Radio Triangulation with Scratch & Python

A Guided Math and Coding Workbook

April 2025

Contents

1 Why Triangulation?

Finding an emitter's location from measurements at a distance is a classic applied-math problem. The same ideas drive GPS, seismic localization, and even wildlife-tracking collars. This workbook teaches the underlying geometry and linear-algebra step by step, then shows how to implement each formula first in Scratch (for intuition) and finally in Python + NumPy (for power and realism).

2 Angle Basics and the Need for `atan2`

When we convert Cartesian differences (Δx, Δy) into an angle we want two things:

1. an *unambiguous* angle for all four quadrants and
2. continuity at the $\pm 180°$ wrap-around.

The one-argument inverse tangent $\tan^{-1}(y/x)$ fails on both counts, while `atan2`(dy, dx) succeeds by inspecting the signs of both arguments. We formalise that idea in Equation (1) and will reference it throughout.

$$\theta \;=\; \operatorname{atan2}(y,\, x)\,; \qquad -\pi < \theta \leq \pi \tag{1}$$

Why this matters. By returning an angle on the open interval $(-\pi, \pi]$, Equation (1) produces a unique bearing no matter which quadrant the point (x, y) lies in. This eliminates the ambiguity that would otherwise plague our later algebra when we subtract bearings.

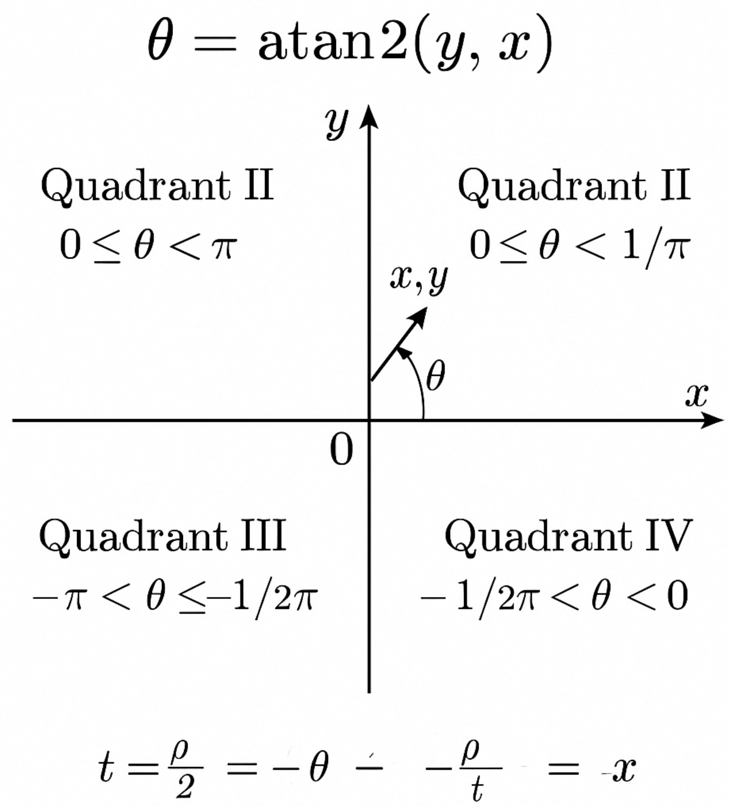

Figure 1: Quadrant handling by `atan2`. The line under the diagram, $t = \frac{\rho}{2} = \ldots$, is explained in Section 2.1.

2.1 Interpreting the $t = \frac{\rho}{2} = \ldots$ Annotation

The extra formula sometimes shown beneath quadrant diagrams originates from polar-to-Cartesian conversion proofs. Let ρ be the radial distance and θ the polar angle. Setting $t = \rho/2$ and rearranging $x = \rho\cos\theta$, $y = \rho\sin\theta$ one can show

$$t = -\theta \iff (x, y) = \left(-\frac{\rho}{t}, \ldots\right),$$

which is simply an algebraic trick for eliminating ρ. It is *not* something we will use operationally, but you may see it in textbooks; now you know where it comes from!

3 Two-Receiver Geometry in Depth

3.1 Why Can We Set $\mathbf{p}_1 = \mathbf{p}_2$?

Each receiver's line-of-bearing (LOB) is the set of all points that satisfy its parametric equation

$$\mathbf{p}_i(t_i) = \mathbf{r}_i + t_i\mathbf{d}_i.$$

If the transmitter lies on *both* LOBs simultaneously, then it must satisfy both equations. Therefore the true location $\mathbf{p}_{\text{true}}$ obeys

$$\mathbf{p}_{\text{true}} = \mathbf{p}_1(t_1) = \mathbf{p}_2(t_2).$$

Replacing $\mathbf{p}_{\text{true}}$ by our estimated $\hat{\mathbf{p}}$ gives us two linear equations in the two unknowns t_1 and t_2. Solving those yields $\hat{\mathbf{p}}$. The algebraic steps follow, and each one is annotated with its geometric meaning.

3.2 Step-by-Step Elimination

Start with component form. Writing out the x and y components explicitly we obtain

$$x_1 + t_1\cos\theta_1 = x_2 + t_2\cos\theta_2, \tag{2}$$
$$y_1 + t_1\sin\theta_1 = y_2 + t_2\sin\theta_2. \tag{3}$$

Interpretation. Equation (2) states that the unknown x-coordinate of the transmitter has two alternative expressions—one from each receiver. Equation (3) says the same for the y-coordinate. By equating them we force the intersection.

Eliminate t_2. Multiply Equation (2) by $\sin\theta_2$ and Equation (3) by $\cos\theta_2$ then subtract to eliminate t_2:

$$(x_1 - x_2)\sin\theta_2 + t_1(\cos\theta_1\sin\theta_2) \tag{4}$$
$$- \left[(y_1 - y_2)\cos\theta_2 + t_1(\sin\theta_1\cos\theta_2)\right] = 0. \tag{5}$$

Use a trig identity. Factoring t_1 and applying the sine-difference identity $\sin(\theta_2 - \theta_1) = \sin\theta_2 \cos\theta_1 - \cos\theta_2 \sin\theta_1$ simplifies Equation (5) to

$$t_1 \sin(\theta_2 - \theta_1) = (x_2 - x_1)\sin\theta_2 - (y_2 - y_1)\cos\theta_2. \tag{6}$$

Finally we isolate t_1 to obtain the celebrated intersection formula

$$t_1 = \frac{(x_2 - x_1)\sin\theta_2 - (y_2 - y_1)\cos\theta_2}{\sin(\theta_2 - \theta_1)}. \tag{7}$$

Why Equation (7)? The numerator measures how far the two LOBs are offset from one another, projected into receiver 2's reference frame; the denominator rescales that distance according to the angular separation between the LOBs. A small denominator (nearly parallel LOBs) inflates t_1—in other words, a shallow intersection angle greatly amplifies positional uncertainty.

3.3 Python Helper to Check Our Hand Algebra

Using symbolic math (SymPy) we can verify Equation (7) programmatically:

```
import sympy as sp
x1,x2,y1,y2,t1,t2 = sp.symbols('x1 x2 y1 y2 t1 t2', real=True)
th1, th2 = sp.symbols('th1 th2', real=True)
sol = sp.solve([
    x1 + t1*sp.cos(th1) - (x2 + t2*sp.cos(th2)),
    y1 + t1*sp.sin(th1) - (y2 + t2*sp.sin(th2))
], (t1,t2))
sp.simplify(sol[t1])  # matches Equation (t1)...
```

3.4 Visualising the Geometry (Jupyter Cell)

The following cell produces a plot of the two receivers, their LOBs, and the transmitter. Run it inside your Jupyter notebook to "see" what the algebra is doing.

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

# receiver positions and true transmitter
r = np.array([[0,0], [400,0]])
p_true = np.array([180.0, 120.0])

# bearings (deg) derived from true geometry
th = np.degrees(np.arctan2(p_true[1]-r[:,1], p_true[0]-r[:,0]))

# build LOBs for plotting
L = 500  # plot length
lob_pts = []
for rx, angle in zip(r, th):
    direction = np.array([np.cos(np.radians(angle)), np.sin(np.radians(angle))])
    t = np.linspace(0, L, 100)[:,None]
    lob_pts.append(rx + t*direction)

```

```python
plt.figure()
plt.plot(r[:,0], r[:,1], 'ko', label='Receivers')
plt.plot(p_true[0], p_true[1], 'r*', markersize=12, label='Transmitter')
for idx, line in enumerate(lob_pts):
    plt.plot(line[:,0], line[:,1], label=f'LOB {idx+1}')
plt.axis('equal')
plt.legend()
plt.title('Two-Receiver Triangulation Geometry')
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.show()
```

Interpretation. Notice how the transmitter lies exactly at the intersection of the two infinite LOBs, confirming the algebraic solution.

4 Three Receivers and Least Squares, Step By Step

4.1 Re-using the Same Vector Form

Keep each LOB in the familiar form

$$\mathbf{p}_i(t_i) = \mathbf{r}_i + t_i\mathbf{d}_i \qquad (i = 1, 2, 3).$$

Writing out x and y components of all three yields six linear equations. Rearranging gives Equation (**??**). We now dissect that equation line-by-line so you understand what every term means.

4.2 What Does $A\hat{\mathbf{p}} = \mathbf{b}$ Mean?

- A is a 3×2 matrix whose rows contain the direction-vector components.
- $\hat{\mathbf{p}} = \langle \hat{x}, \hat{y} \rangle^\mathsf{T}$ is the column vector we want.
- $\mathbf{b}$ encodes the known constants $d_{ix}x_i - d_{iy}y_i$.

If A were square we could simply invert it, but with 3 equations vs. 2 unknowns we instead find the vector that minimises the residual error. Linear-algebra shows the solution is the *normal equation*

$$\hat{\mathbf{p}} \; = \; (A^\mathsf{T} A)^{-1} A^\mathsf{T} \mathbf{b}. \tag{8}$$

Why minimising the squared error? Squared error penalises large misses more heavily than small ones and leads to a unique analytic solution for linear problems. The projection interpretation also gives geometric intuition: we are projecting $\mathbf{b}$ onto A's column space.

4.3 Walking Through $(A^\mathsf{T} A)$

Take a concrete numerical example.

```
import numpy as np
# example bearings (degrees) and receiver coords
th = np.radians([ 10, 110, 250 ])
r  = np.array([[0,0], [400,0], [200,300]])
# build A and b
A = np.column_stack((np.cos(th), -np.sin(th)))
b = np.cos(th)*r[:,0] - np.sin(th)*r[:,1]
# Manual multiplication
AtA = A.T @ A           # 2x2
Atb = A.T @ b           # 2x1
print('A^T A =\n',AtA)
print('A^T b =',Atb)
```

Explain each product:

$$(A^\mathsf{T}A)_{11} = \sum_{i=1}^{3} d_{ix}^2, \; (A^\mathsf{T}A)_{12} = \sum_{i=1}^{3}(-d_{ix}d_{iy}), \text{ etc.}$$

Because $A^\mathsf{T}A$ is symmetric we only need compute its upper triangle manually.

4.4 Finding the Inverse by Hand (2×2 Case)

For a 2×2 matrix $M = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$

$$M^{-1} = \frac{1}{ac - b^2} \begin{bmatrix} c & -b \\ -b & a \end{bmatrix}.$$

NumPy does this for us, but it is instructive to check the determinant $ac - b^2$ is non-zero, which requires the three bearings not to be co-linear.

```
det = AtA[0,0]*AtA[1,1] - AtA[0,1]**2
assert det != 0, 'Geometry is degenerate!'
inv = np.linalg.inv(AtA)
print('inv(AtA)=\n',inv)
# finally p_hat
p_hat = inv @ Atb
print('Estimated transmitter =',p_hat)
```

4.5 Why Does This Work?

The normal-equation solution arises from setting the gradient of the squared error $\|A\mathbf{p} - \mathbf{b}\|^2$ with respect to $\mathbf{p}$ to zero, which yields $A^\mathsf{T}A\mathbf{p} = A^\mathsf{T}\mathbf{b}$. That minimum-error vector is exactly Equation (8). You can read any linear-algebra text for proof, but the intuition is we are projecting $\mathbf{b}$ onto the column space of A.

4.6 Covariance Revisited

Plugging $M = (A^\mathsf{T}A)^{-1}$ into

$$\mathrm{Cov}(\hat{\mathbf{p}}) = \sigma^2 M$$

reveals how the variances scale with geometry. In Python:

```
# assume sigma = 3° measurement noise (in radians)
sigma = np.deg2rad(3)
Cov = sigma**2 * inv
print('Position covariance matrix:\n', Cov)
print('Std-dev in x, y =', np.sqrt(np.diag(Cov)))
```

Interpret those numbers: one standard deviation encloses ≈ 68% of true transmitter locations under repeated noise trials.

5 Putting It All Together

1. Build the Scratch two-receiver demo; verify the lines intersect visually.
2. Run the Python script with randomised noise to see statistical scatter.
3. Add or move receivers; observe how $A^\mathsf{T}A$ and the covariance change.

Appendix A – Full Python Listing

```
"""Full three-receiver least-squares demo."""
import numpy as np, math, random
# receiver positions
r = np.array([[0,0], [400,0], [200,300]], float)
# true transmitter (unknown to algorithm)
p_true = np.array([180.0, 120.0])
# simulate noisy bearings
sigma_deg = 3.0
th_true = np.degrees(np.arctan2(p_true[1]-r[:,1], p_true[0]-r[:,0]))
noise = np.random.normal(0, sigma_deg, size=3)
th_meas = np.radians(th_true + noise)
# build A, b
A = np.column_stack((np.cos(th_meas), -np.sin(th_meas)))
b = np.cos(th_meas)*r[:,0] - np.sin(th_meas)*r[:,1]
# solve
p_hat = np.linalg.inv(A.T @ A) @ (A.T @ b)
print('True :', p_true, '\nEst. :', p_hat)
print('Error:', np.linalg.norm(p_hat-p_true), 'pixels')
```