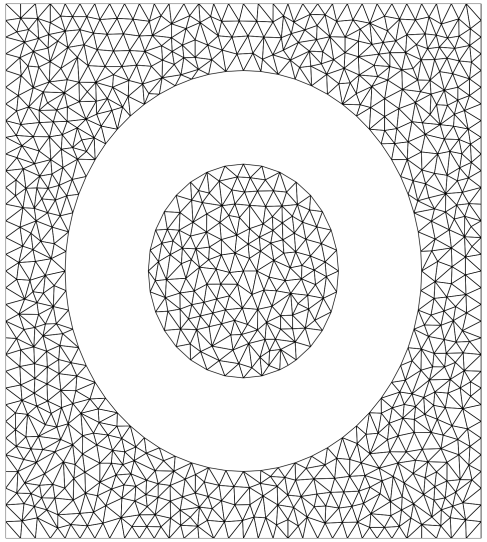
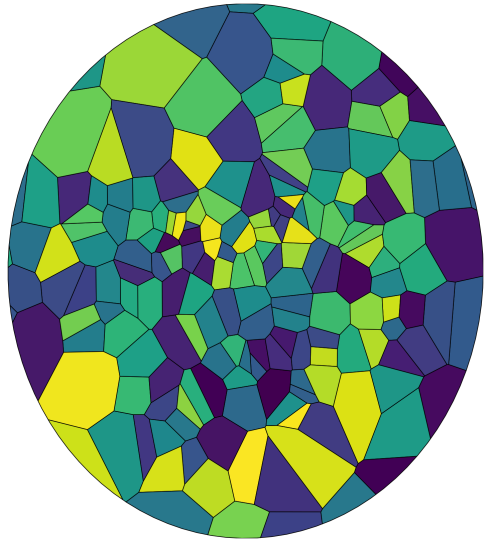
DelaunayTriangulation.jl

Two-dimensional Delaunay Triangulations and Voronoi Tessellations in Julia





Daniel VandenHeuvel

d.vandenheuvel24@imperial.ac.uk

Department of Mathematics, Imperial College London

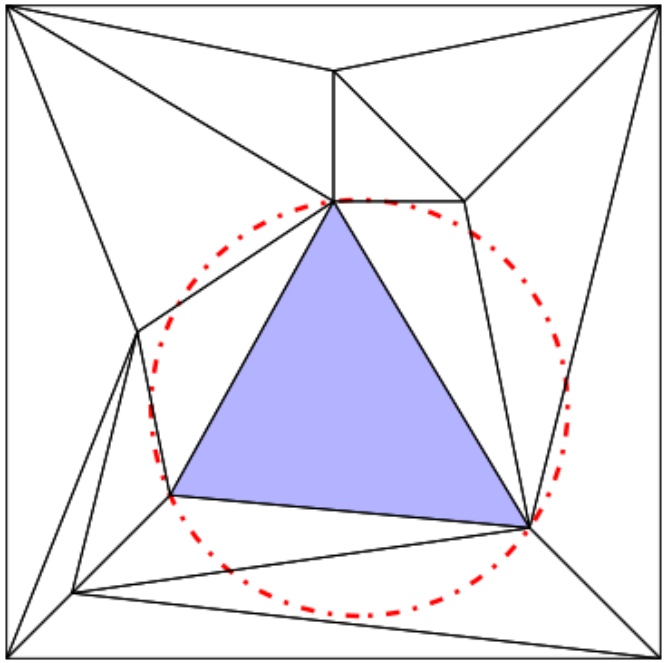
2024-10-09

Overview

- Delaunay triangulations
- Constrained Delaunay triangulations
- Mesh refinement
- Voronoi tessellations
- Interpolation and solving PDEs

Delaunay Triangulations

- A triangulation of a set of points $\mathcal{P}$ in the plane such that no triangle's circumcircle contains any other point in the set.



Delaunay Triangulations

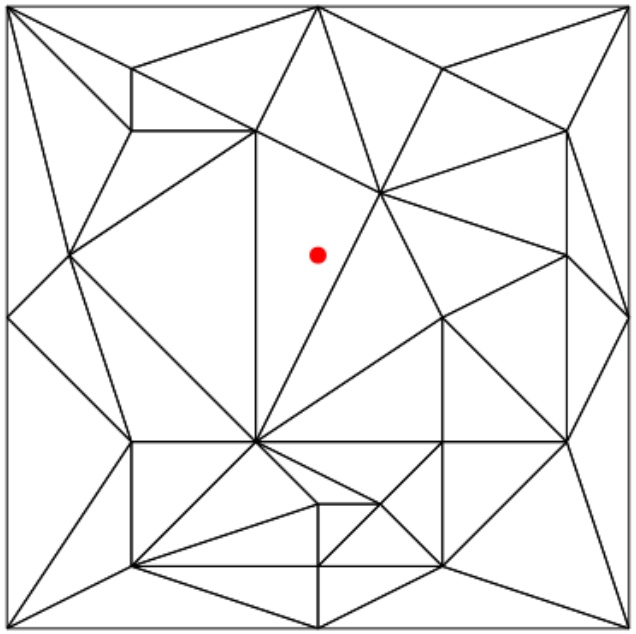
- A triangulation of a set of points $\mathcal{P}$ in the plane such that no triangle's circumcircle contains any other point in the set.
- Maximizes the minimum angle over all other triangulations of $\mathcal{P}$.

Delaunay Triangulations

- A triangulation of a set of points $\mathcal{P}$ in the plane such that no triangle's circumcircle contains any other point in the set.
- Maximizes the minimum angle over all other triangulations of $\mathcal{P}$.
- Minimises the worst-case pointwise piecewise linear interpolation error for curvature-bounded functions.

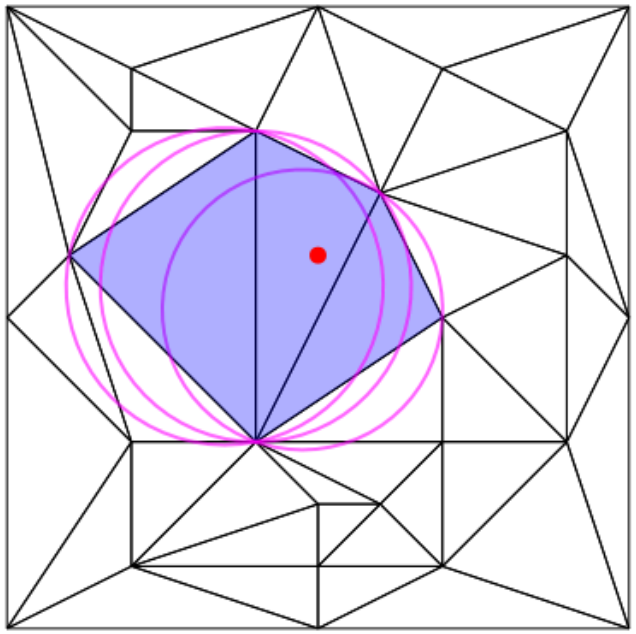
Bowyer-Watson Algorithm

- Triangulations are computed incrementally.



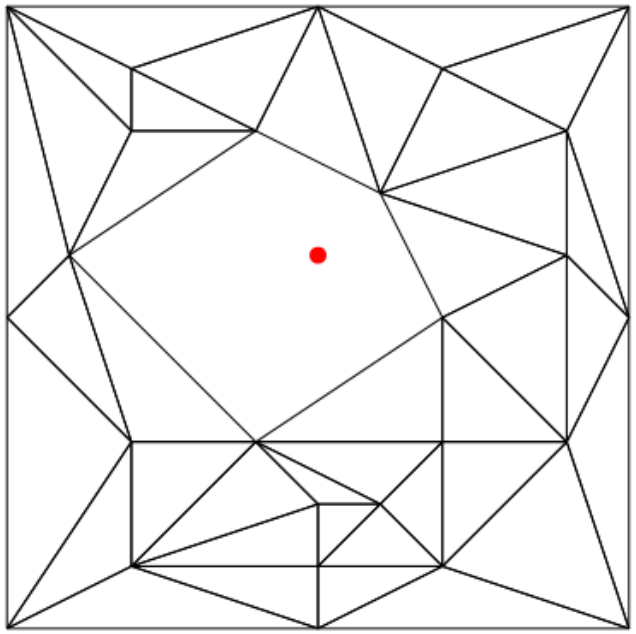
Bowyer-Watson Algorithm

- Triangulations are computed incrementally.



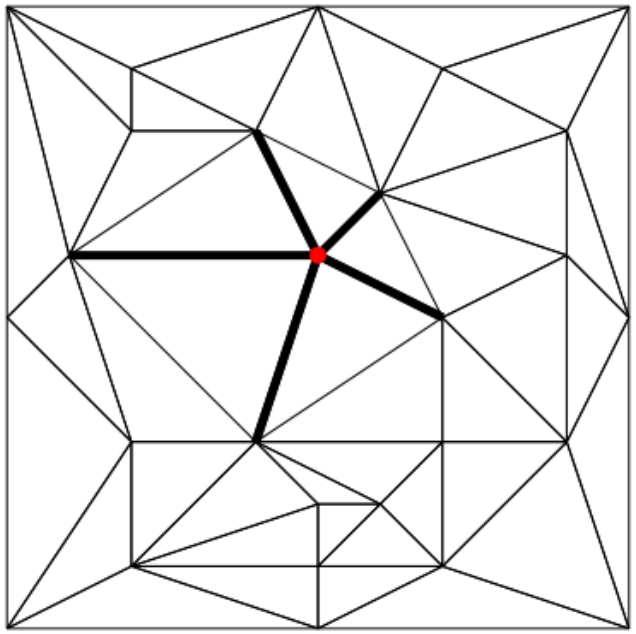
Bowyer-Watson Algorithm

- Triangulations are computed incrementally.



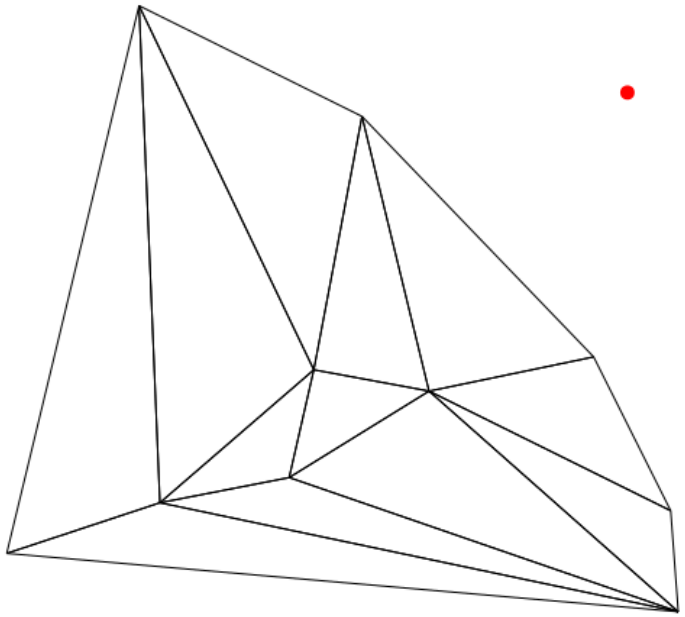
Bowyer-Watson Algorithm

- Triangulations are computed incrementally.



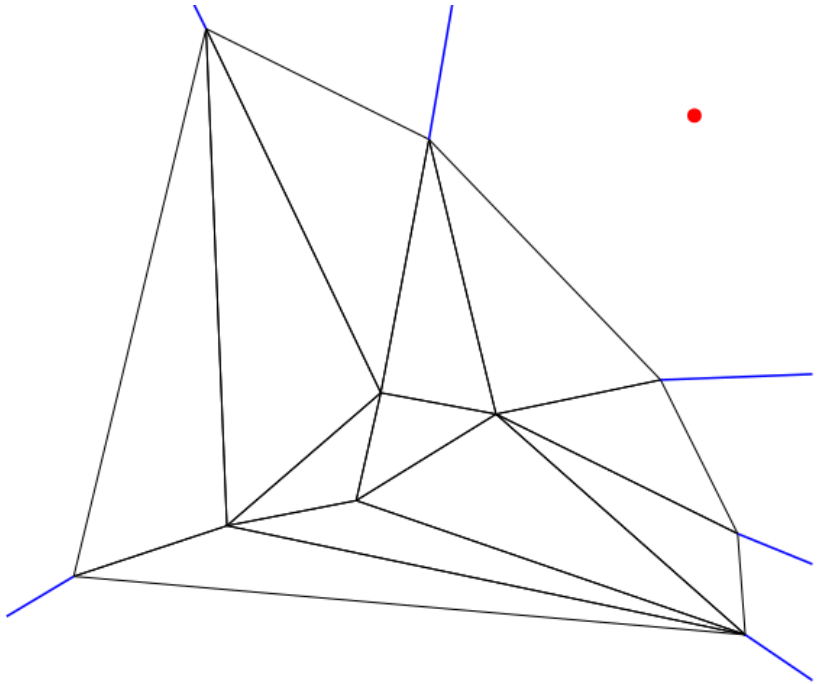
Ghost Vertices

- How to handle points outside of the triangulation?
- Ghost vertices.



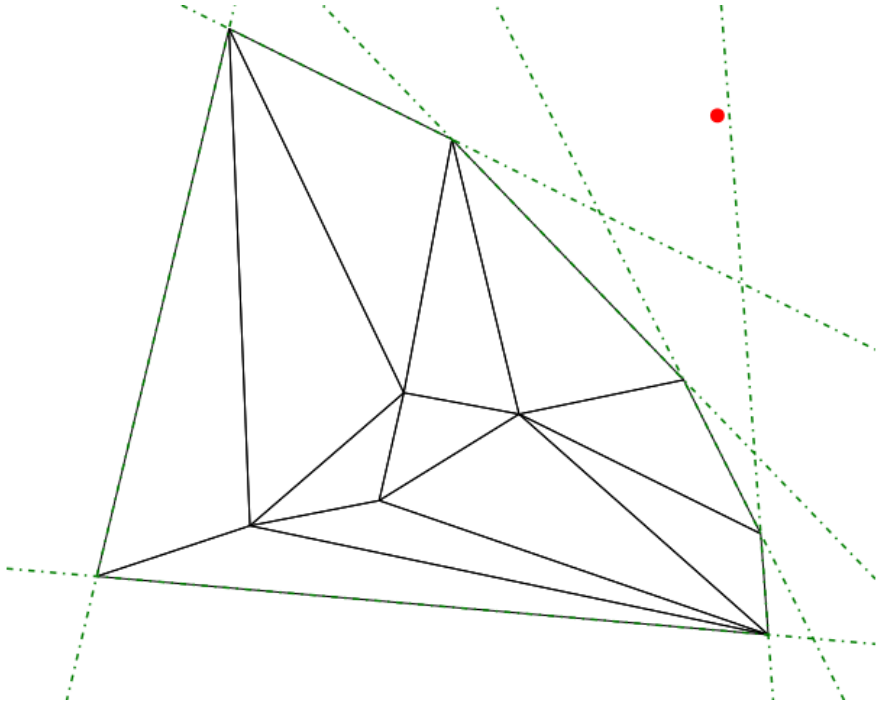
Ghost Vertices

- How to handle points outside of the triangulation?
- Ghost vertices.



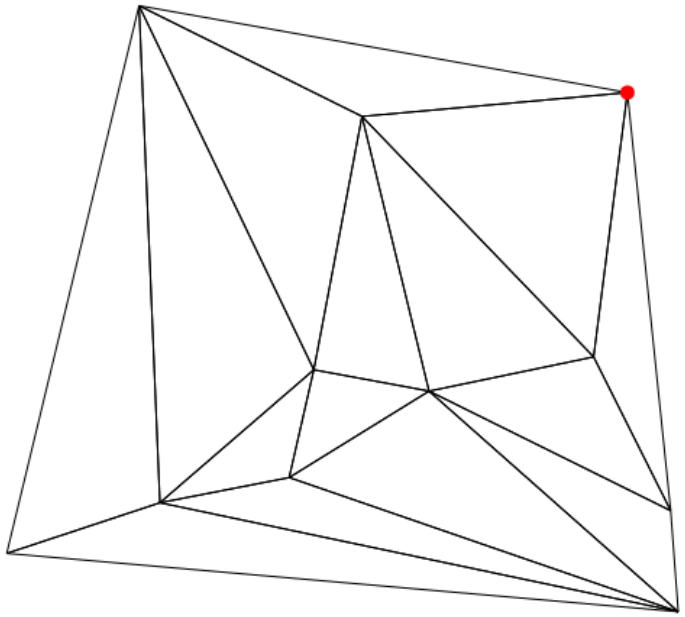
Ghost Vertices

- How to handle points outside of the triangulation?
- Ghost vertices.



Ghost Vertices

- How to handle points outside of the triangulation?
- Ghost vertices.



Using DelaunayTriangulation.jl

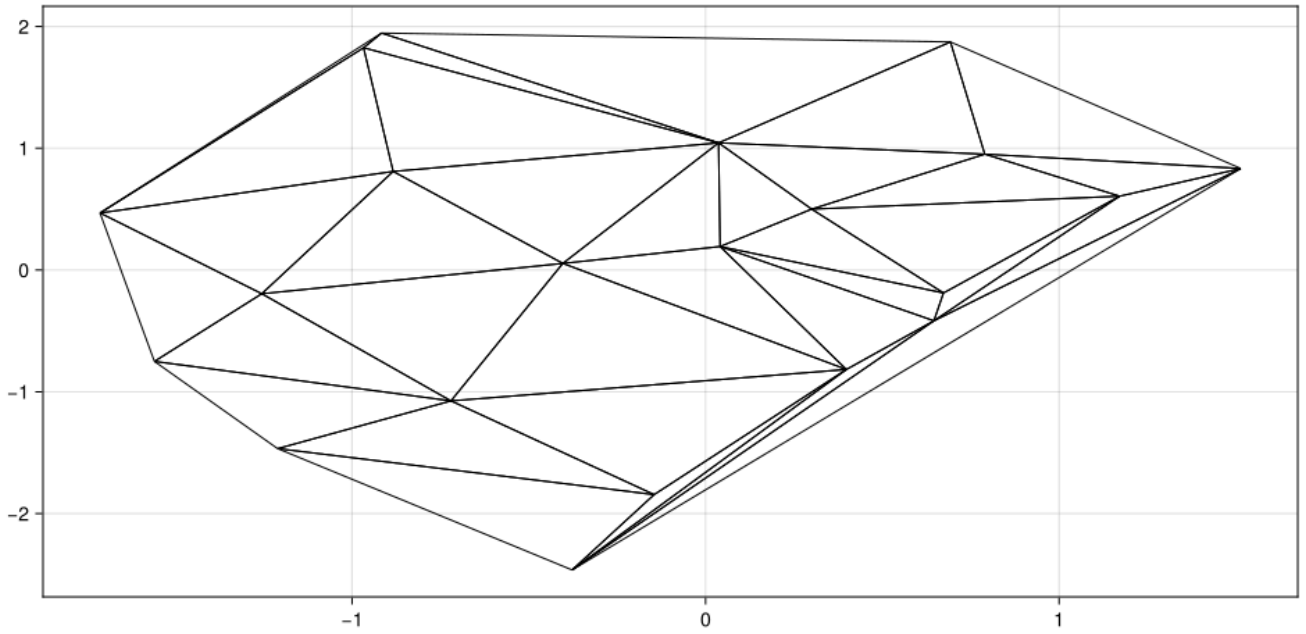
```
using DelaunayTriangulation
points = [randn(2) for _ in 1:20]
tri = triangulate(points)
```

Using DelaunayTriangulation.jl

```
using DelaunayTriangulation
points = [randn(2) for _ in 1:20]
tri = triangulate(points)
add_point!(tri, 0.3, 0.5)
```

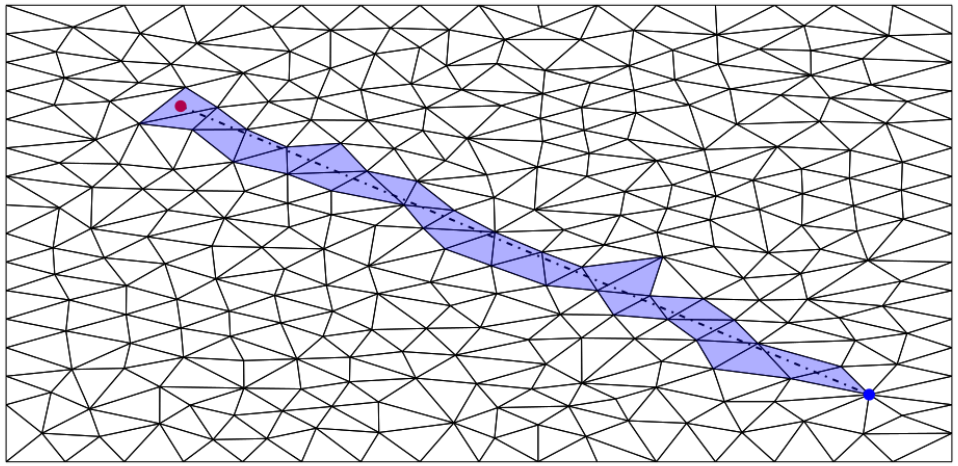
Using DelaunayTriangulation.jl

```
1 using DelaunayTriangulation
2 points = [randn(2) for _ in 1:20]
3 tri = triangulate(points)
4 add_point!(tri, 0.3, 0.5)
5 using CairoMakie
6 triplot(tri)
```



Point Location

- At each stage, we need to find the triangle T containing the inserted point q.
- Jump and march. Accelerate by sampling $\mathcal{O}(n^{1/3})$ vertices to start from.



Using DelaunayTriangulation.jl

```
1 using DelaunayTriangulation
2 points = rand(2, 500)
3 tri = triangulate(points)
4 find_triangle(tri, (0.2, 0.3))
```

```
(402, 142, 409)
```

Using DelaunayTriangulation.jl

```
1 using DelaunayTriangulation
2 points = rand(2, 500)
3 tri = triangulate(points)
4 find_triangle(tri, (0.2, 0.3))
5 find_triangle(tri, (1.2, 1.5))
```

```
(209, 199, -1)
```

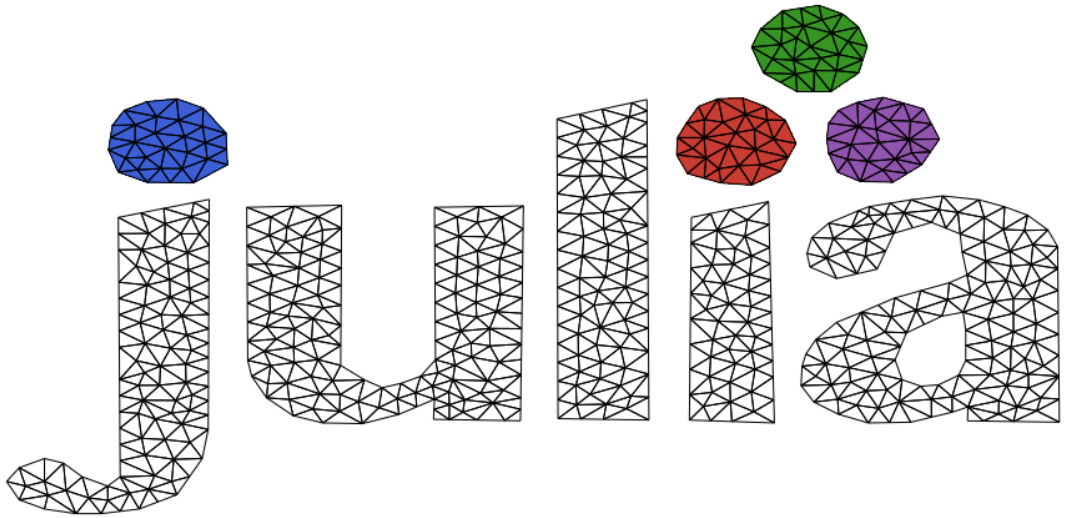
Using DelaunayTriangulation.jl

```
1 using DelaunayTriangulation
2 points = rand(2, 500)
3 tri = triangulate(points)
4 find_triangle(tri, (0.2, 0.3))
5 find_triangle(tri, (1.2, 1.5))
6 find_triangle(tri, (0.5, 0.5); k=7)
```

```
(321, 304, 186)
```

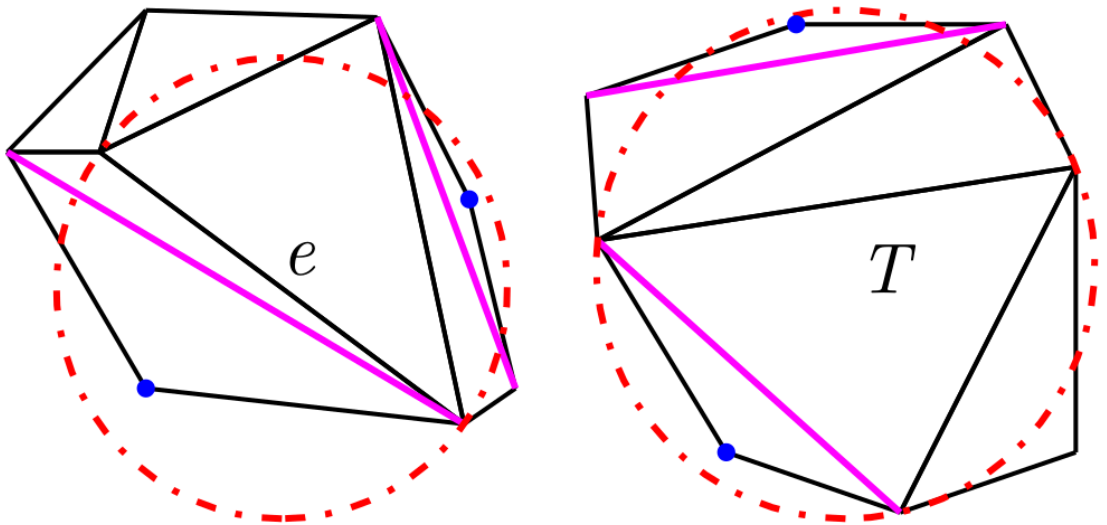
Constrained Triangulations

- Triangulation of points, segments, and boundaries.
- Useful for enforcing constraints and setting up domains for PDEs.



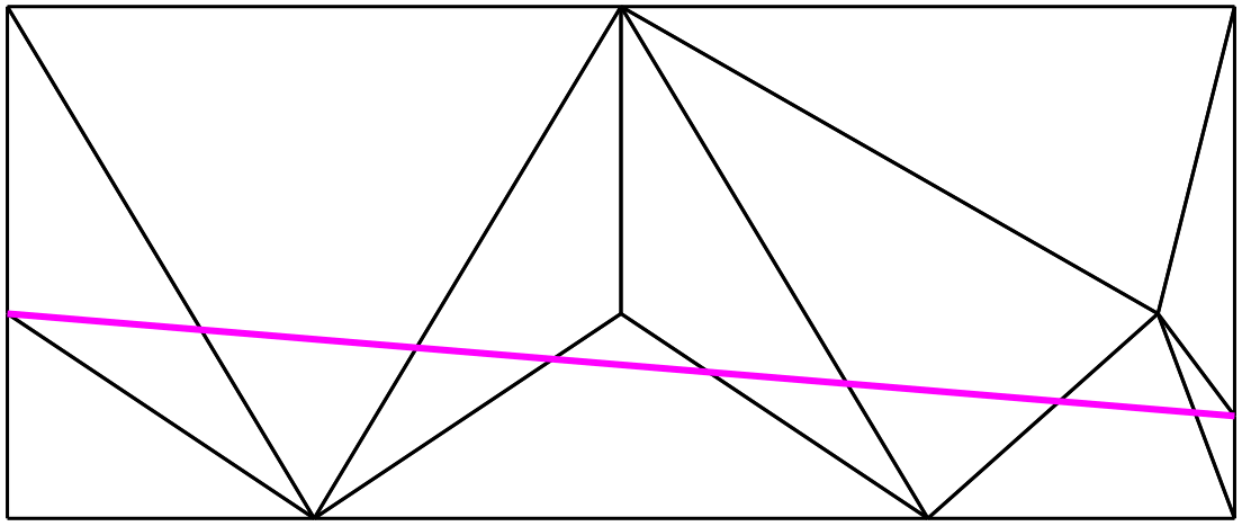
Constrained Delaunay Property

- A triangle is *constrained Delaunay* if its circumcircle contains no points in its interior *visible* from the triangle.
- The edge e and triangle T are both constrained Delaunay.



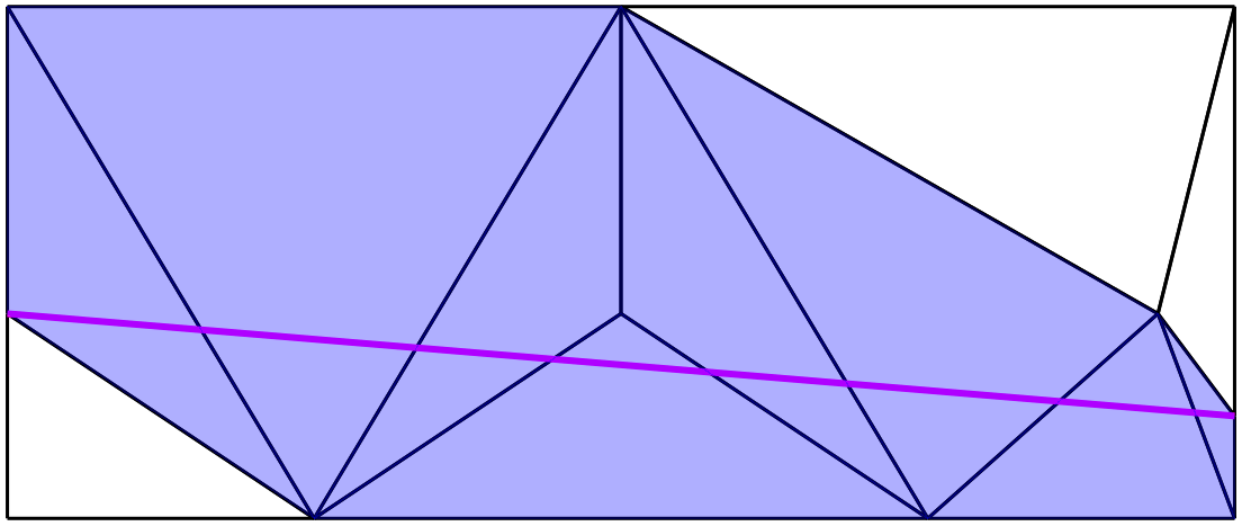
Inserting Segments

- Insert segments one at a time.
- Observation: Segments split the triangulation into two localised parts on either side.



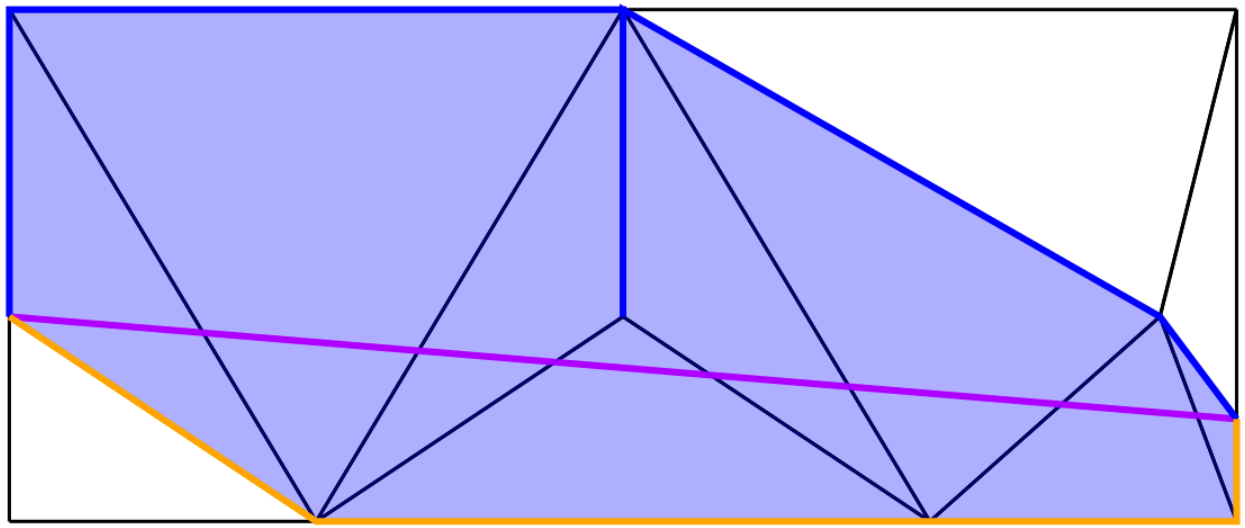
Inserting Segments

- Insert segments one at a time.
- Observation: Segments split the triangulation into two localised parts on either side.



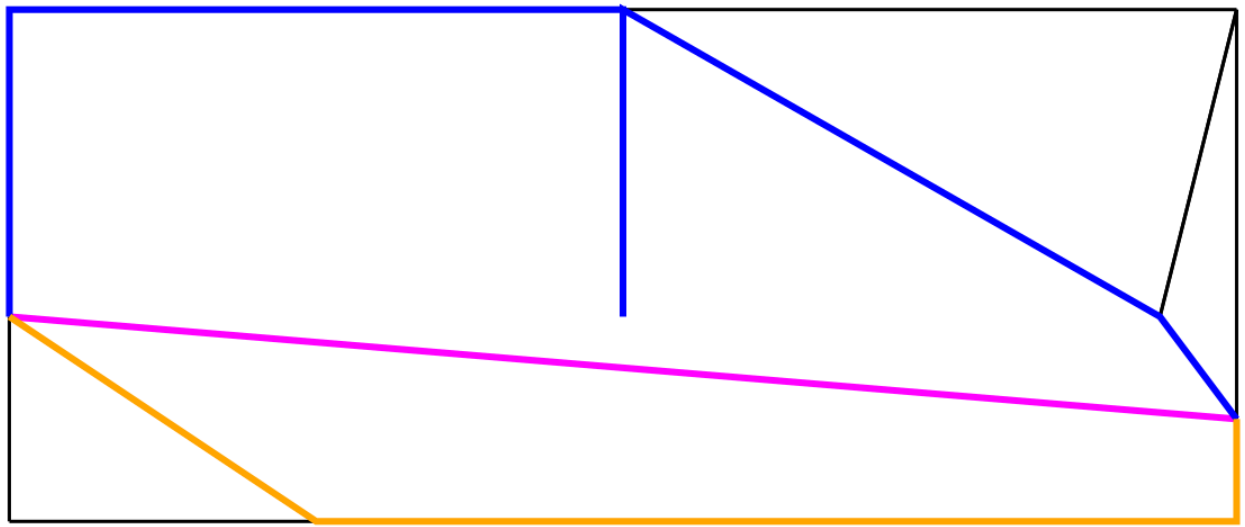
Inserting Segments

- Insert segments one at a time.
- Observation: Segments split the triangulation into two localised parts on either side.



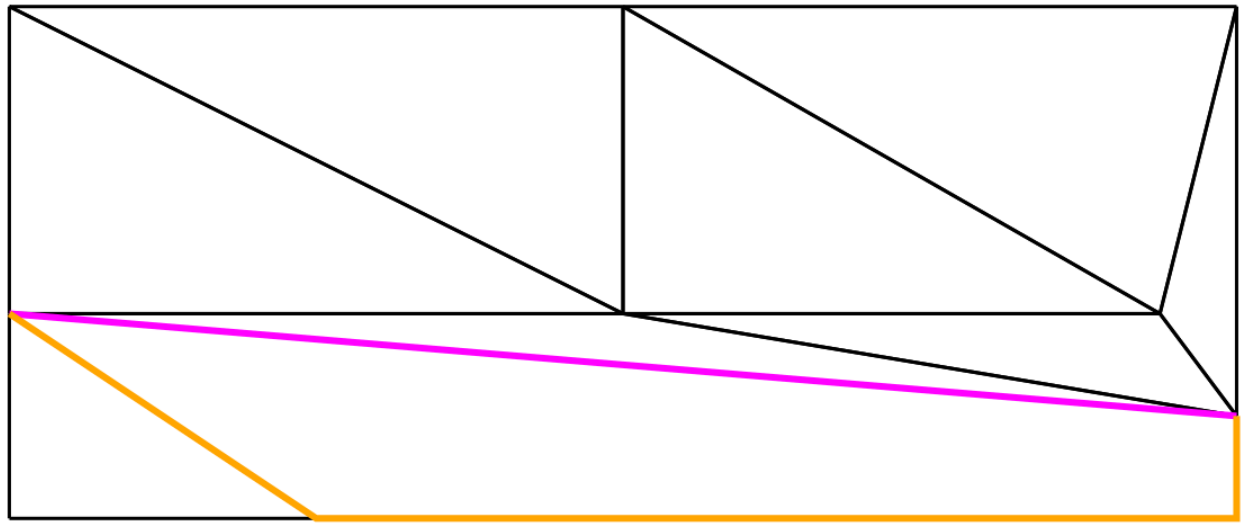
Inserting Segments

- Insert segments one at a time.
- Observation: Segments split the triangulation into two localised parts on either side.



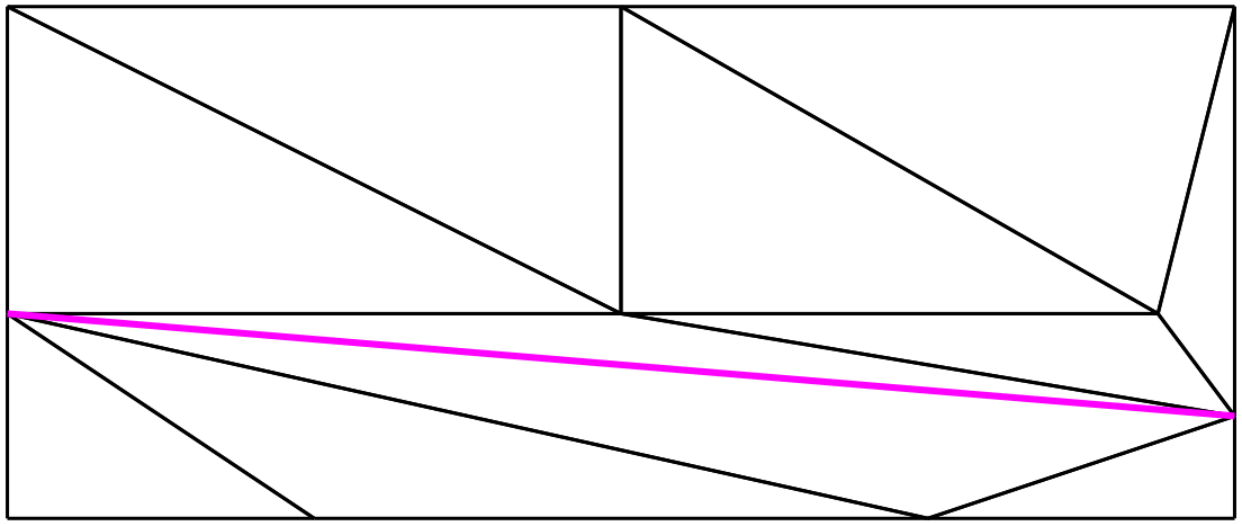
Inserting Segments

- Insert segments one at a time.
- Observation: Segments split the triangulation into two localised parts on either side.



Inserting Segments

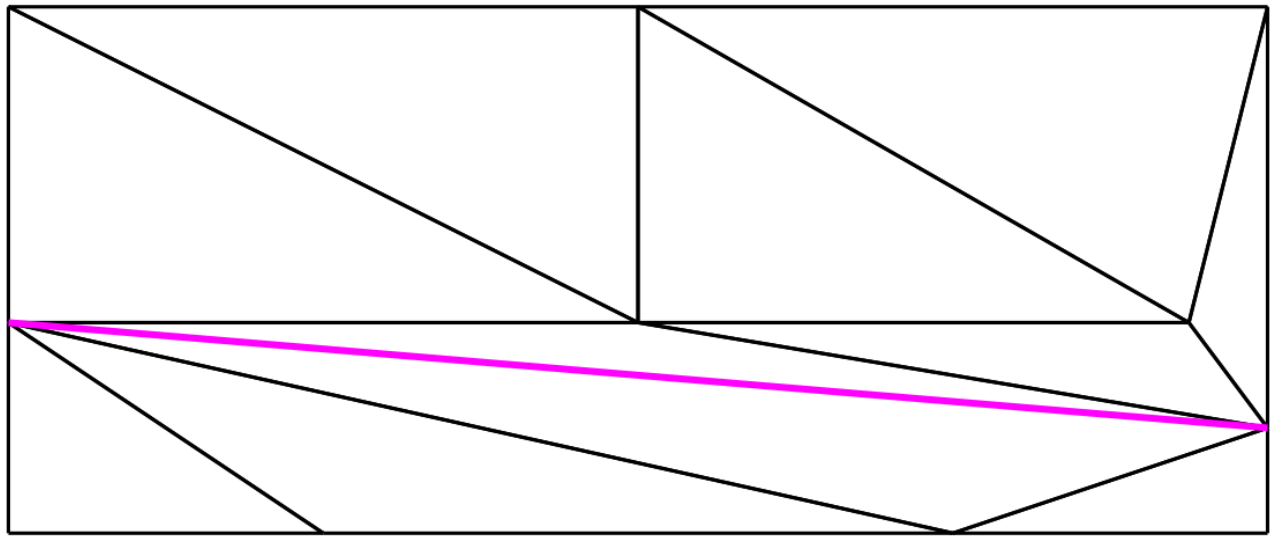
- Insert segments one at a time.
- Observation: Segments split the triangulation into two localised parts on either side.



Using DelaunayTriangulation.jl

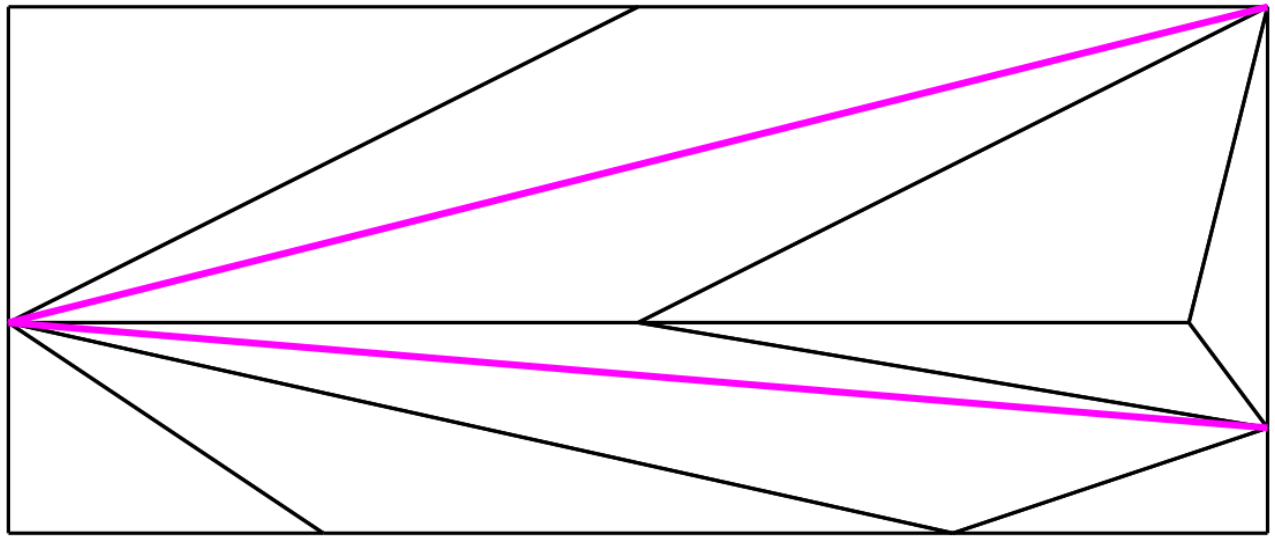
```
using DelaunayTriangulation, CairoMakie
points = [(0.0, 0.0), (0.0, 1.0), (0.0, 2.5), (2.0, 0.0),
    (6.0, 0.0), (8.0, 0.0), (8.0, 0.5), (7.5, 1.0),
    (4.0, 1.0), (4.0, 2.5), (8.0, 2.5)]
tri = triangulate(points; segments=Set([(2, 7)]))

triplot(tri)
```



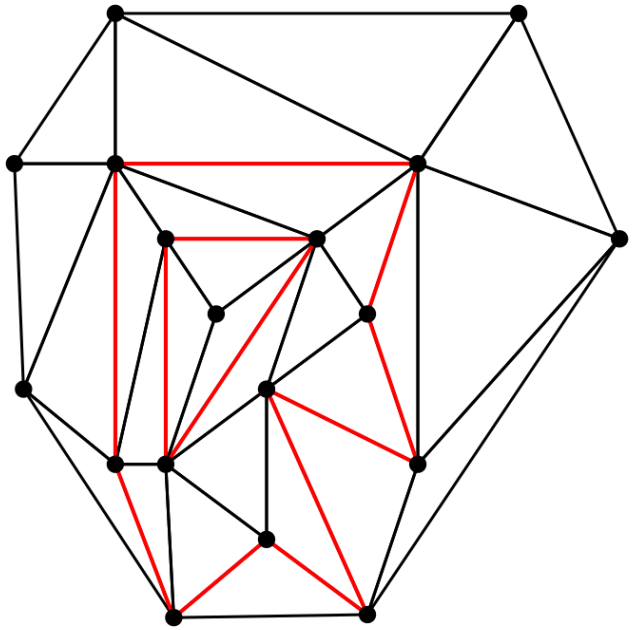
Using DelaunayTriangulation.jl

```
using DelaunayTriangulation, CairoMakie
points = [(0.0, 0.0), (0.0, 1.0), (0.0, 2.5), (2.0, 0.0),
    (6.0, 0.0), (8.0, 0.0), (8.0, 0.5), (7.5, 1.0),
    (4.0, 1.0), (4.0, 2.5), (8.0, 2.5)]
tri = triangulate(points; segments=Set([(2, 7)]))
add_segment!(tri, 2, 11)
triplot(tri)
```



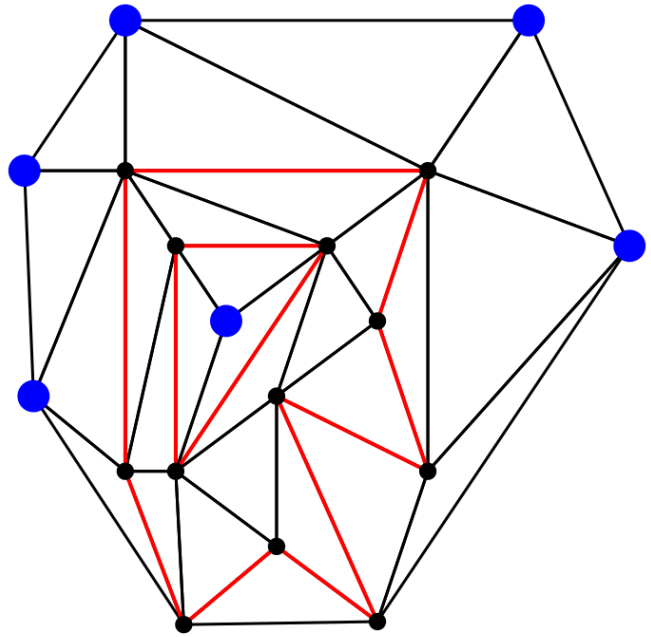
Boundaries

- Boundaries are just sequences of segments.
- Exterior vertices are deleted, identified recursively.



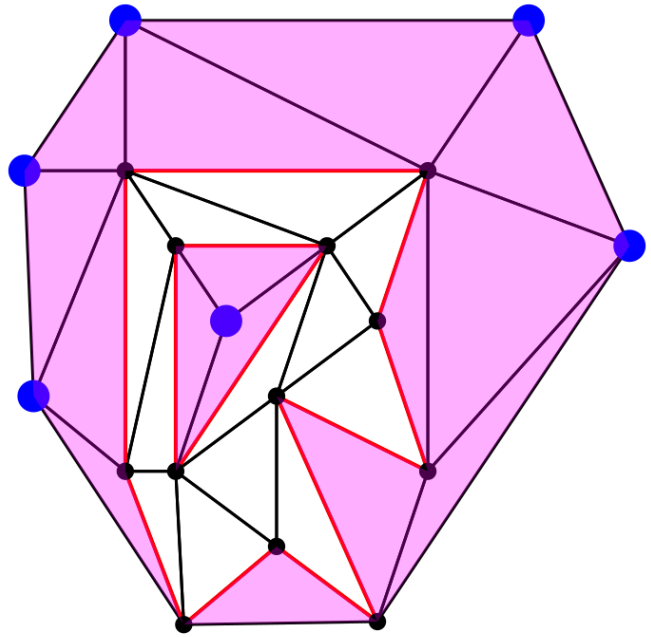
Boundaries

- Boundaries are just sequences of segments.
- Exterior vertices are deleted, identified recursively.



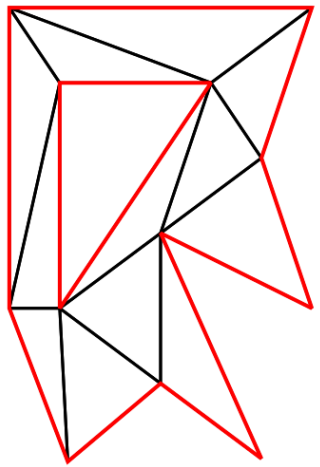
Boundaries

- Boundaries are just sequences of segments.
- Exterior vertices are deleted, identified recursively.



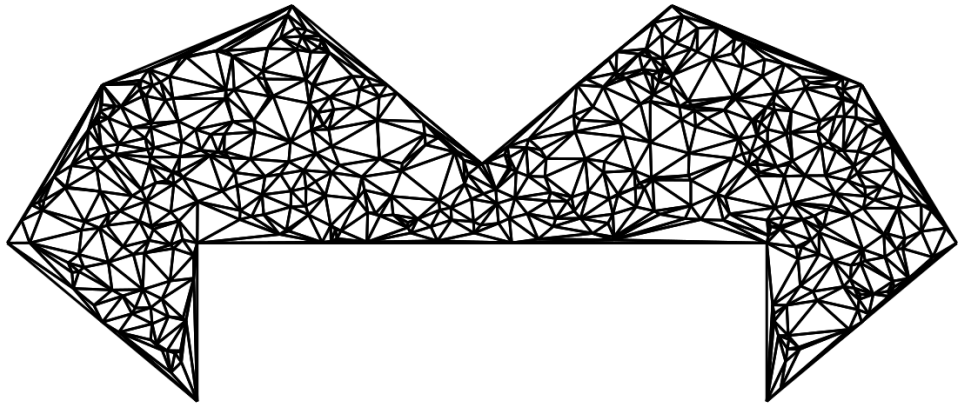
Boundaries

- Boundaries are just sequences of segments.
- Exterior vertices are deleted, identified recursively.



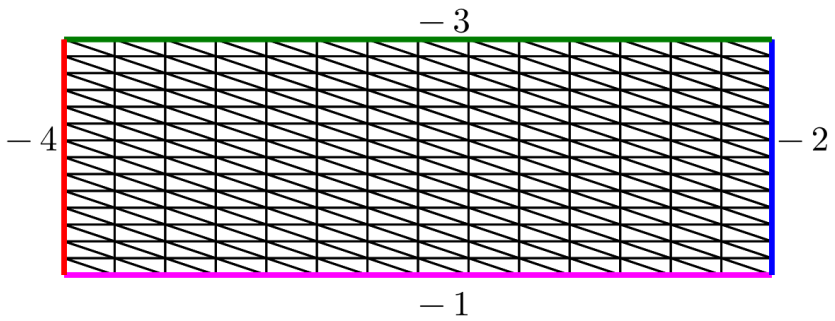
Using DelaunayTriangulation.jl

```
using DelaunayTriangulation, CairoMakie
boundary = [(0.0, 0.0), (0.1, -0.1), (0.2, -0.2), (0.2, 0.0),
    (0.8, 0.0), (0.8, -0.2), (0.9, -0.1), (1.0, 0.0),
    (0.9, 0.2), (0.7, 0.3), (0.6, 0.2), (0.5, 0.1),
    (0.4, 0.2), (0.3, 0.3), (0.1, 0.2), (0.0, 0.0)]
points = [(rand(), 0.5rand() - 0.2) for _ in 1:1000]
boundary_nodes, points = convert_boundary_points_to_indices(
    boundary; existing_points=points)
tri = triangulate(points; boundary_nodes)
triplot(tri)
```



Identifying Boundaries

- To support boundary conditions, we need a way to assign IDs to boundaries.
- Define boundaries as sequences of individual sections, with each section having its own ghost vertex.
- Leveraged by FiniteVolumeMethod.jl.



Using DelaunayTriangulation.jl

```
using DelaunayTriangulation, CairoMakie
section_1 = [(0.0, 0.0), (0.1, -0.1)]
section_2 = [(0.1, -0.1), (0.2, -0.2), (0.2, 0.0),
    (0.8, 0.0), (0.8, -0.2)]
section_3 = [(0.8, -0.2), (0.9, -0.1), (1.0, 0.0)]
section_4 = [(1.0, 0.0), (0.9, 0.2), (0.7, 0.3), (0.6, 0.2), (0.5, 0.1),
    (0.4, 0.2), (0.3, 0.3), (0.1, 0.2), (0.0, 0.0)]
boundary = [section_1, section_2, section_3, section_4]
points = [(rand(), 0.5rand() - 0.2) for _ in 1:1000]
boundary_nodes, points = convert_boundary_points_to_indices(
    boundary; existing_points=points)
tri = triangulate(points; boundary_nodes)
```

Using DelaunayTriangulation.jl

- You can access features of the boundary easily.

```
1  get_boundary_nodes(tri)
```

```
4-element Vector{Vector{Int64}}:
 [1001, 1002]
 [1002, 1003, 1004, 1005, 1006]
 [1006, 1007, 1008]
 [1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1001]
```

Using DelaunayTriangulation.jl

- You can access features of the boundary easily.

```
1  get_boundary_nodes(tri)
```

```
4-element Vector{Vector{Int64}}:
 [1001, 1002]
 [1002, 1003, 1004, 1005, 1006]
 [1006, 1007, 1008]
 [1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1001]
```

```
1  get_adjacent2vertex(tri, -2)
```

```
Set{Tuple{Int64, Int64}} with 4 elements:
  (1004, 1003)
  (1006, 1005)
  (1003, 1002)
  (1005, 1004)
```

Using DelaunayTriangulation.jl

- You can access features of the boundary easily.

```
1  get_boundary_nodes(tri)
```

```
4-element Vector{Vector{Int64}}:
 [1001, 1002]
 [1002, 1003, 1004, 1005, 1006]
 [1006, 1007, 1008]
 [1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1001]
```

```
1  get_adjacent2vertex(tri, -2)
```

```
Set{Tuple{Int64, Int64}} with 4 elements:
  (1004, 1003)
  (1006, 1005)
  (1003, 1002)
  (1005, 1004)
```

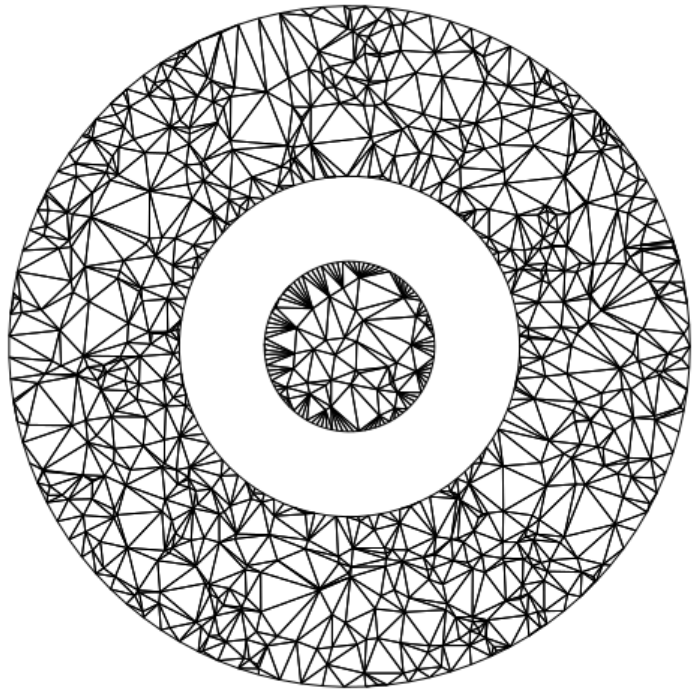
```
1  get_neighbours(tri, -2)
```

```
Set{Int64} with 5 elements:
  1005
  1002
  1006
  1003
```

Using DelaunayTriangulation.jl

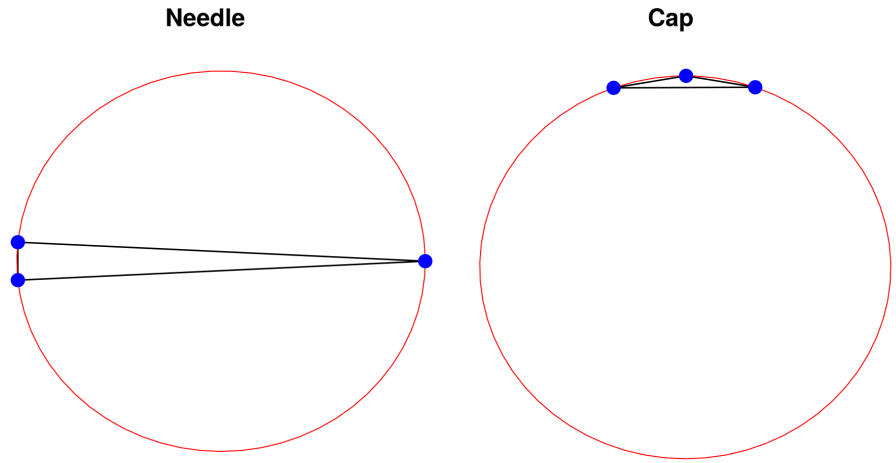
- Disjoint domains and holes follow the same representation.

```
using DelaunayTriangulation
using CairoMakie
θ = LinRange(0, 2pi, 100);
θ = [θ[1:99]; θ[1]];
θr = reverse(θ)
otrx, otry = 10cos.(θ), 10sin.(θ)
innx, inny = 5cos.(θr), 5sin.(θr)
inn2x, inn2y = 2.5cos.(θ), 2.5sin.(θ)
x = [[otrx], [innx], [inn2x]]
y = [[otry], [inny], [inn2y]]
points = [(20rand() - 10, 20rand() - 10)
          for _ in 1:1000]
boundary_nodes, points =
    convert_boundary_points_to_indices(
        x, y; existing_points=points)
tri = triangulate(points;
    boundary_nodes)
triplot(tri)
```



Mesh Refinement

- Solutions using meshes are highly dependent on the quality of the triangles.
- Poor quality triangles can cause interpolation errors, gradient errors, and ill-conditioning.



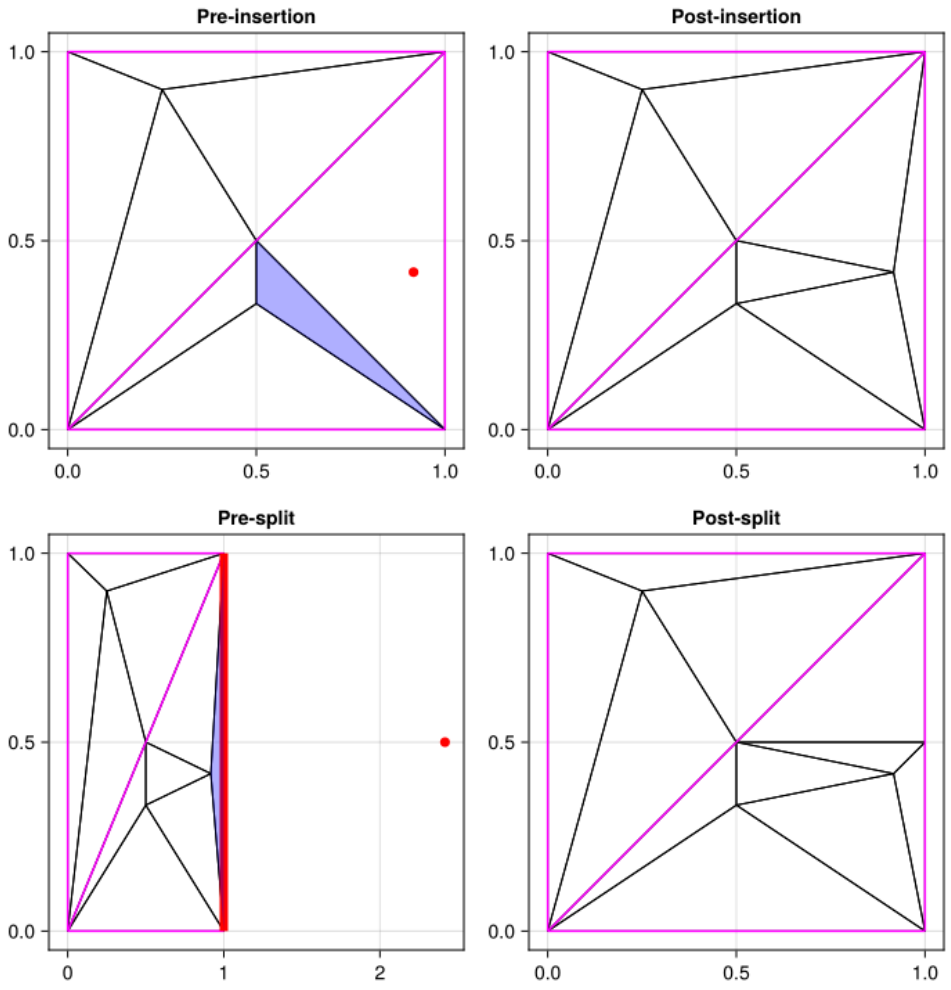
Assessing Element Quality

- Assess triangles using the *radius-edge ratio* $\rho = R/\ell_{\min}$.
- $1/\sqrt{3} \leq \rho < \infty$, with $1/\sqrt{3}$ attained for equilateral triangles.
- Related to the smallest angle by

$$\rho = \frac{1}{2 \sin \theta_{\min}}.$$

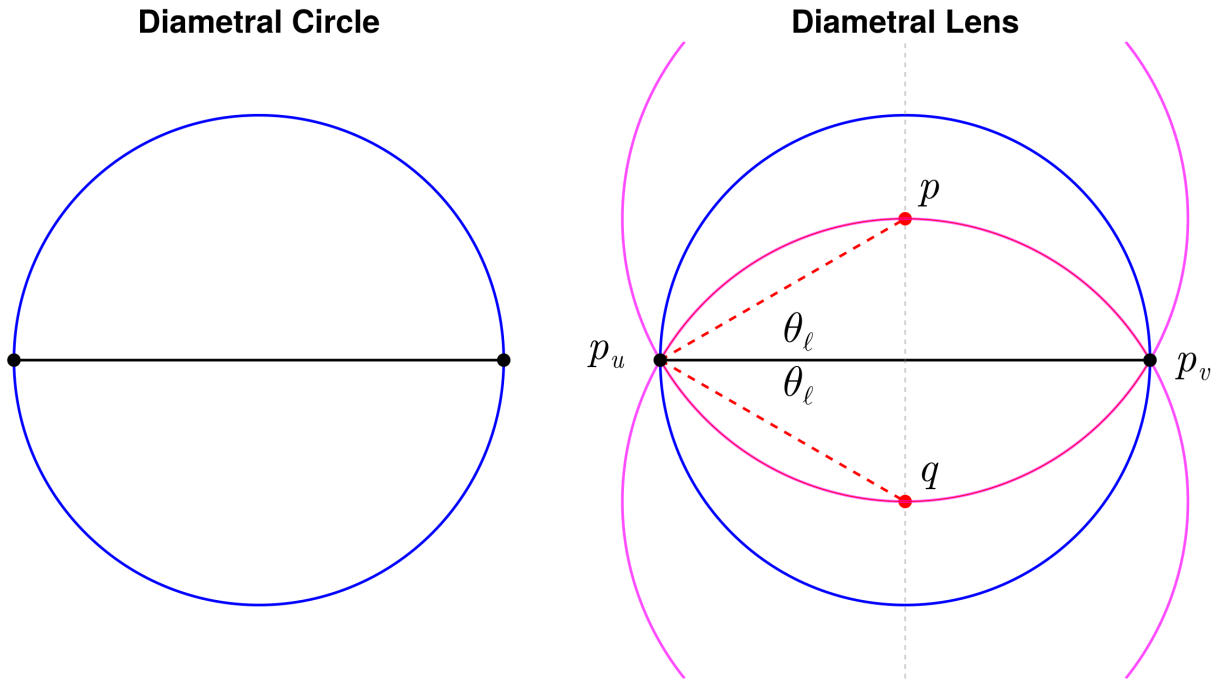
Thus, controlling ρ controls the smallest angle.

Triangle and Segment Splitting



Mesh Encroachment

- Mesh refinement inserts points into the triangulation.
- One issue with insertion is *encroachment*.



Ruppert's Algorithm

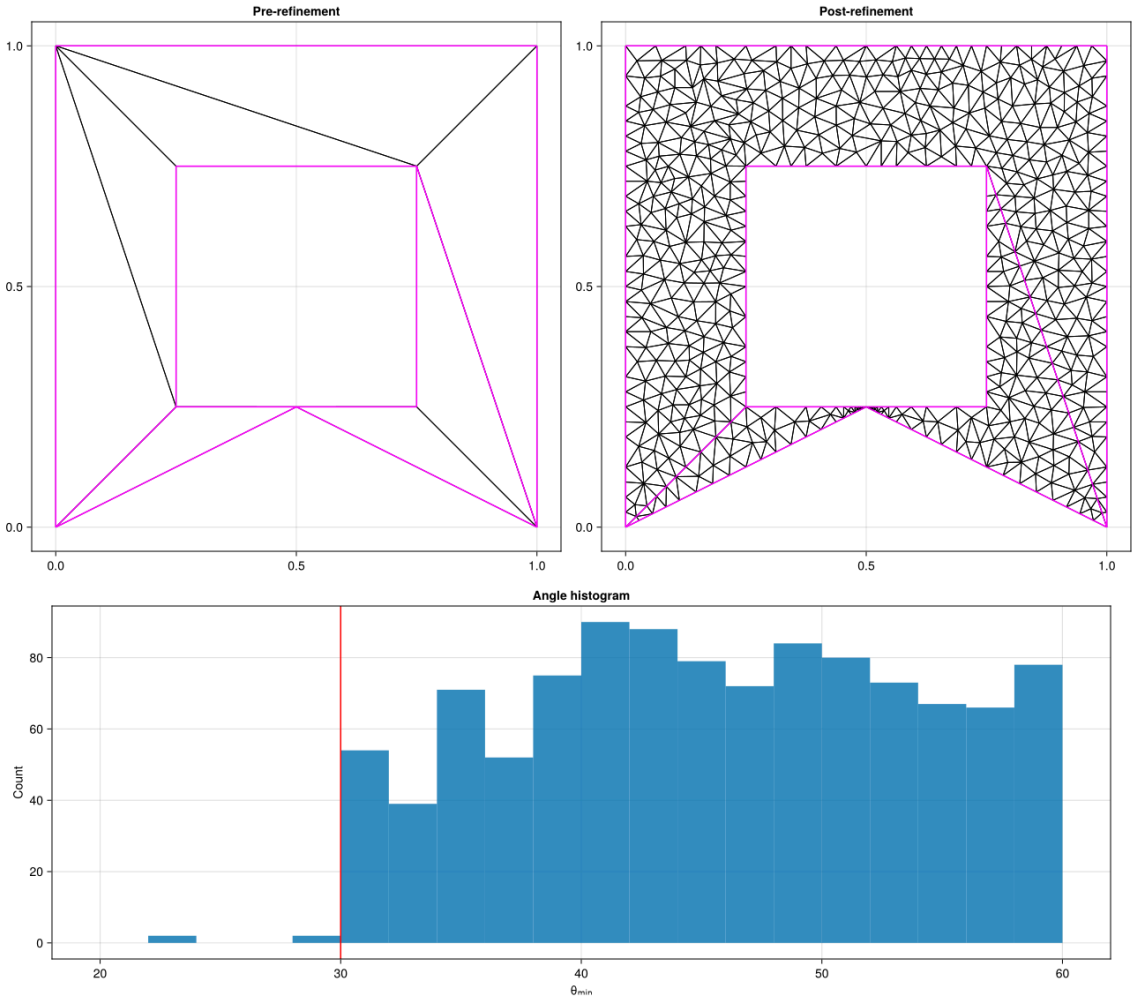
```
function refine(tri)
    split_all_encroached_segments!(tri)
    while has_bad_triangles(tri)
        T = get_bad_triangle(tri)
        insert_circumcenter!(tri, T)
        if circumcenter_encroaches(tri)
            undo_insertion!(tri, T)
            split_all_encroached_segments!(tri)
        else
            for V in new_triangles(tri)
                ρ, A = radius_edge_ratio(V), area(V)
                (ρ > ρmax || A > Amax) && mark_bad!(tri, V)
            end
        end
    end
end
```

- Typically converges for $\theta_{\min} < 33.9^\circ$.
- Extra care needed for small angles.

Using DelaunayTriangulation.jl

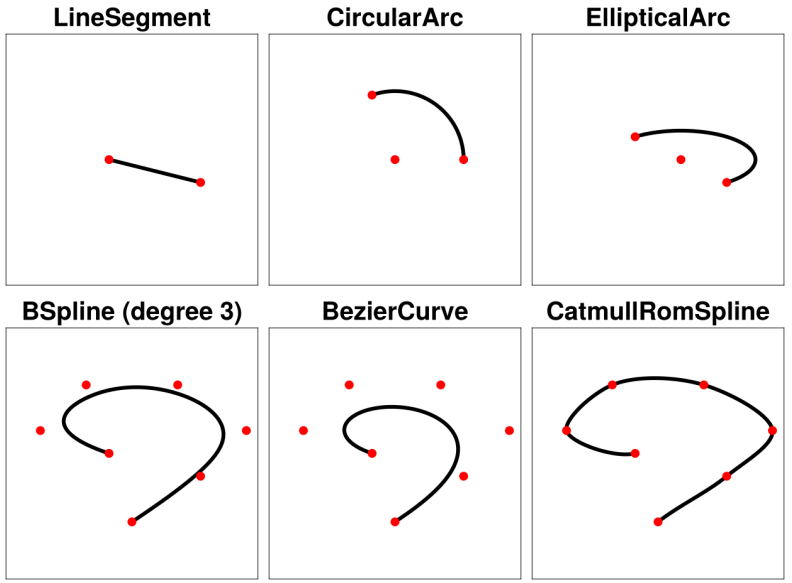
```
using DelaunayTriangulation, CairoMakie
outer = [(0.0, 0.0), (0.5, 0.2499), (1.0, 0.0), (1.0, 1.0), (0.0, 1.0), (0.0, 0.0)]
inner = [(0.25, 0.25), (0.25, 0.75), (0.75, 0.75), (0.75, 0.25), (0.25, 0.25)]
boundary = [[outer], [inner]]
boundary_nodes, points = convert_boundary_points_to_indices(boundary)
segments = Set([(1, 5), (8, 3)])
tri = triangulate(points; boundary_nodes, segments)
refine!(tri; max_area=0.001)
triplot(tri)
```

Using DelaunayTriangulation.jl



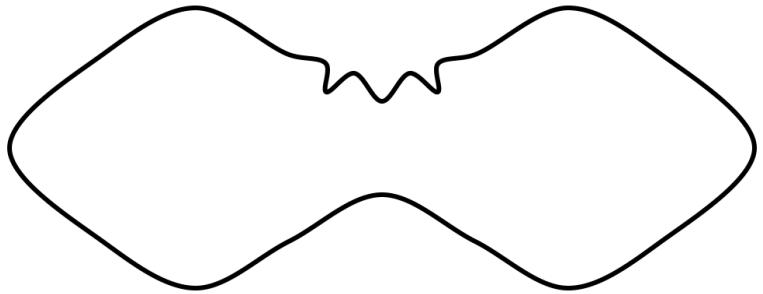
Curve-Bounded Domains

- Generalise triangulations to allow for *parametric curves*.
- Six types of parametric curves are supported, but the interface can be extended easily to new curves.



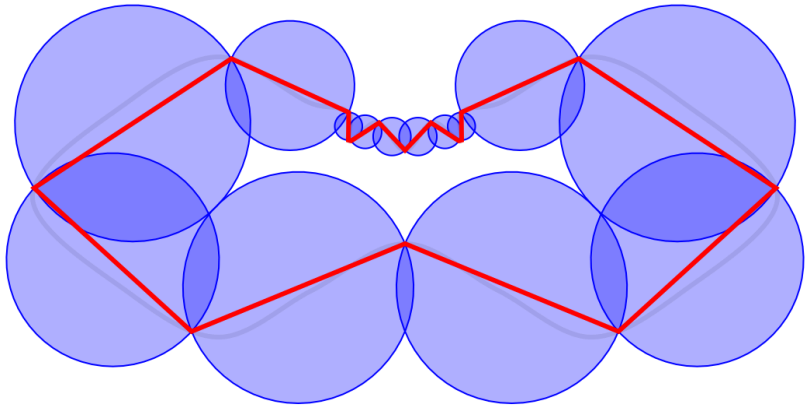
Boundary Enrichment

- Equally spaced discretisations can lead to excess triangles. Use curvature.
- Discretise until all edges are *constrained Delaunay* and the *total variation* $\int_C |\kappa(s)| \, \mathrm{d}s < \pi/2$ over each subcurve.



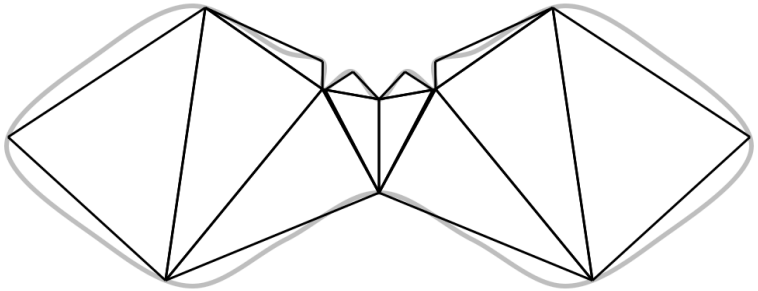
Boundary Enrichment

- Equally spaced discretisations can lead to excess triangles. Use curvature.
- Discretise until all edges are *constrained Delaunay* and the *total variation* $\int_C |\kappa(s)| \, \mathrm{d}s < \pi/2$ over each subcurve.



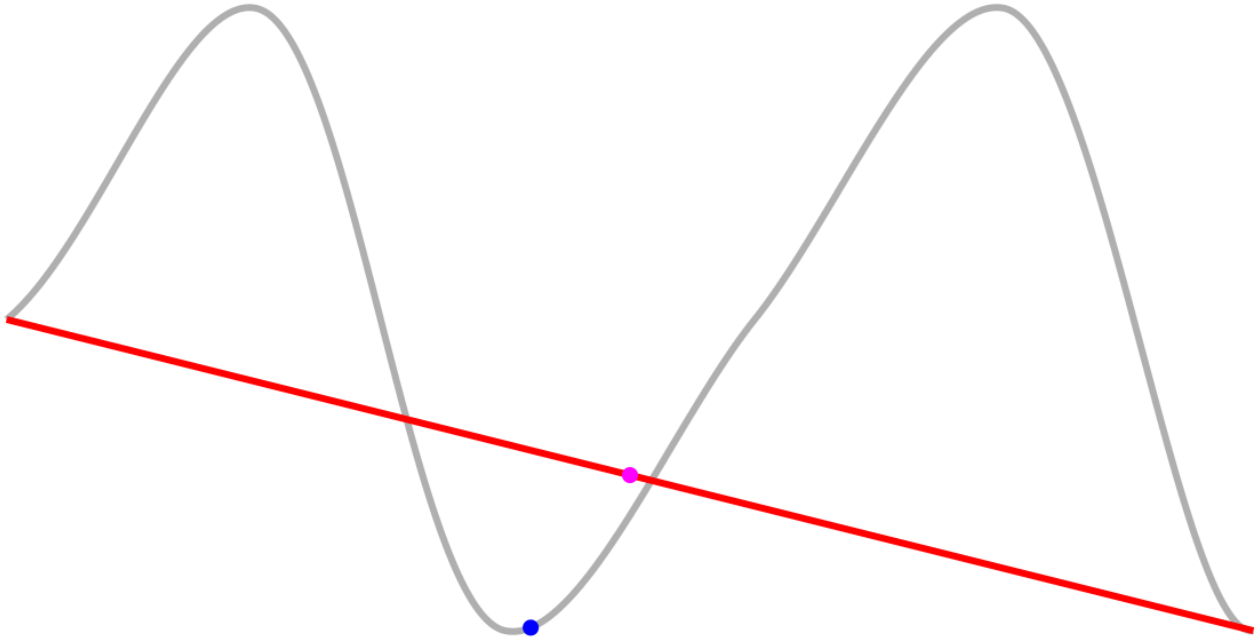
Boundary Enrichment

- Equally spaced discretisations can lead to excess triangles. Use curvature.
- Discretise until all edges are *constrained Delaunay* and the *total variation* $\int_C |\kappa(s)| \, \mathrm{d}s < \pi/2$ over each subcurve.



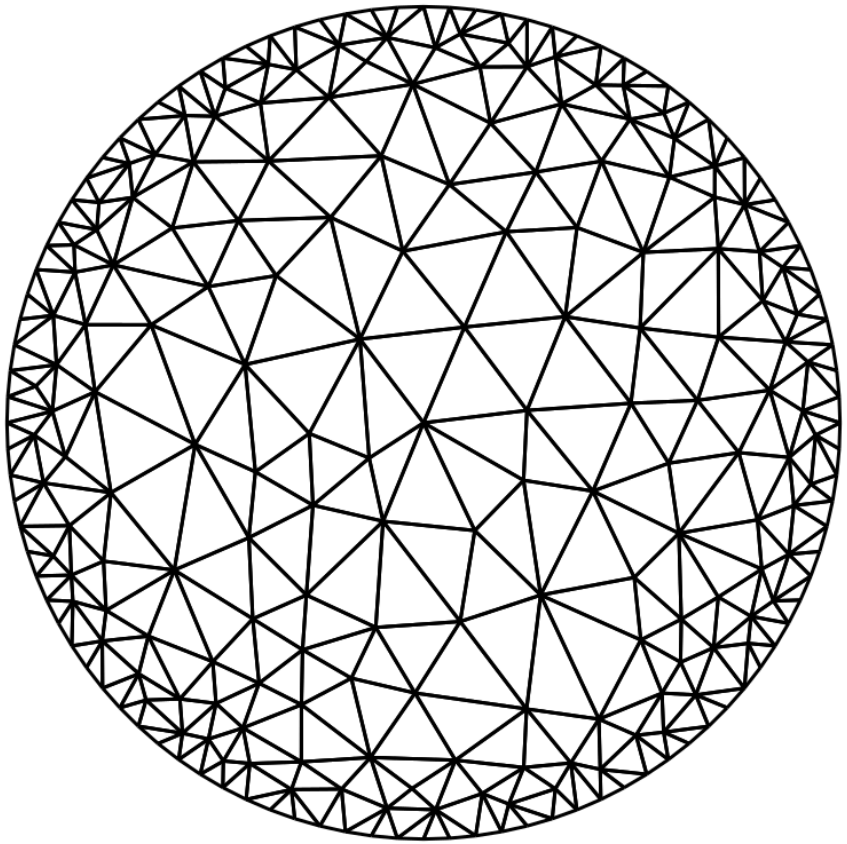
Curve-Bounded Refinement

- Works just as in the standard case, except encroached edges are split at the *equivariation point* instead of their midpoint.

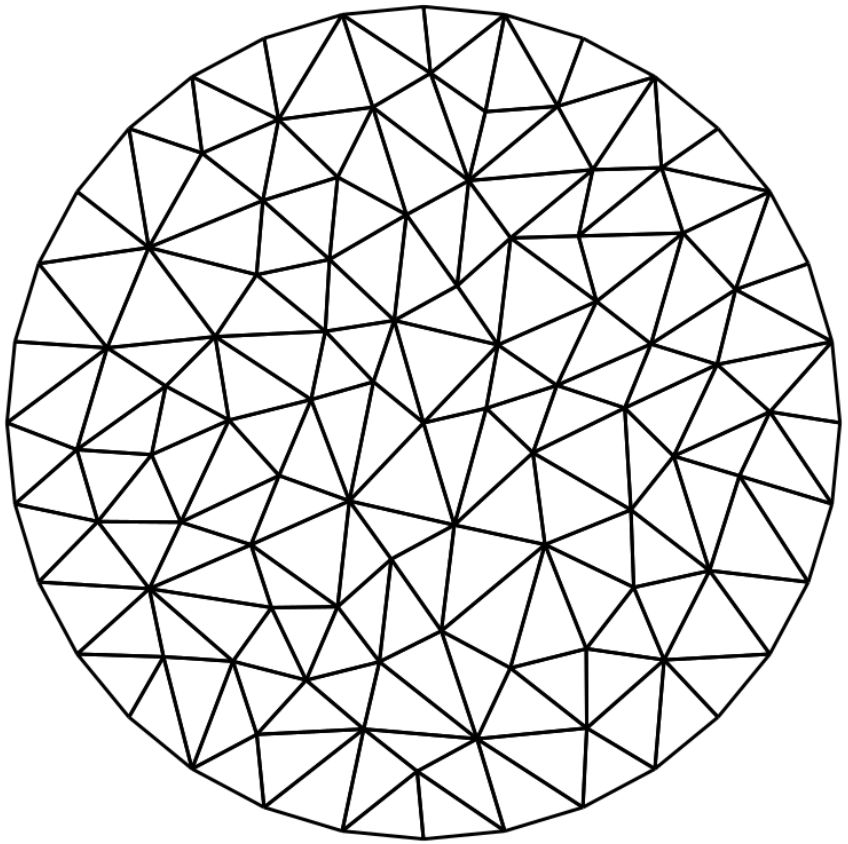


Comparison with Discretisation

Manual

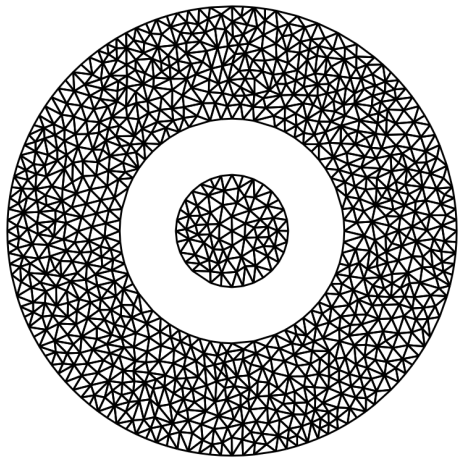


CircularArc



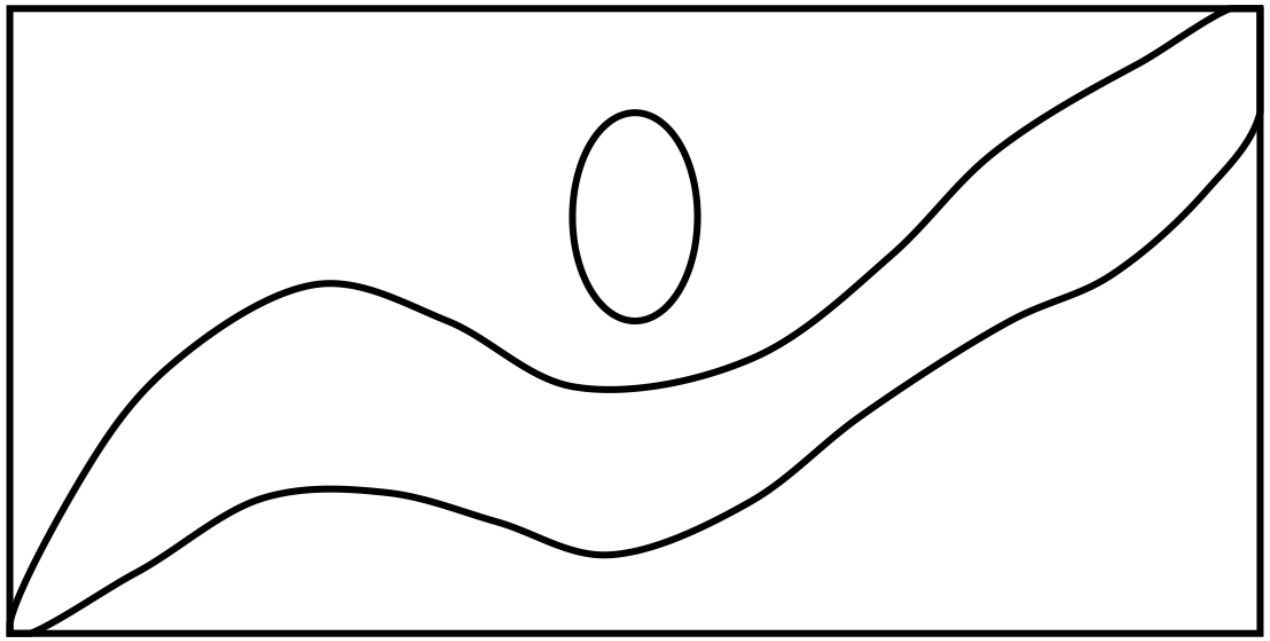
Using DelaunayTriangulation.jl

```
Random.seed!(123)
using DelaunayTriangulation, CairoMakie
r₁, r₂, r₃ = 10.0, 5.0, 2.5
c₁ = CircularArc((r₁, 0.0), (r₁, 0.0), (0.0, 0.0))
c₂ = CircularArc((r₂, 0.0), (r₂, 0.0), (0.0, 0.0), positive=false)
c₃ = CircularArc((r₃, 0.0), (r₃, 0.0), (0.0, 0.0))
tri = triangulate(NTuple{2,Float64}[]; boundary_nodes=[[[c₁]], [[c₂]], [[c₃]]])
refine!(tri; max_area=0.001get_area(tri))
triplot(tri)
```



Heterogenous Constraints

- May want to apply different refinement criteria to different regions.

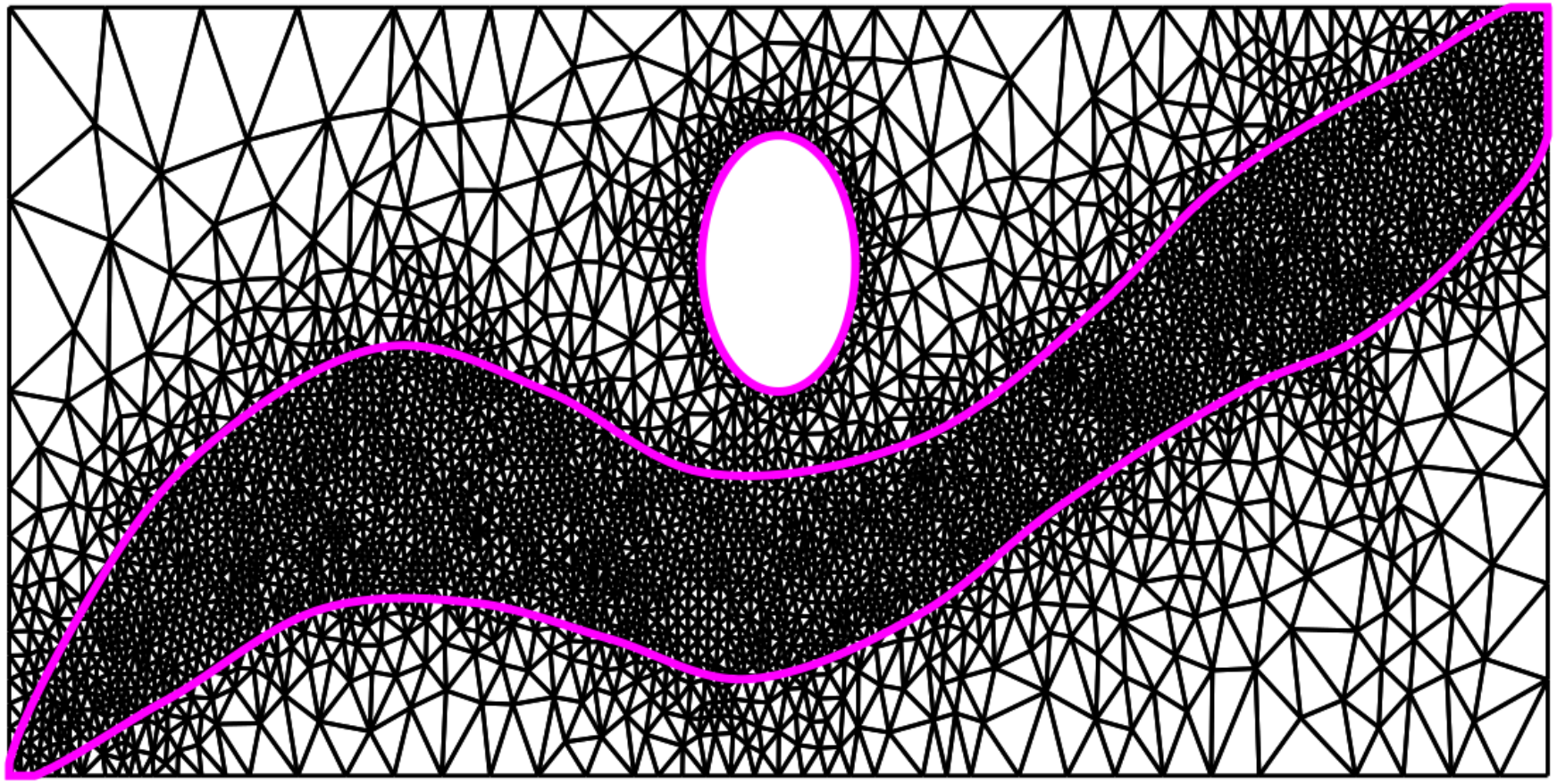


Heterogenous Constraints

- Use `custom_constraint`.

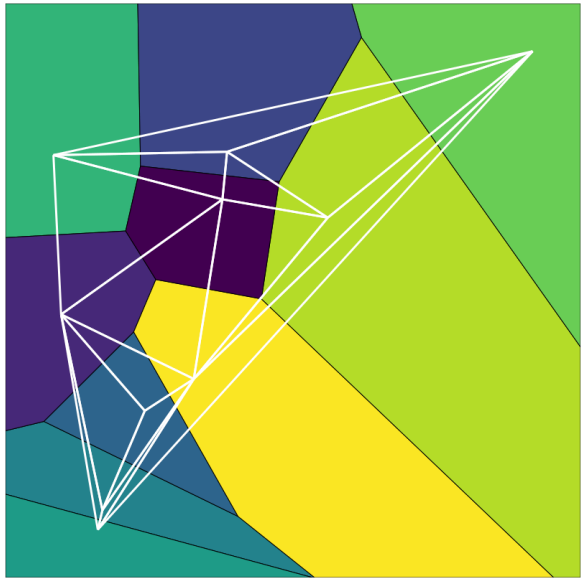
```
domain = [A, B, C, D, A]
river = [A, N, O, P, Q, R, S, T, U, V, W, Z, C, M, L, K, J, I, H, G, F, E, A]
A1, B1 = (50.0, 20.0), (50.0, 25.0)
cir = CircularArc(B1, B1, A1, positive=false)
boundary_nodes, points = convert_boundary_points_to_indices(domain)
river_spl = CatmullRomSpline(river)
tri = triangulate(points; boundary_nodes=[[[cir]], [boundary_nodes]])
Ar = 100 * 30
custom_constraint = (tri, T) -> begin
    p, q, r = get_point(tri, triangle_vertices(T)...)
    c = (p .+ q .+ r) ./ 3
    TA = triangle_area(p, q, r)
    is_left(point_position_relative_to_curve(river_spl, c)) && return TA > 1e-4Ar
    abs(dist(c, A1) - dist(A1, B1)) / dist(A1, B1) < 1e-1 && return TA > 1e-4Ar
    return false
end
refine!(tri; custom_constraint=custom_constraint, max_area=1e-1Ar)
triplot(tri)
```

Heterogenous Constraints



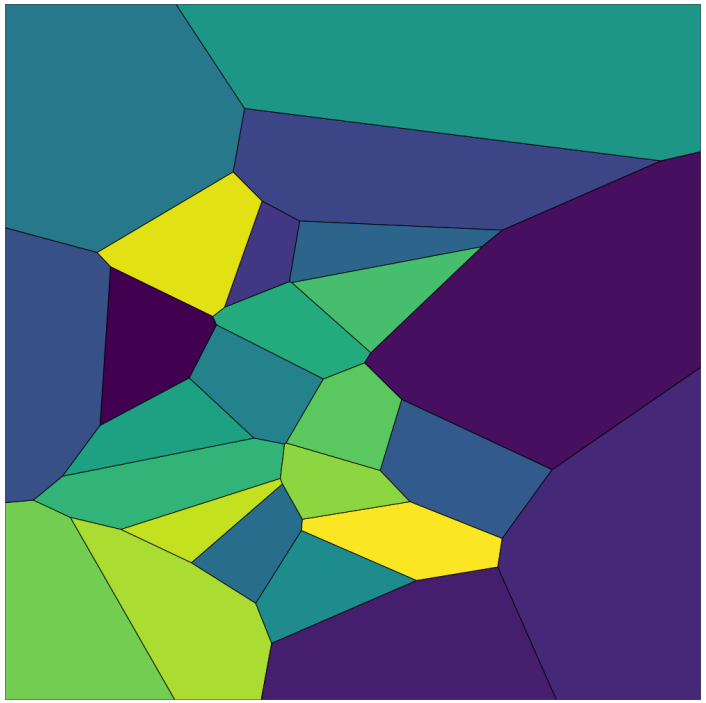
Voronoi Tessellations

- Dual graph to the Delaunay triangulation.
- Each point in a Voronoi cell is closer to its associated *generator* than to any other generator.



Using DelaunayTriangulation.jl

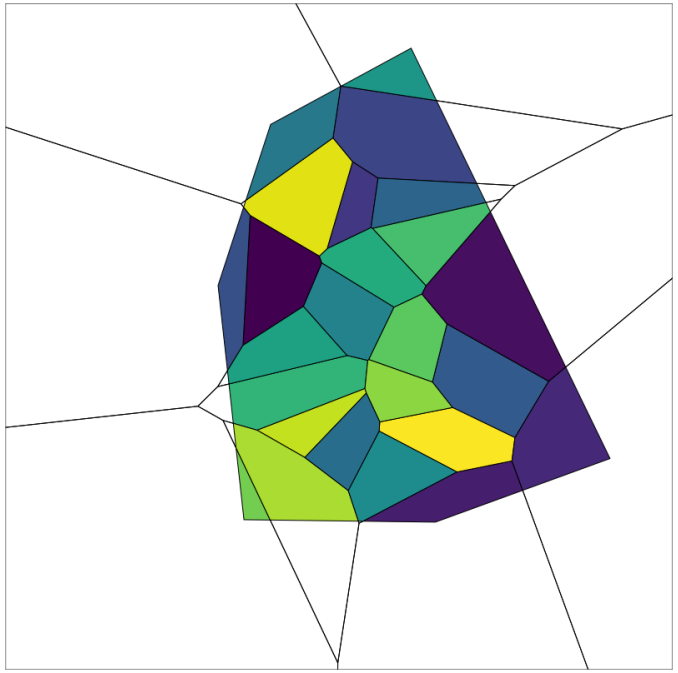
```
1 using DelaunayTriangulation, CairoMakie
2 points = randn(2, 25)
3 tri = triangulate(points)
4 vorn = voronoi(tri)
5 voronoiplot(vorn)
```



Clipped Voronoi Tessellations

- Support for clipping to convex polygons.

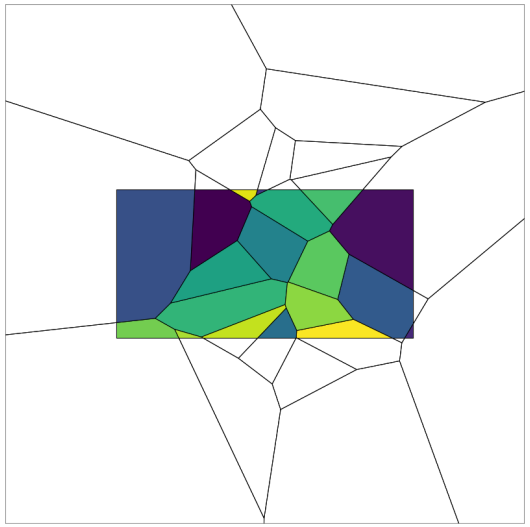
```
1  vorn = voronoi(tri; clip=true) # clips to the convex hull
2  voronoiplot(vorn)
```



Clipped Voronoi Tessellations

- Support for clipping to convex polygons.

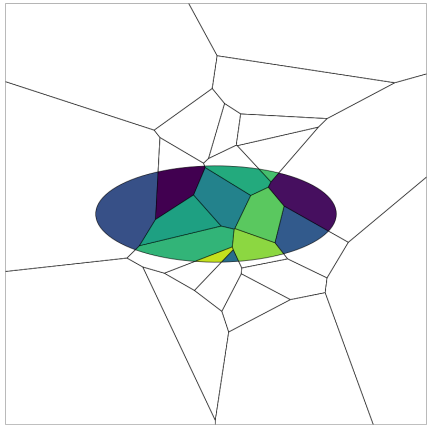
```
clip_points = [(-2.0, -1.0), (2.0, -1.0), (2.0, 1.0), (-2.0, 1.0)]
clip_vertices = [1, 2, 3, 4, 1]
vorn = voronoi(tri; clip=true,
    clip_polygon=(clip_points, clip_vertices)) # clips to the square
voronoiplot(vorn)
```



Clipped Voronoi Tessellations

- Support for clipping to convex polygons.

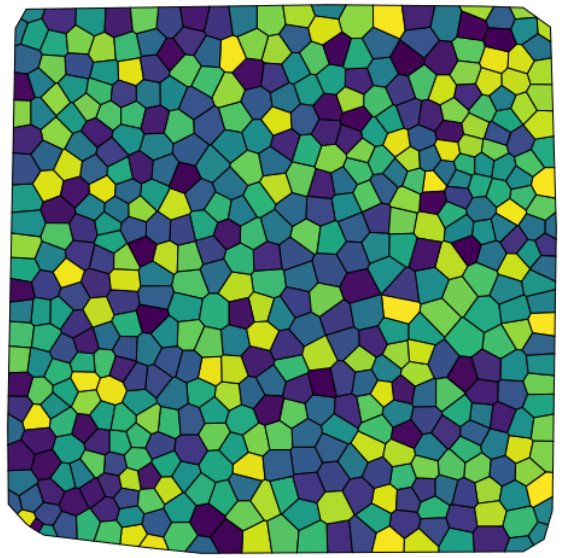
```
ellip = EllipticalArc((2.0, 0.0), (2.0, 0.0), (0.0, 0.0), 2.0, 0.8, 0.0)
t = LinRange(0, 1, 100)
clip_points = ellip.(t)[1:end-1]
clip_vertices = [1:length(clip_points); 1]
vorn = voronoi(tri; clip=true,
    clip_polygon=(clip_points, clip_vertices)) # clips to the ellipse
voronoiplot(vorn)
```



Centroidal Voronoi Tessellations

- Support for centroidal Voronoi tessellations.

```
points = rand(2, 500)
vorn = voronoi(triangulate(points); clip=true, smooth=true)
voronoiplot(vorn)
```



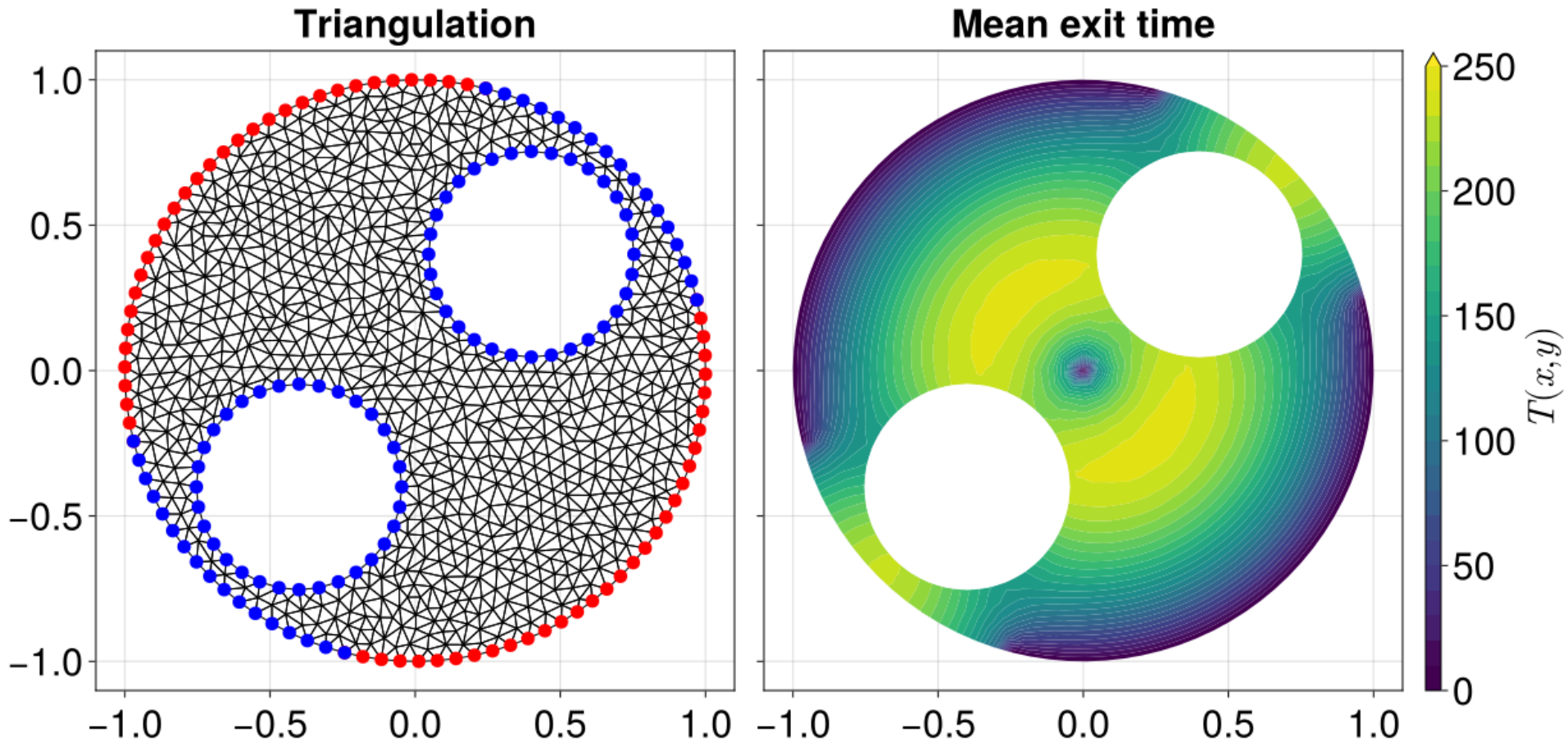
FiniteVolumeMethod.jl

- FiniteVolumeMethod.jl leverages DelaunayTriangulation.jl for meshing, solving PDEs of the form

$$\frac{\partial u(\boldsymbol{x}, t)}{\partial t} + \boldsymbol{\nabla} \cdot \boldsymbol{q}(\boldsymbol{x}, t, u) = S(\boldsymbol{x}, t, u)$$

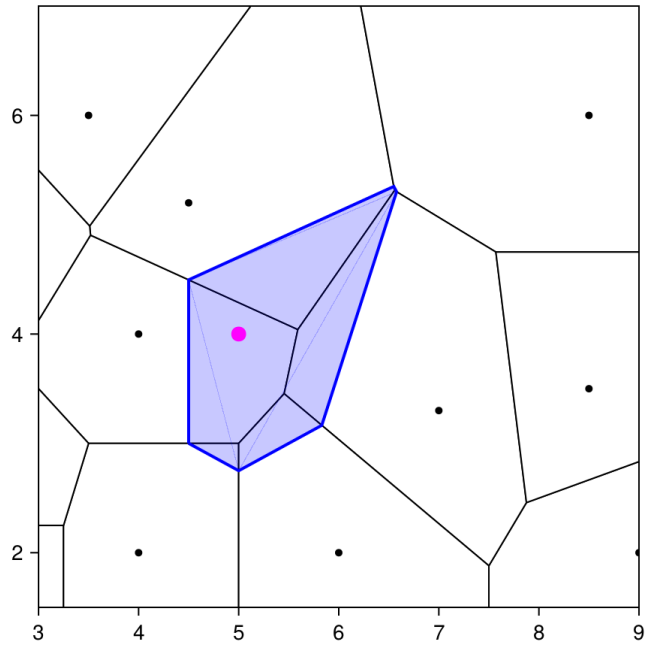
- Hooks into the SciML interface to support a wide range of solvers.

FiniteVolumeMethod.jl



NaturalNeighbours.jl

- Use interpolants of the form $f(\boldsymbol{x}_0) = \sum_{i \in N_0} \lambda_i f(\boldsymbol{x}_i)$.
- Sibson coordinates: $\lambda_i = A(\boldsymbol{x}_i)/A(\boldsymbol{x})$.



Conclusion

- Delaunay triangulations are a powerful tool for meshing and interpolation, among other applications.
- DelaunayTriangulation.jl is a feature-rich package for working with Delaunay triangulations and Voronoi tessellations.
- Documentation contains many other examples, details, and applications such as cell biology.
- Experimental support for spherical Delaunay triangulations and Voronoi tessellations.
- Also supports weighted triangulations and power diagrams.