Understanding Straight Line forms: Analytical Geometry and beyond (the case of Linear Programming)

Jose Portillo

Abstract: This easy walk the path of Descartes himself armed with algebra and geometry: extract pure spatial meaning from symbolic expressions. No measuring tape, no compass, no guesswork. Just: equations, logic and system solving. This easy was written with a little help from an AI GPT making this a use case of human subject using the AI as copilot for math syntax checking and LaTeX formatting.

1 Straight Line Equations: Forms and Applications

A straight line in the Cartesian plane can be represented in several forms, each with its own advantages and applications. The most common forms include:

1.1 Slope-Intercept Form

The slope-intercept form is given by

$$y = mx + b, \tag{1}$$

where m is the slope of the line and b is the y-intercept. This form is especially convenient for graphing since it directly shows how the line behaves with respect to the y-axis.

1.2 General Form

The general form of a line is written as

$$Ax + By + C = 0, \tag{2}$$

where A, B, and C are real constants, and A and B are not both zero. This implicit form is very powerful because it can represent all lines, including vertical lines which cannot be expressed in slope-intercept form due to undefined slope.
From the general form, the slope m can be recovered as

$$m = -\frac{A}{B}, \quad \text{provided } B \neq 0. \tag{3}$$

1.3 Point-Slope Form

Given two points (x_1, y_1) and (x_2, y_2), the slope is

$$m = \frac{y_2 - y_1}{x_2 - x_1}. \tag{4}$$

The point-slope form of the line passing through (x_1, y_1) is

$$y - y_1 = m(x - x_1). \tag{5}$$

Note: the coordinates x and y are not a second fixed point, but rather variables representing ANY point on the line. The pair (x_1, y_1) is the GIVEN, fixed point used to construct the line, and m is the known slope. Once these are known, the equation defines all possible (x, y) points that lie along the same straight path.

1.4 Why Use General Form?

While the slope-intercept form is intuitive for graphing, the general form is indispensable for:

- Representing vertical lines ($x = k$) where slope is undefined.
- Algebraic manipulation and solving systems of linear equations.
- Applications in higher dimensions (planes and hyperplanes).
- Implicit representation of lines, useful in computational geometry.

1.5 Example: Constructing and Solving a Triangle System

Consider three points in the plane:

$$A(1, 2), \quad B(4, 3), \quad C(2, 5).$$

1.6 Checking for Collinearity Before Forming a Triangle

Before using three points to define a triangle, we must first confirm that they are not collinear — that is, they do not all lie on the same straight line. If they are collinear, they cannot enclose any area, and thus, no triangle can be formed.
Given three points $A(x_1, y_1)$, $B(x_2, y_2)$, and $C(x_3, y_3)$, we can check for collinearity using the area formula for a triangle derived from the determinant:

$$\text{Area} = \frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \tag{6}$$

This evaluates to:

$$\text{Area} = \frac{1}{2}\left|x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)\right| \tag{7}$$

If this area is zero, then the points are collinear — geometrically aligned — and no triangle exists.
This check is essential before proceeding with triangle-based constructions or region definitions in linear programming, as degeneracies (like zero-area shapes) can lead to incorrect or undefined results.
Back to our three point example

$$\text{Area} = \frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = \frac{1}{2}\left|1(3-5) - 4(2-5) + 2(2-3)\right| = \frac{1}{2}\left|-2+12-2\right| = \frac{1}{2}\cdot 8 = 4. \tag{8}$$

Since the area is non-zero, the points are not collinear and do form a triangle.
We seek the equations of the lines forming the triangle ABC and solve their systems to recover the vertices algebraically.
Note: Collinear points have zero area and thus cannot define a triangle. Always verify this before proceeding with symbolic derivations.

Line AB: Calculate the slope:

$$m_{AB} = \frac{3-2}{4-1} = \frac{1}{3}.$$

Point-slope form using point $A(1, 2)$:

$$y - 2 = \frac{1}{3}(x - 1).$$

Solve for y:

$$y = \frac{1}{3}x + \frac{5}{3}.$$

Multiply both sides by 3 so we get the line's *General form*:

$$3y = x + 5 \implies x - 3y + 5 = 0 \quad \text{(General form)}.$$

Line BC: Calculate the slope:

$$m_{BC} = \frac{5-3}{2-4} = -1.$$

Point-slope form using point $B(4, 3)$

$$y - 3 = -1(x - 4).$$

Solve for y:

$$y = -x + 7.$$

Rewrite so we get the line's *General form*:

$$x + y - 7 = 0 \quad \text{(General form)}.$$

Line CA: Calculate the slope:

$$m_{CA} = \frac{2-5}{1-2} = 3.$$

Point-slope form using point $C(2, 5)$

$$y - 5 = 3(x - 2).$$

Solve for y:

$$y = 3x - 1.$$

Rewrite so we get the line's *General form*:

$$-3x + y + 1 = 0.$$

Solving Intersections: Intersection of AB and BC:

$$\begin{cases} x - 3y + 5 = 0, \\ x + y - 7 = 0. \end{cases}$$

Subtract second from first:

$$(x - 3y + 5) - (x + y - 7) = -4y + 12 = 0 \implies y = 3.$$

Substitute back:

$$x + 3 - 7 = 0 \implies x = 4.$$

Vertex $B = (4, 3)$.
Intersection of BC and CA:

$$\begin{cases} x + y - 7 = 0, \\ -3x + y + 1 = 0. \end{cases}$$

Subtract second from first:

$$(x + y - 7) - (-3x + y + 1) = 4x - 8 = 0 \implies x = 2.$$

Substitute back:

$$2 + y - 7 = 0 \implies y = 5.$$

Vertex $C = (2, 5)$.
Intersection of CA and AB:

$$\begin{cases} -3x + y + 1 = 0, \\ x - 3y + 5 = 0. \end{cases}$$

Multiply the second equation by 3:

$$3x - 9y + 15 = 0.$$

Add to the first:

$$(-3x+y+1)+(3x-9y+15) = -8y+16 = 0 \implies y = 2.$$

Substitute back:

$$x - 3(2) + 5 = 0 \implies x = 1.$$

Vertex $A = (1, 2)$.
Lets see a PYTHON implementation:

```
import matplotlib.pyplot as plt
import numpy as np

# Define points
A = np.array([1, 1])
B = np.array([4, 1])
C = np.array([2, 3])
points = np.array([A, B, C, A])  # Close triangle

# Function to compute triangle area using determinant
def triangle_area(p1, p2, p3):
    return 0.5 * abs(
        p1[0]*(p2[1] - p3[1]) +
        p2[0]*(p3[1] - p1[1]) +
        p3[0]*(p1[1] - p2[1])
    )

area = triangle_area(A, B, C)

if area == 0:
    print("The points are collinear | no triangle can be formed.")
else:
    print(f"Triangle area: {area:.2f} | points are NOT collinear. Proceeding...\n")

    # Function to compute line equation
    def line_equation(p1, p2):
        if p2[0] != p1[0]:
            m = (p2[1] - p1[1]) / (p2[0] - p1[0])
            b = p1[1] - m * p1[0]
            return m, b
        else:
            return np.inf, p1[0]  # Vertical line

    # Compute and print equations
    lines = {'AB': (A, B), 'BC': (B, C), 'CA': (C, A)}
    print("Line equations:")
    for name, (p1, p2) in lines.items():
        m, b = line_equation(p1, p2)
        if m == np.inf:
            print(f"{name}: vertical line at x = {b}")
        else:
            print(f"{name}: y = {m:.2f}x + {b:.2f}")

    # Plot triangle
    plt.figure(figsize=(6, 6))
    plt.plot(points[:, 0], points[:, 1], 'bo-', label='Triangle ABC')
    plt.fill(points[:, 0], points[:, 1], color='skyblue', alpha=0.3)

    # Label points
    for P, name in zip([A, B, C], ['A', 'B', 'C']):
        plt.text(P[0] + 0.1, P[1] + 0.1, name, fontsize=12, fontweight='bold')

    # Axis decoration
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid(True)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.xlim(0, 5)
    plt.ylim(0, 4)
    plt.title("Triangle from Three Points")
    plt.legend()
    plt.show()
```

1.7 Universality of the Method

This procedure can be applied to *any* three distinct, non-collinear points in the Cartesian plane. For any such triple, one can:

- Derive the line equations connecting each pair of points,
- Express those lines in general form,
- Solve the pairwise systems of equations,
- Recover the original triangle's vertices purely through algebraic means.

This approach demonstrates the power of analytic geometry: using symbolic equations and systems of linear equations to understand and reconstruct geometric figures without requiring any prior graphical or coordinate plotting.

2 Beyond the Triangle: Polygons and Linear Programming

Having established the power of line equations and algebraic systems in reconstructing a triangle, we now ask deeper questions: Can this procedure be extended to more complex geometric structures? And is this type of system solving related to techniques used in optimization problems such as Linear Programming (LP)? The answer to both is yes with important nuances.

2.1 Connection to Linear Programming (LP)

Let's now revisit our old friend, the straight line $y = mx+b$, and explore how this humble object turns into the central character in the theory of Linear Programming (LP). The idea is this: LP problems are built entirely out of straight lines — some act as boundaries, and one plays the role of the quantity we want to optimize.

In LP, we deal with:

- A collection of **linear constraints** — these are inequalities like $a_1x + a_2y \leq b$. Each one defines not just a line, but a *region*: a half-plane on one side of that line.
- A **feasible region** — this is the intersection of all those constraint regions. In two dimensions, it forms a polygon, possibly bounded (like a triangle or pentagon) or unbounded.
- A **linear objective function** — something like $z = cx + dy$ — which we aim to either maximize or minimize. This too defines a straight line, but it doesn't restrict the region. Instead, we imagine sliding this line (or its level curves) across the feasible region to find where it reaches its highest or lowest value.

Geometrically, the LP setup looks like this:

- The constraints draw lines on the plane, then pick one side (via inequality signs).
- The feasible region is what's left after taking the intersection of all those "sided" lines.
- The objective function defines a direction — and we slide it along that direction until it touches the farthest edge of the region.

So the entire LP framework is just an elegant game of straight lines. The constraints form a polygonal "arena." The objective function is a line that's allowed to float freely across the plane — and our job is to find where it last touches the arena before flying off.

In this sense, our previous exploration of finding a triangle from three intersecting lines was a kind of warm-up for LP: in LP, we also look for intersections of lines (constraints), but we care not just about the corners — we care about which corner makes us the most profit (or the least cost), according to the direction of our objective line.

2.2 Generalizing to Polygons with N Sides

Let us now generalize our triangle reconstruction method to arbitrary polygons with N sides. Suppose we have a simple, closed polygon defined not by its vertices but by the equations of its N sides. Our goal is to recover the polygonś corner points (vertices) using only algebra.

Step-by-step Procedure:

1. **Line Equations:** Begin with N linear equations, each representing one side of the polygon. These should be in general form:
$$A_i x + B_i y + C_i = 0, \quad \text{for } i = 1, 2, \ldots, N.$$
2. **Solve Intersections:** For each pair of adjacent lines L_i and L_{i+1}, solve the system:
$$\begin{cases} A_i x + B_i y + C_i = 0, \\ A_{i+1} x + B_{i+1} y + C_{i+1} = 0. \end{cases}$$
This yields the vertex V_{i+1}, the point of intersection between sides i and $i + 1$.
3. **Closing the Polygon:** To complete the loop, solve the final system between the last side and the first:
$$\begin{cases} A_N x + B_N y + C_N = 0, \\ A_1 x + B_1 y + C_1 = 0, \end{cases}$$
giving the closing vertex V_1.

Conditions and Assumptions:

For this process to work reliably:

- The polygon must be **simple** (its sides should not intersect except at the vertices).
- The sides must be ordered such that each one connects logically to the next (i.e., the lines form a loop).
- No three consecutive sides should be concurrent (i.e., intersect at a single point), and adjacent lines should not be parallel (to avoid degenerate cases).

This approach allows the polygon to be reconstructed entirely through symbolic means no coordinates required up front. Each vertex emerges naturally from the algebraic solution of systems of equations.

2.3 Conclusion

The method of using general form line equations and solving algebraic systems is more than a tool for plotting a triangle it is a gateway into a wide range of geometric and optimization problems. From hand solving triangle vertices to designing constraint systems in industrial optimization, the underlying math is the same: understand the geometry through equations, and extract structure from the intersections.