicted circulation via Green's theorem (approximation):
(curl at center)*(area) // Here curl F = 2x - 1 evaluated at squareCenter, and area
= squareSize^2. let predicted = (2 * squareCenter.x - 1) * (squareSize *
squareSize); fill(0); noStroke(); let msg = 'Circulation (line integral): ' +
nf(circ, 1, 2); msg += ' | Predicted (curl*Area): ' + nf(predicted, 1, 2);
text(msg, xyLeft + 10, xyTop + 60); } function drawLabels() { fill(0); noStroke();
text('Domain (x,y) with F(x,y) = (x+y, x^2-y^2), vector field & rotation', uvLeft,
uvTop - 20); text('Transformed Region, Divergence, & Curl Circulation', xyLeft,
xyTop - 20); } // Mapping from domain (x,y) in [-5,5] to screen coordinates in left
panel. function domainToScreen(pt) { let sx = map(pt.x, -5, 5, uvLeft, uvRight);
let sy = map(pt.y, -5, 5, uvBottom, uvTop); return createVector(sx, sy); } //
Transformation: F(x,y) = (x+y, x^2-y^2). function transformPoint(pt) { return
createVector(pt.x + pt.y, pt.x*pt.x - pt.y*pt.y); } // Mapping for transformed
region (assume x' ∈ [-10,10] and y' ∈ [-25,25]). function transformToScreen(pt) {
let sx = map(pt.x, -10, 10, xyLeft, xyRight); let sy = map(pt.y, -25, 25, xyBottom,
xyTop); return createVector(sx, sy); } function mouseDragged() { // Allow dragging
the green center dot within the domain panel. if (mouseX >= uvLeft && mouseX <=
uvRight && mouseY >= uvTop && mouseY <= uvBottom) { let mousePt =
screenToDomain(mouseX, mouseY); if (dist(mousePt.x, mousePt.y, squareCenter.x,
squareCenter.y) < 1) { squareCenter.set(mousePt.x, mousePt.y); } } } function
screenToDomain(x, y) { let u = map(x, uvLeft, uvRight, -5, 5); let v = map(y,
uvBottom, uvTop, -5, 5); return createVector(u, v); }
```

## How It Works

1. **Webcam Background & Panels:**
   The video feed is captured via `createCapture(VIDEO)` and drawn with partial transparency. White, semi-transparent panels define the left (domain) and right (transformed) regions.
2. **Vector Field & Rotation Animation:**
   In the left panel, a grid of green arrows shows the vector field. Additionally, for each grid point a small purple dot revolves around a faint circle at a rate proportional to the local curl $(2x-1)$ so you can "see" the local rotation and circulation.
3. **Interactive Square & Transformation:**
   The blue square (with its corners labeled) can be repositioned by dragging its center (green dot) and its size adjusted by the slider. Its transformed image under $F$ appears in the right panel.
4. **Divergence & Circulation:**
   The divergence at the square's center and the circulation along its boundary (numerically integrated and compared with the prediction via curl×area) are computed and displayed.

This animated sketch visualizes both the local rotational behavior (via the rotating dots at each grid point) and the overall circulation and divergence of the field. Enjoy exploring the interplay of vector field rotation and circulation!

now show how much vectors engaged in boundry by blinking vectors and flux erupts by animation along the square

Reasoned for 15 seconds

Below is a complete **p5.js** sketch that builds on the previous versions. In this sketch you'll see:

• A live, semi-transparent webcam feed as the background.

• In the left (domain) panel:

- A grid of green arrows shows the vector field
$$F(x, y) = (x + y,\ x^2 - y^2)$$
- At each grid point a small purple dot rotates (its angular speed is proportional to the local curl $2x - 1$) to visualize local circulation.
- An interactive blue square is drawn with its four (x,y) corners labeled and a draggable green center; a slider adjusts its side length.
- Along the boundary of the square, "flux animations" appear: at several boundary sample points blinking arrows (in orange) "erupt" along the outward normal. Their length (and blinking/pulsation) is modulated by the local flux (i.e. $F \cdot n$) so you can see which parts of the boundary are "engaged" by the flow. • In the right (transformed) panel the red image of the square under $F$ is drawn with its corners labeled, and the divergence (and circulation) computed on the square is displayed.

Copy and paste the code into your p5.js editor to try it out live:

javascript

```javascript
let video; let uvLeft = 50, uvRight = 350, uvTop = 50, uvBottom = 350; // Domain (left) panel boundaries let xyLeft = 400, xyRight = 700, xyTop = 50, xyBottom = 350; // Transformed (right) panel boundaries let squareCenter, squareSize; let sizeSlider; function setup() { createCanvas(750, 520); // Start video capture and hide its default element. video = createCapture(VIDEO); video.size(width, height); video.hide(); textSize(14); // Initialize square center (domain coordinates) and default side length. squareCenter = createVector(0, 0); squareSize = 2; // Create slider to adjust the square's side length. sizeSlider = createSlider(0.5, 5, 2, 0.1); sizeSlider.position(20, 20); } function draw() { background(240); // Draw the live webcam feed as a transparent background. tint(255, 150); image(video, 0, 0, width, height); noTint(); // Draw white, semi-transparent panel backgrounds. drawPanels(); // Draw the vector field on the domain. drawVectorField(); // Animate a rotating dot at each grid point to indicate local circulation. drawRotationAnimation(); // Update square size from slider. squareSize = sizeSlider.value(); // Draw the interactive blue square with labeled corners and a draggable center. drawDomainSquare(); // Animate blinking flux arrows along the square boundary. drawFluxAnimation(); // Draw the transformed region in the right panel. drawTransformedRegion(); // Display divergence (evaluated at the square's center) and circulation. displayDivergence(); displayCirculation(); // Overall labels. drawLabels(); } function drawPanels() { noStroke(); fill(255, 255, 255, 200); rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); stroke(150); noFill(); rect(uvLeft, uvTop, uvRight - uvLeft, uvBottom - uvTop); rect(xyLeft, xyTop, xyRight - xyLeft, xyBottom - xyTop); } function drawVectorField() { // Plot F(x,y) = (x+y, x^2-y^2) over a grid in the domain. let spacing = 1.0; // grid spacing in domain units stroke(0, 150, 0); fill(0, 150, 0); // The left panel covers 10 domain units horizontally. let pixelScale = (uvRight - uvLeft) / 10; let arrowScale = 0.1; // scale factor to reduce arrow lengths for (let x = -5; x <= 5; x += spacing) { for (let y = -5; y <= 5; y += spacing) { let pos = createVector(x, y); let vec = createVector(x + y, x*x - y*y); vec.mult(arrowScale); let base = domainToScreen(pos); let arrow = vec.copy().mult(pixelScale); drawArrow(base, arrow); } } } function drawArrow(base, vec) { push(); translate(base.x, base.y); line(0, 0, vec.x, vec.y); push(); translate(vec.x, vec.y); rotate(vec.heading()); let arrowSize = 4; noStroke(); fill(0, 150, 0); triangle(0, arrowSize/2, 0, -arrowSize/2, arrowSize, 0); pop(); pop(); } function drawRotationAnimation() { // At each grid point, animate a small purple dot revolving around a faint circle. let spacing = 1.0; // domain grid spacing let pixelScale = (uvRight - uvLeft) / 10; let R = 5; // radius (in pixels) for the animated circle for (let x = -5; x <= 5; x += spacing) { for (let y = -5; y <= 5; y += spacing) { let pos = createVector(x, y); // Local curl: for F(x,y) = (x+y, x^2-y^2), curl = 2x - 1. let curl = 2 * pos.x - 1; let angularSpeed = 0.05 * curl; let angle = frameCount * angularSpeed; let dx = R * cos(angle); let dy = R * sin(angle); let screenPos = domainToScreen(pos); noFill(); stroke(200, 0, 200, 150); ellipse(screenPos.x, screenPos.y, 2*R, 2*R); noStroke(); fill(200, 0, 200); ellipse(screenPos.x + dx, screenPos.y + dy, 4, 4); } } } function drawDomainSquare() { // Compute square corners in domain coordinates. let half = squareSize / 2; let bl = createVector(squareCenter.x - half, squareCenter.y - half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y + half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); let corners = [bl, br,
```

```
tr, tl]; // Draw blue square. stroke(0, 0, 255); noFill(); beginShape(); for (let
pt of corners) { let screenPt = domainToScreen(pt); vertex(screenPt.x, screenPt.y);
} endShape(CLOSE); // Draw draggable center (green dot). fill(0, 255, 0); let
centerScreen = domainToScreen(squareCenter); noStroke(); ellipse(centerScreen.x,
centerScreen.y, 10, 10); // Label each corner with its (x,y) coordinates. fill(0);
noStroke(); for (let pt of corners) { let screenPt = domainToScreen(pt); let label
= '(' + nf(pt.x, 1, 2) + ', ' + nf(pt.y, 1, 2) + ')'; text(label, screenPt.x + 5,
screenPt.y - 5); } } function drawFluxAnimation() { // Animate blinking flux arrows
along the square's boundary. let half = squareSize / 2; // Define square corners in
domain coordinates. let bl = createVector(squareCenter.x - half, squareCenter.y -
half); let br = createVector(squareCenter.x + half, squareCenter.y - half); let tr
= createVector(squareCenter.x + half, squareCenter.y + half); let tl =
createVector(squareCenter.x - half, squareCenter.y + half); // For an axis-aligned
square, the outward normals are fixed: let edges = [ { start: bl, end: br, normal:
createVector(0, -1) }, // bottom edge: normal points downward { start: br, end: tr,
normal: createVector(1, 0) }, // right edge: normal points rightward { start: tr,
end: tl, normal: createVector(0, 1) }, // top edge: normal points upward { start:
tl, end: bl, normal: createVector(-1, 0) } // left edge: normal points leftward ];
// For each edge, sample points and draw a blinking arrow. for (let e of edges) {
let numSamples = 5; for (let i = 0; i <= numSamples; i++) { let t = i / numSamples;
let pos = p5.Vector.lerp(e.start, e.end, t); // Evaluate vector field F(x,y) =
(x+y, x^2-y^2) at pos. let F = createVector(pos.x + pos.y, pos.x * pos.x - pos.y *
pos.y); // Compute flux (normal component) at pos. let flux = F.dot(e.normal); //
Create a blinking effect using sine modulation. let blink = abs(sin(frameCount *
0.1)); // Arrow length scales with the flux and blink effect. let arrowLength =
flux * 0.2 * blink; // Add an oscillating “eruption” offset. let erupt =
sin(frameCount * 0.2) * 5; let totalLength = arrowLength + erupt; let arrowVec =
p5.Vector.mult(e.normal, totalLength); // Convert the domain position to screen
coordinates. let base = domainToScreen(pos); push(); translate(base.x, base.y);
stroke(255, 150, 0); fill(255, 150, 0); line(0, 0, arrowVec.x, arrowVec.y); push();
translate(arrowVec.x, arrowVec.y); rotate(arrowVec.heading()); let arrowSize = 3;
noStroke(); triangle(0, arrowSize, 0, -arrowSize, arrowSize, 0); pop(); pop(); } }
} function drawTransformedRegion() { // Compute square corners in domain
coordinates. let half = squareSize / 2; let bl = createVector(squareCenter.x -
half, squareCenter.y - half); let br = createVector(squareCenter.x + half,
squareCenter.y - half); let tr = createVector(squareCenter.x + half, squareCenter.y
+ half); let tl = createVector(squareCenter.x - half, squareCenter.y + half); //
Transform corners: F(x,y) = (x+y, x^2-y^2). let f_bl = transformPoint(bl); let f_br
= transformPoint(br); let f_tr = transformPoint(tr); let f_tl = transformPoint(tl);
// Draw the red transformed region. stroke(255, 0, 0); fill(255, 100, 100, 150);
beginShape(); vertex(transformToScreen(f_bl).x, transformToScreen(f_bl).y);
vertex(transformToScreen(f_br).x, transformToScreen(f_br).y);
vertex(transformToScreen(f_tr).x, transformToScreen(f_tr).y);
vertex(transformToScreen(f_tl).x, transformToScreen(f_tl).y); endShape(CLOSE); //
Label each transformed corner. fill(0); noStroke(); let transformedCorners = [f_bl,
f_br, f_tr, f_tl]; for (let pt of transformedCorners) { let screenPt =
transformToScreen(pt); let label = '(' + nf(pt.x, 1, 2) + ', ' + nf(pt.y, 1, 2) +
')'; text(label, screenPt.x + 5, screenPt.y - 5); } } function displayDivergence()
{ // For F(x,y) = (x+y, x^2-y^2), divergence = 1 - 2y (evaluated at squareCenter).
let div = 1 - 2 * squareCenter.y; fill(0); noStroke(); let msg = 'Divergence: ' +
nf(div, 1, 2); if (div > 0) { msg += ' > (expansion)'; } else if (div < 0) { msg +=
' < (contraction)'; } else { msg += ' = (neutral)'; } text(msg, xyLeft + 10, xyTop
+ 40); } function computeCirculation() { // Compute circulation (line integral)
along the square boundary using a discrete sum. let half = squareSize / 2; let bl =
createVector(squareCenter.x - half, squareCenter.y - half); let br =
createVector(squareCenter.x + half, squareCenter.y - half); let tr =
createVector(squareCenter.x + half, squareCenter.y + half); let tl =
createVector(squareCenter.x - half, squareCenter.y + half); let edges = [[bl, br],
[br, tr], [tr, tl], [tl, bl]]; let circulation = 0; let steps = 20; // sub-
intervals per edge for (let edge of edges) { let start = edge[0]; let end =
edge[1]; let d = p5.Vector.sub(end, start); let dt = 1 / steps; for (let i = 0; i <
steps; i++) { let t = i * dt; let pos = p5.Vector.lerp(start, end, t); let ds =
d.copy().mult(dt); // Evaluate F(x,y) = (x+y, x^2-y^2) at pos. let F =
createVector(pos.x + pos.y, pos.x * pos.x - pos.y * pos.y); circulation +=
F.dot(ds); } } return circulation; } function displayCirculation() { let circ =
computeCirculation(); // Predicted circulation via Green's theorem: (curl at
center)*(area). // Here curl F = 2x - 1 evaluated at squareCenter, and area =
squareSize^2. let predicted = (2 * squareCenter.x - 1) * (squareSize * squareSize);
fill(0); noStroke(); let msg = 'Circulation (line integral): ' + nf(circ, 1, 2);
msg += ' | Predicted (curl×Area): ' + nf(predicted, 1, 2); text(msg, xyLeft + 10,
xyTop + 60); } function drawLabels() { fill(0); noStroke(); text('Domain (x,y) with
F(x,y) = (x+y, x^2-y^2), vector field & rotation', uvLeft, uvTop - 20);
text('Transformed Region, Divergence, & Curl Circulation', xyLeft, xyTop - 20); }
// Mapping: from domain (x,y) in [-5,5] to screen coordinates in the left panel.
```

```
function domainToScreen(pt) { let sx = map(pt.x, -5, 5, uvLeft, uvRight); let sy =
map(pt.y, -5, 5, uvBottom, uvTop); return createVector(sx, sy); } //
Transformation: F(x,y) = (x+y, x^2-y^2). function transformPoint(pt) { return
createVector(pt.x + pt.y, pt.x * pt.x - pt.y * pt.y); } // Mapping for transformed
region (assume x' ∈ [-10,10] and y' ∈ [-25,25]). function transformToScreen(pt) {
let sx = map(pt.x, -10, 10, xyLeft, xyRight); let sy = map(pt.y, -25, 25, xyBottom,
xyTop); return createVector(sx, sy); } function mouseDragged() { // Allow dragging
the green center dot within the domain panel. if (mouseX >= uvLeft && mouseX <=
uvRight && mouseY >= uvTop && mouseY <= uvBottom) { let mousePt =
screenToDomain(mouseX, mouseY); if (dist(mousePt.x, mousePt.y, squareCenter.x,
squareCenter.y) < 1) { squareCenter.set(mousePt.x, mousePt.y); } } } function
screenToDomain(x, y) { let u = map(x, uvLeft, uvRight, -5, 5); let v = map(y,
uvBottom, uvTop, -5, 5); return createVector(u, v); }
```

## How It Works

1. **Webcam & Panels:**
   The live webcam (via `createCapture(VIDEO)` ) is drawn with partial transparency, while two white, semi-transparent panels define the left (domain) and right (transformed) regions.
2. **Vector Field & Local Rotation:**
   The left panel shows the vector field $F(x, y) = (x + y,\ x^2 - y^2)$ as green arrows on a grid. At each grid point, a small purple dot rotates with an angular speed proportional to the local curl $2x - 1$ to visualize local circulation.
3. **Interactive Square:**
   A blue, axis-aligned square (with corner coordinates labeled) is drawn in the domain. Its center (marked in green) is draggable, and a slider adjusts its side length.
4. **Flux Animation on the Boundary:**
   For each edge of the square, several sample points are taken. At each point the flux (i.e. the normal component $F \cdot n$) is computed and used to modulate a blinking, pulsing arrow (in orange) that "erupts" from the boundary along its outward normal. This animation visually represents how much of the vector field is "engaged" in crossing the square's boundary.
5. **Transformed Region, Divergence & Circulation:**
   The red region in the right panel shows the image of the square under $F$, with labels. The divergence (at the square's center) and the circulation (both numerically computed and predicted via curl×area) are displayed.

Experiment by moving the green dot and adjusting the slider. The blinking flux arrows along the square's boundary will pulse in strength and direction—demonstrating how much flow (or flux) crosses the boundary at each point.

Enjoy exploring this dynamic visualization of vector flux, circulation, and local rotation!