Linear Geometric Transformations

December 26, 2025

Team Members

Name	ID
Omar Mohamed Abd-El-Mohsen	24010467
Youssef Diaa Abd-El-Mohsen	24010851
Mohamed Saber Abbas	24010617
Abd-El-Rahman Mohamed Mohamed	24010397
Fares Tahseen Mohamed	24010493
Youssef Shaban Abd-El-Wahed	24010849

1 2D Geometric Linear Transformations: Theory and Visualizations

This notebook explores 2D linear geometric transformations from both theoretical and practical perspectives. We introduce the matrix representations of common transformations—rotation, scaling and reflection derive their properties, and show how composition of these matrices work. Emphasis is placed on intuition (what a matrix does to points and vectors) and clear visualizations.

What we will discuss: - How to construct and interpret 2×2 transformation matrices. - Interactive visualizations and code examples for building and combining transformations.

How the notebook is organized: 1. Mathematical foundations and matrix forms for common transforms. 2. Plotting with matplotlib.

Prerequisites: basic familiarity with Python, NumPy, and linear algebra.

1.1 Mathematical foundations and matrix forms for common transforms:

Any linear transformation can be summarized by where the basis vectors of the domain subspace land after applying the transformation. — linearity means it **preserves vector addition and scalar multiplication**. **We will try to understand this statement by giving a concrete example**

Linearity Definition:

$$T\left(\sum_{i=1}^{n} c_i \mathbf{v}_i\right) = \sum_{i=1}^{n} c_i \, T(\mathbf{v}_i)$$

For the standard basis:

$$T\left(\sum_{i=1}^{n} c_i \, \mathbf{e}_i\right) = \sum_{i=1}^{n} c_i \, T(\mathbf{e}_i)$$

Now to see a concrete example, consider the 90° counterclockwise rotation.

Suppose

$$T(\mathbf{e}_1) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \qquad T(\mathbf{e}_2) = \begin{pmatrix} -1 \\ 0 \end{pmatrix}.$$

For a general vector $\mathbf{v} = x\mathbf{e}_1 + y\mathbf{e}_2$, linearity gives

$$T(\mathbf{v}) = x\,T(\mathbf{e}_1) + y\,T(\mathbf{e}_2) = x \begin{pmatrix} 0 \\ 1 \end{pmatrix} + y \begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}.$$

Writing this as $T(\mathbf{v}) = A\mathbf{v}$ we see A has columns $T(\mathbf{e}_1)$ and $T(\mathbf{e}_2)$, so

$$A = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

For a general vector $\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix}$,

$$T(\mathbf{v}) = A\mathbf{v} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}.$$

As we have shown the we transformed our original vector Equation into an Equivalent matrix Equation to make our notation easier.

Now we have the basic idea let's see common linear transformations:

1.For scaling by factors (s_x) and (s_y):

$$A = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}, \qquad A \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \end{pmatrix}.$$

For uniform scaling (s):

$$A = sI = \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix}.$$

For reflection through the y-axis:

$$A = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad A \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -x \\ y \end{pmatrix}.$$

For Rotation by phi degrees:

$$R(\varphi) = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix}, \qquad R(\varphi) \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\varphi - y\sin\varphi \\ x\sin\varphi + y\cos\varphi \end{pmatrix}.$$

Composition of linear transforms is just matrix multiplication.

- If A and B represent linear maps T_A and T_B, then

  $$(T_B \circ T_A)(\mathbf{v}) = T_B(T_A(\mathbf{v})) = B(A\mathbf{v}) = (BA)\mathbf{v},$$

  so the rightmost matrix is applied first.

Why this works (briefly): a linear map is determined by its action on the standard basis. Let $\mathbf{e}_i$ be the basis and let $A = [\mathbf{a}_1 \ \mathbf{a}_2]$ where $\mathbf{a}_i = A\mathbf{e}_i$ are the columns of A. For any $\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2$,

$$A\mathbf{v} = v_1\mathbf{a}_1 + v_2\mathbf{a}_2.$$

Applying B gives

$$B(A\mathbf{v}) = v_1B\mathbf{a}_1 + v_2B\mathbf{a}_2,$$

so the composite map has matrix whose columns are $B\mathbf{a}_i$, i.e.

$$BA = [B\mathbf{a}_1 \ \ B\mathbf{a}_2].$$

Thus composition corresponds exactly to multiplying matrices (apply rightmost first). Short takeaway: composition = multiply matrices; columns of the product are the image under the left matrix of the columns of the right matrix.

That's it for the mathematical foundation part now let's see this theory in action.

1.2 Visualizing Transformations: Constructing the "Frog"

Before applying linear transformations, we need a geometric object to transform. In this cell, we define a character (a Frog) by breaking it down into component polygons: **Legs, Stomach, Body, Smile, Eyes and Iris**.

Matrix Representation of Shapes: In our code, each geometric part is represented as an $N \times 2$ NumPy array (matrix). Each row corresponds to a vertex coordinate (x, y) on the 2D plane:

$$\text{Part} = \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}$$

Why this shape? Standard linear algebra textbooks often represent a single vector as a column $\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix}$. However, in data science and computer graphics libraries (like NumPy or OpenGL), it is often more efficient to store a list of N points as an $N \times 2$ matrix.

Procedural Symmetry (The Right Leg): Notice that we do not manually define the coordinates for `RightLeg`. Since the frog is symmetric, we calculate the right leg by reflecting the `LeftLeg` across the vertical line $x = 13$.

The mathematical formula for reflecting a coordinate x across a line $x = c$ is:

$$x' = c - (x - c) = 2c - x$$

Substituting our axis of symmetry $c = 13$:

$$x_{new} = 2(13) - x_{old} = 26 - x_{old}$$

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

LeftLeg = np.array([[2, 2], [2, 3], [4.25, 3], [2, 4], [2, 5], [3, 6], [4, 6],␣
 ↪[9, 2], [2, 2]])

# reflect about line x=13: x' = 2*13 - x = 26 - x
RightLeg = LeftLeg.copy()
RightLeg[:, 0] = 26 - RightLeg[:, 0]

Stomach = np.array([[9, 2], [8, 5], [8, 8], [10, 9], [16, 9], [18, 8], [18, 5],␣
 ↪[17, 2]])

# the body moves around the stomach and doesn't include it
Body = np.array([[9, 2],[8, 5],[8, 8],[10, 9],[16, 9],[18, 8],[18, 5],[17, 2],
        [22, 6],[23, 8],[23, 11],[20, 15],[20, 17],[18, 19],[15, 19],[14,␣
 ↪17],[14, 16],
        [12, 16],[12, 17],[11, 19],[8, 19],[6, 17],[6, 15],[3, 11],[3, 8],[4,␣
 ↪6]])

Smile = np.array([[6, 13],[7, 12],[19, 12],[20, 13]])

EyeLeft = np.array([[8, 14], [7, 15], [7, 17], [8, 18], [10, 18], [11, 17],␣
 ↪[11, 15], [10, 14]])

EyeLeftIris = np.array([[8, 16], [8, 17], [9, 17], [9, 16]])

EyeRight = EyeLeft.copy()
EyeRight[:, 0] = 26 - EyeRight[:, 0]

EyeRightIris = np.array([[16, 16],[16, 17],[17, 17],[17, 16]])
```

1.3 Symbolic Definition of Transformation Matrices

In this cell, we bridge the gap between **symbolic mathematics** and **numerical computation**.

We use the **SymPy** library to define our transformation matrices exactly as they appear in linear algebra textbooks. This allows us to work with variables like ϕ (phi) for angles or x, y for scale factors without assigning them specific numbers yet.

1.3.1 Defining the Matrices Symbolically

We define the standard 2×2 linear transformation matrices:

- **Reflection across Y-axis:** Negates the x-coordinate.

$$M_{RefY} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

- **Reflection across X-axis:** Negates the y-coordinate.

$$M_{RefX} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- **Rotation:** Rotates counterclockwise by an angle ϕ.

$$R(\phi) = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix}$$

- **Scaling:** Scales x by s_x and y by s_y.

$$S(s_x, s_y) = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

[2]:
```
from sympy import *
from sympy.abc import phi, x, y

# Define transformation matrices symbolically
ReflectionY = Matrix([[-1, 0], [0, 1]])  # Reflects across y-axis (negates x)

ReflectionX = Matrix([[1, 0], [0, -1]])  # Reflects across x-axis (negates y)

ReflectionOrigin = Matrix([[-1, 0], [0, -1]])  # Reflects through origin␣
 ↪(negates both)
RotatePhi = Matrix(
    [[cos(phi), -sin(phi)], [sin(phi), cos(phi)]]
)  # Counterclockwise rotation

ScaleS = Matrix([[x, 0], [0, y]])  # Scale by x in x-direction, y in y-direction
```

1.3.2 "Lambdifying" for Performance

While SymPy is great for algebra, it is too slow for transforming thousands of points for animation or plotting. We use `lambdify` to convert these symbolic matrices into high-performance **NumPy functions**.

For example, `RotatePhi_f` becomes a Python function that takes a number (radians) and returns a 2×2 NumPy array of floats.

[3]:
```
# Lambdify the matrices for numerical computation

# ReflectionY_f takes no arguments, returns the matrix
ReflectionY_f = lambdify([], ReflectionY, "numpy")
# ReflectionX_f takes no arguments, returns the matrix
ReflectionX_f = lambdify([], ReflectionX, "numpy")
# ReflectionOrigin_f takes no arguments, returns the matrix
ReflectionOrigin_f = lambdify([], ReflectionOrigin, "numpy")
```

```
# RotatePhi_f takes angle phi in radians, returns the rotation matrix
RotatePhi_f = lambdify([phi], RotatePhi, "numpy")
# ScaleS_f takes scale factors (sx, sy), returns the scaling matrix
ScaleS_f = lambdify([x, y], ScaleS, "numpy")
```

1.3.3 Applying the Transformation

This function `apply_transformation(part, transformation_matrix)` applies a 2×2 linear transformation to a geometric part represented as an $(N \times 2)$ array.

Parameters: * `part`: A NumPy array of shape $(N, 2)$, where each row represents a point (x, y). * `transformation_matrix`: A NumPy array of shape $(2, 2)$ representing the linear map A.

Mathematical Explanation: Standard linear algebra usually defines transformations on column vectors: $\mathbf{v}_{new} = A\mathbf{v}_{old}$. However, our points are stored as **row vectors** in an $N \times 2$ matrix P. To apply the transformation A to every row simultaneously, we must multiply by the **transpose** of A from the right:

$$P_{new} = P \cdot A^T$$

Dimensional Analysis:

$$\underbrace{(N \times 2)}_{\text{N Points}} \cdot \underbrace{(2 \times 2)}_{\text{Matrix}^T} = \underbrace{(N \times 2)}_{\text{Transformed points}}$$

[4]:
```
def apply_transformation(part, transformation_matrix):
    """
    Apply a 2x2 linear transformation to a part represented as an (N x 2) array.

    Parameters
    - part: numpy array of shape (N, 2), each row is a point (x, y).
    - transformation_matrix: numpy array of shape (2, 2).

    Explanation:
    - Points are stored as row vectors (1 x 2). To apply a 2x2 matrix A
      (which expects column vectors), we compute part @ A.T.
      This multiplies each row by the matrix correctly:
        (1x2) @ (2x2).T -> (1x2) @ (2x2) = (1x2)
    - The function returns a new (N x 2) array of transformed points.
    """
    return part @ transformation_matrix.T
```

1.4 Rendering the Frog: Helper Functions & Assembly

In this cell, we define the rendering pipeline that converts our numerical data into a visual plot.

1. **Drawing Helpers (`draw_part`, `fill_part`):** These functions abstract away Matplotlib's syntax. `draw_part` handles the logic for closing shapes (connecting the last point back to the first) to ensure outlines are complete.

[5]:
```
def draw_part(ax, part, closed=False, **kwargs):
    x = part[:, 0]
    y = part[:, 1]
    if closed:
        x = np.append(x, x[0])
        y = np.append(y, y[0])
    ax.plot(x, y, **kwargs)


def fill_part(ax, part, color, **kwargs):
    ax.fill(part[:, 0], part[:, 1], color=color, **kwargs)
```

2. **The Transformation Pipeline (`apply_full_transform`):** *(This part will be used for translating the house more on that later)*

   This function implements a full **Affine Transformation**(These are not linear transformations). It combines the linear transformation (matrix multiplication) with a translation (shifting):

$$\text{Transformed} = (\text{Original} \cdot A^T) + \begin{pmatrix} dx & dy \end{pmatrix}$$

   This allows us to rotate/scale the frog *and* move it to a specific location on the plot.

[6]:
```
# APPLY TRANSFORM


def apply_full_transform(part, matrix, dx=0, dy=0):
    transformed = part @ matrix.T
    transformed[:, 0] += dx
    transformed[:, 1] += dy
    return transformed
```

3. **Assembly (`draw_complete_frog`):** This high-level function acts as a container. It groups all the individual body parts (Legs, Eyes, Stomach, etc.), applies the same transformation to each part individually.

[ ]:
```
# DRAW COMPLETE FROG

def draw_complete_frog(ax, matrix, dx=0, dy=0):
    def tr(p):
        return apply_full_transform(p, matrix, dx, dy)

    style = dict(color="black", linewidth=1.5)

    # Filled parts
    fill_part(ax, tr(Body), color="#90BCA0", edgecolor="black", linewidth=1.5)
    fill_part(ax, tr(Stomach), color="#B5C55C", edgecolor="black", linewidth=1.
 ↪5)
    fill_part(ax, tr(EyeLeft), color="black")
    fill_part(ax, tr(EyeLeftIris), color="grey")
```

```
    fill_part(ax, tr(EyeRight), color="black")
    fill_part(ax, tr(EyeRightIris), color="white")
    fill_part(ax, tr(LeftLeg), color="#90BCA0")
    fill_part(ax, tr(RightLeg), color="#90BCA0")

    # Lines
    draw_part(ax, tr(LeftLeg), closed=True, **style)
    draw_part(ax, tr(RightLeg), closed=True, **style)
    draw_part(ax, tr(Smile), **style)
```

1.5 Visualizing the Results

In this cell, we bring everything together to visualize the linear transformations.

1.5.1 Setting up the Cartesian Plane

To properly visualize geometric transformations, we need a coordinate system that looks like a standard mathematical graph: * **Equal Aspect Ratio:** `ax.set_aspect("equal")` ensures that 1 unit on the x-axis is the same physical length as 1 unit on the y-axis. Without this, rotations would look distorted (e.g., a square rotated 45° might look like a diamond or rectangle).

- **Centered Spines:** We move the plot spines (the borders) to the center $(0, 0)$ to create a traditional x, y crosshair axis, rather than the default box style used in statistical plots.

1.5.2 Drawing the Frogs

We draw three variations of the frog to demonstrate different linear maps:

1. **Original (Center/Right):** We apply the **Identity Matrix** (I).

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

   This leaves the frog unchanged.

2. **Scaled & Reflected (Top Left):** We compose scaling and reflection matrices:

$$T_1 = S(0.5, 0.5) \cdot M_{RefY}$$

   - $S(0.5, 0.5)$: Shrinks the frog to half size.
   - M_{RefY}: Flips it across the vertical axis.

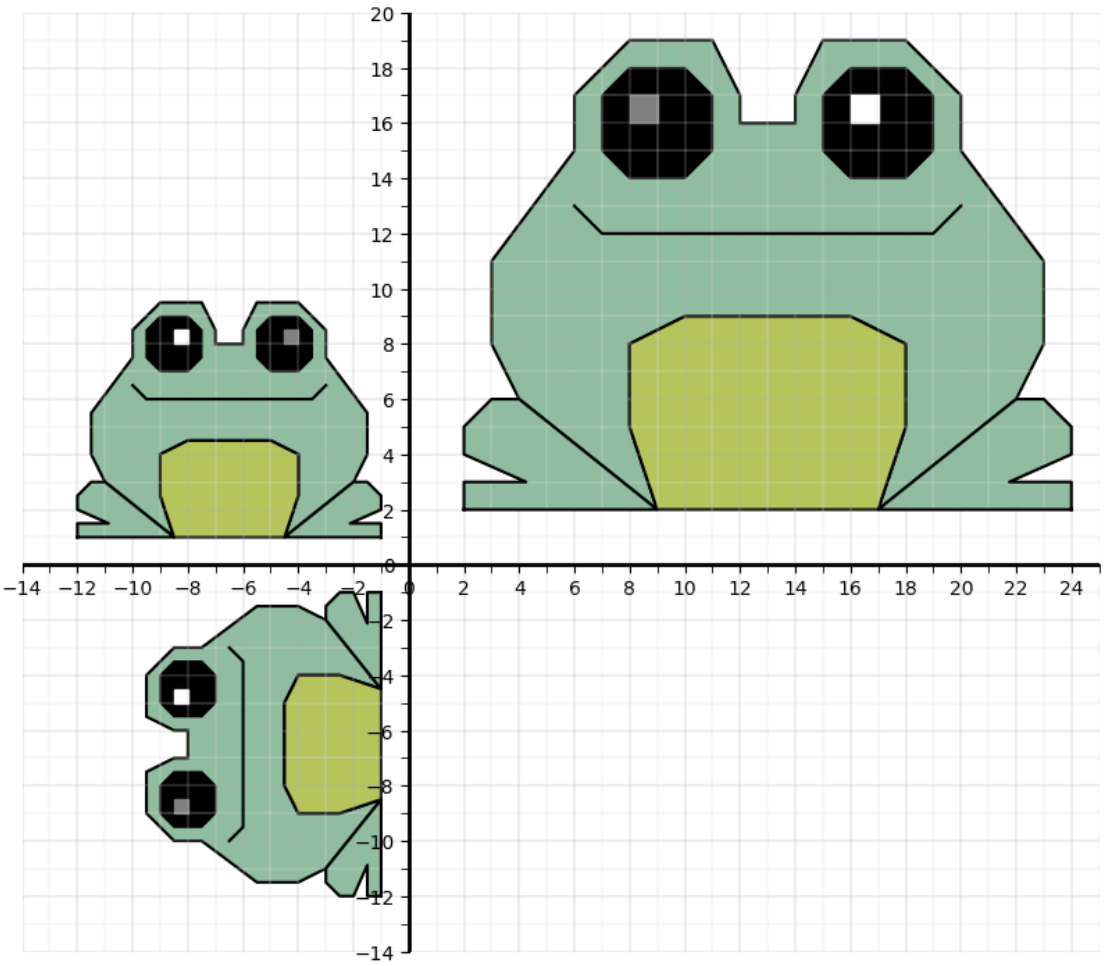
3. **Scaled, Rotated & Translated (Bottom Left):**

$$T_2 = S(0.5, 0.5) \cdot R(90^\circ)$$

   - We rotate the frog 90° counter-clockwise.
   - We then apply a translation $(dx = 0, dy = -13)$

[8]:
```
# SETUP CARTESIAN PLANE
fig, ax = plt.subplots(figsize=(10, 10))
```

```
ax.set_xlim(-14, 25)
ax.set_ylim(-14, 20)

ax.set_aspect("equal", adjustable="box")

# Major ticks (every 2 units)
ax.set_xticks(np.arange(-14, 26, 2))
ax.set_yticks(np.arange(-14, 21, 2))

# Minor ticks (every 1 unit)
ax.set_xticks(np.arange(-14, 26, 1), minor=True)
ax.set_yticks(np.arange(-14, 21, 1), minor=True)

# Grid
ax.grid(which="major", color="#b0b0b0", linewidth=0.3)
ax.grid(which="minor", color="#e0e0e0", linewidth=0.3)

# Axes through origin
ax.axhline(0, color="black", linewidth=2)
ax.axvline(0, color="black", linewidth=2)

# Move spines to origin (true Cartesian look)
ax.spines["left"].set_position("zero")
ax.spines["bottom"].set_position("zero")
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")

ax.tick_params(axis="both", which="both", direction="out", length=4)

# DRAW FROGS

#  Original (Top Right / Center)
draw_complete_frog(ax, np.eye(2), dx=0, dy=0)

#  Scaled + Reflected (Top Left)
T1 = ScaleS_f(0.5, 0.5) @ ReflectionY_f()
draw_complete_frog(ax, T1, dx=0, dy=0)

# Scaled + Rotated 90° Clockwise (Bottom Left)

T2 = ScaleS_f(0.5, 0.5) @ RotatePhi_f(np.pi / 2)
draw_complete_frog(ax, T2, dx=0, dy=-13)

plt.show()
```



1.6 Defining the House Shape

We define a simple polygon (a House) using an $N \times 2$ NumPy array, following the same convention as the Frog components. Each row represents a vertex coordinate (x, y)

[ ]:
```
Home = np.array(
    [[1, 1],[1, 3],[0, 3],[3, 5],[4, 4.3],[4, 4.5],
     [4.5, 4.5],[4.5, 4],[6, 3],[5, 3],[5, 1]])
```

1.6.1 Drawing the House

This function `draw_house` is a simplified version of the frog drawing logic. Since the house is a single polygon, we don't need to group multiple parts.

[10]:
```
def draw_house(ax, matrix, dx=0, dy=0, color="black", linewidth=2,␣
 ↪fill_color=False):
    transformed = apply_full_transform(Home, matrix, dx, dy)
```

```
    if fill_color:
        fill_part(
            ax, transformed, color=fill_color, edgecolor=color,␣
↪linewidth=linewidth
        )
    else:
        draw_part(ax, transformed, closed=True, color=color,␣
↪linewidth=linewidth)

    # Draw dots at vertices
    ax.plot(transformed[:, 0], transformed[:, 1], "ks", markersize=6)
```

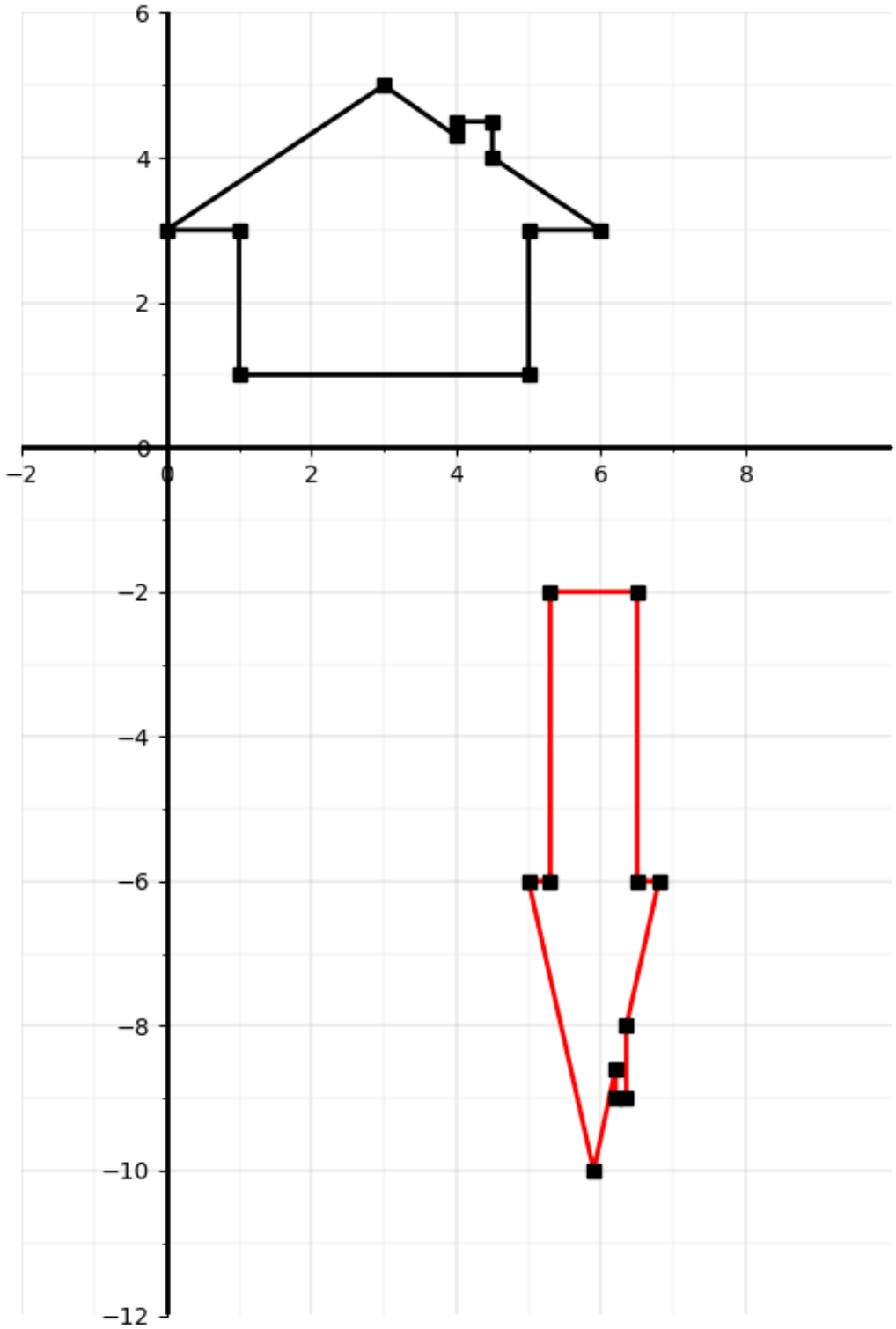
1.6.2 Visualizing the House Transformation

Finally, we visualize the house transformation:

1. **Setup:** We configure the Cartesian plane (limits, grid, centered axes) similar to the frog example.
2. **Original House (Black):** Drawn using the identity matrix.
3. **Transformed House (Red):**
   - **Scale:** x is scaled by 0.3 (squashed), y by 2 (stretched).
   - **Reflection:** Reflected across thx x-axis (flipped upside down).
   - **Translation**: Shifted right by 5 units (dx=5).

[11]:
```
# Setup plot
fig, ax = plt.subplots(figsize=(10, 10))

ax.set_xlim(-2, 10)
ax.set_ylim(-12, 6)
ax.set_aspect("equal", adjustable="box")

# Major and minor ticks
ax.set_xticks(np.arange(-2, 9, 2))
ax.set_yticks(np.arange(-12, 7, 2))
ax.set_xticks(np.arange(-2, 9, 1), minor=True)
ax.set_yticks(np.arange(-12, 7, 1), minor=True)

# Grid
ax.grid(which="major", color="#b0b0b0", linewidth=0.3)
ax.grid(which="minor", color="#e0e0e0", linewidth=0.3)

# Axes
ax.axhline(0, color="black", linewidth=2)
ax.axvline(0, color="black", linewidth=2)
```

```
ax.spines["left"].set_position("zero")
ax.spines["bottom"].set_position("zero")
ax.spines["right"].set_color("none")
ax.spines["top"].set_color("none")

# Draw original house (black, top)
draw_house(ax, np.eye(2), dx=0, dy=0, color="black", linewidth=2)

# Draw transformed house (red, bottom)
# Scale down, reflect over x-axis, then translate
T = ScaleS_f(0.3, 2) @ ReflectionX_f()
draw_house(ax, T, dx=5, dy=0, color="red", linewidth=2)

plt.show()
```


6
4
2
0
−2
0
2
4
6
8
−2
−4
−6
−8
−10
−12