design a 2d system for triangulation of position detection system based on 3 microphones each giving a trigger signal when a sound is received. Please code in c

Below is a **complete 2D triangulation (actually multilateration) design** for locating a sound source using **3 microphones** that each produce a **trigger timestamp** when the sound arrives.

This is the standard **Time Difference of Arrival (TDOA)** approach.

---

1. System Overview

Inputs

- 3 microphones at **known fixed positions**:
  - Mic1: (x_1, y_1)
  - Mic2: (x_2, y_2)
  - Mic3: (x_3, y_3)
- Each microphone produces:
  - `t1`, `t2`, `t3` → trigger timestamps (seconds or microseconds)

Known Constant

- Speed of sound in air:

```c
c = 343.0 m/s   // at ~20°C
```

Output

- Estimated 2D sound source position:

```text
(x, y)
```

---

2. Mathematical Model

Distance from source `(x, y)` to each microphone:

```ini
d1 = sqrt((x - x1)² + (y - y1)²)
d2 = sqrt((x - x2)² + (y - y2)²)
d3 = sqrt((x - x3)² + (y - y3)²)
```

Since sound emission time is unknown, we use **time differences**:

```
Δt21 = t2 - t1
```

```
Δt31 = t3 - t1
```

Convert to distance differences:

r

Copy code

```
Δd21 = c * Δt21
Δd31 = c * Δt31
```

This gives two equations:

nginx

Copy code

```
d2 - d1 = Δd21
d3 - d1 = Δd31
```

These define **hyperbolas**. Solving them yields `(x, y)`.

---

3. Practical Numerical Solution

We solve using **nonlinear least squares** (robust and simple for embedded systems).

We minimize:

csharp

Copy code

```
f1(x,y) = d2 - d1 - Δd21
f2(x,y) = d3 - d1 - Δd31
```

Using **iterative Newton / gradient descent**.

---

4. C Implementation

Features

✔ Works in 2D

✔ Uses only `math.h`

✔ Embedded-friendly

✔ No dynamic memory

C Code (Complete Example)

c

Copy code

```c
#include <stdio.h>
#include <math.h>

#define SPEED_OF_SOUND 343.0
#define MAX_ITER 100
#define TOL 1e-6
#define STEP 0.1

typedef struct {
    double x;
    double y;
```

```
} Point;

/* Distance between two points */
double distance(Point a, Point b) {
    return sqrt((a.x - b.x)*(a.x - b.x) +
                (a.y - b.y)*(a.y - b.y));
}

/* Estimate sound source position */
int triangulate(Point m1, Point m2, Point m3,
                double t1, double t2, double t3,
                Point *result)
{
    double dt21 = t2 - t1;
    double dt31 = t3 - t1;

    double dd21 = SPEED_OF_SOUND * dt21;
    double dd31 = SPEED_OF_SOUND * dt31;

    /* Initial guess: centroid of microphones */
    double x = (m1.x + m2.x + m3.x) / 3.0;
    double y = (m1.y + m2.y + m3.y) / 3.0;

    for (int i = 0; i < MAX_ITER; i++) {
        Point p = {x, y};

        double d1 = distance(p, m1);
        double d2 = distance(p, m2);
        double d3 = distance(p, m3);
```

```
double f1 = (d2 - d1) - dd21;
double f2 = (d3 - d1) - dd31;

double error = fabs(f1) + fabs(f2);
if (error < TOL) {
    result->x = x;
    result->y = y;
    return 1;
}

/* Numerical gradient */
double dx = STEP;
double dy = STEP;

Point px = {x + dx, y};
Point py = {x, y + dy};

double d1x = distance(px, m1);
double d2x = distance(px, m2);
double d3x = distance(px, m3);

double d1y = distance(py, m1);
double d2y = distance(py, m2);
double d3y = distance(py, m3);

double f1x = (d2x - d1x) - dd21;
double f2x = (d3x - d1x) - dd31;
```

```
        double f1y = (d2y - d1y) - dd21;
        double f2y = (d3y - d1y) - dd31;

        double grad_x = (f1x - f1) + (f2x - f2);
        double grad_y = (f1y - f1) + (f2y - f2);

        x -= grad_x * 0.5;
        y -= grad_y * 0.5;
    }

    return 0;  // failed to converge
}

/* Example usage */
int main(void) {
    Point mic1 = {0.0, 0.0};
    Point mic2 = {5.0, 0.0};
    Point mic3 = {0.0, 5.0};

    double t1 = 0.002345;
    double t2 = 0.002360;
    double t3 = 0.002390;

    Point source;

    if (triangulate(mic1, mic2, mic3, t1, t2, t3, &source)) {
        printf("Source position: x=%.3f y=%.3f\n", source.x, source.y);
    } else {
        printf("Failed to converge\n");
```

```
    }

    return 0;
}
```

---

5. Key Design Notes

Microphone Placement

- Non-collinear microphones
- Larger spacing → better accuracy
- Known positions with mm precision

Timing Resolution

- 1 µs timing → ~0.34 mm distance resolution
- Use hardware timers, not software polling

Noise Handling

- Average multiple events
- Use bandpass filter before trigger
- Add 4th microphone for redundancy (recommended)

---

6. Extensions (If You Want)

- Closed-form solution (Chan algorithm)

- Kalman filtering for moving sources
- 3D extension (add microphone 4)
- Fixed-point math for MCU without FPU

If you want, I can:

- Optimize this for **STM32** / **Arduino**
- Provide a **closed-form solver**
- Add **simulation & testing code**
- Extend to **3D localization**

Just