$$\text{Area} = \sum \frac{(x_{i+1} - x_i) \cdot (y_i + y_{i+1})}{2}$$

Got it — you want a **C# LINQ-to-SQL** query (targeting **PostgreSQL**) that, given an `IQueryable` of `(x, y)` points, computes the **integral area under the curve**.

Since PostgreSQL supports **window functions** and LINQ-to-SQL can translate them (via EF Core or LINQ providers like Npgsql), we can compute the **trapezoidal rule** directly in SQL without pulling all data into memory.

Complete Example

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;

public class Point
{
    public int Id { get; set; }
    public double X { get; set; }
    public double Y { get; set; }
}

public class MyDbContext : DbContext
{
    public DbSet<Point> Points { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        // Replace with your PostgreSQL connection string

optionsBuilder.UseNpgsql("Host=localhost;Database=testdb;Username=postgres;Password=yourpasswo
```

```
rd");
    }
}

class Program
{
    static void Main()
    {
        using var db = new MyDbContext();

        // Ensure data exists for demo
        if (!db.Points.Any())
        {
            db.Points.AddRange(
                new Point { X = 0, Y = 0 },
                new Point { X = 1, Y = 2 },
                new Point { X = 2, Y = 3 },
                new Point { X = 3, Y = 5 }
            );
            db.SaveChanges();
        }

        // Compute area using trapezoidal rule in SQL
        var area = db.Points
            .OrderBy(p => p.X)
            .Select(p => new
            {
```

```
                p.X,
                p.Y,
                NextX = db.Points
                    .Where(q => q.X > p.X)
                    .OrderBy(q => q.X)
                    .Select(q => (double?)q.X)
                    .FirstOrDefault(),
                NextY = db.Points
                    .Where(q => q.X > p.X)
                    .OrderBy(q => q.X)
                    .Select(q => (double?)q.Y)
                    .FirstOrDefault()
            })
            .Where(p => p.NextX != null) // skip last point
            .Select(p => ((p.NextX.Value - p.X) * (p.Y + p.NextY.Value) / 2.0))
            .Sum();

        Console.WriteLine($"Area under curve: {area}");
    }
}
```

How It Works

- **Trapezoidal Rule**:
  Area = $\sum 2(x_{i+1}-x_i)\cdot(y_i+y_{i+1})$

- **LINQ Translation**:
  - We order points by X.
  - For each point, we find the **next** point's `(X, Y)` using a subquery.
  - We compute the trapezoid area segment.
  - We sum all segments **in SQL** (no client-side iteration).
- **PostgreSQL Translation**:
  EF Core with Npgsql will generate SQL with correlated subqueries, which PostgreSQL optimizes well.

Edge Cases Handled

- **No points** → returns `null` (you can coalesce to `0`).
- **Single point** → returns `0` (no area).
- **Unsorted input** → ordering ensures correctness.
- **Duplicate X values** → handled by ordering; but you may want to `Distinct()` or aggregate first.

✅ **This runs entirely in the database** — no need to load all points into memory.
If you want **better performance** for large datasets, we can rewrite it using **PostgreSQL** `LEAD()` **window function** via EF Core raw SQL or `EF.Functions`.

Do you want me to **optimize this further** using PostgreSQL's `LEAD()` so it's a **single SQL scan** instead of correlated subqueries? That would make it much faster for big tables.