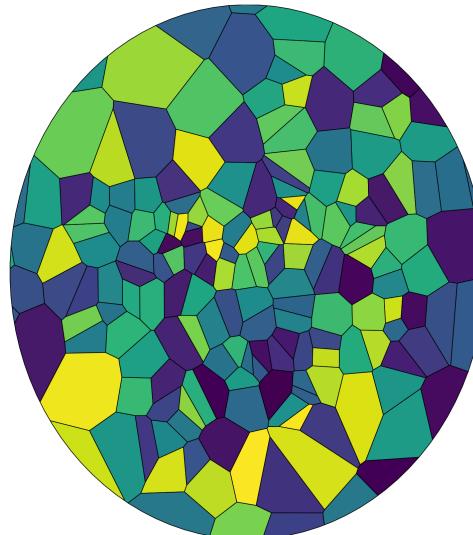
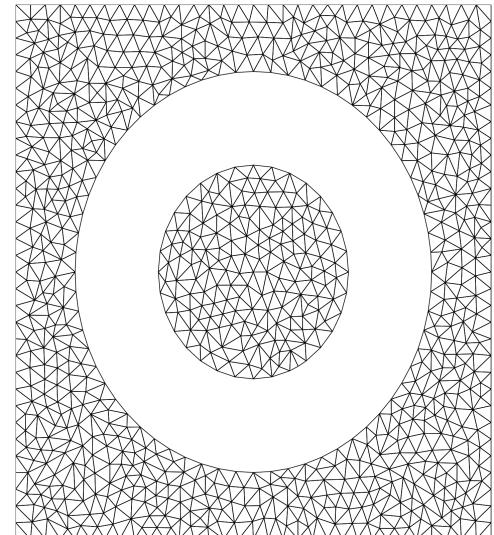


DelaunayTriangulation.jl

Two-dimensional Delaunay Triangulations and Voronoi
Tessellations in Julia



Daniel VandenHeuvel

d.vandenheuvel24@imperial.ac.uk

Department of Mathematics, Imperial College London

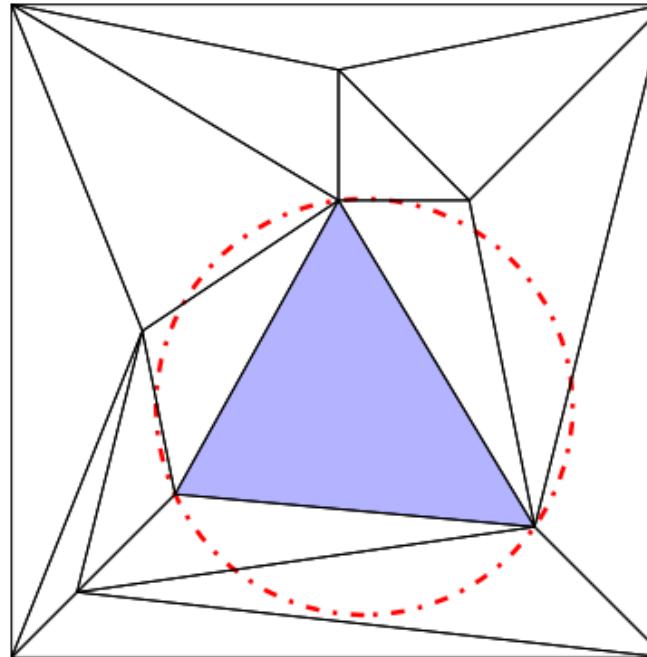
2024-10-09

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

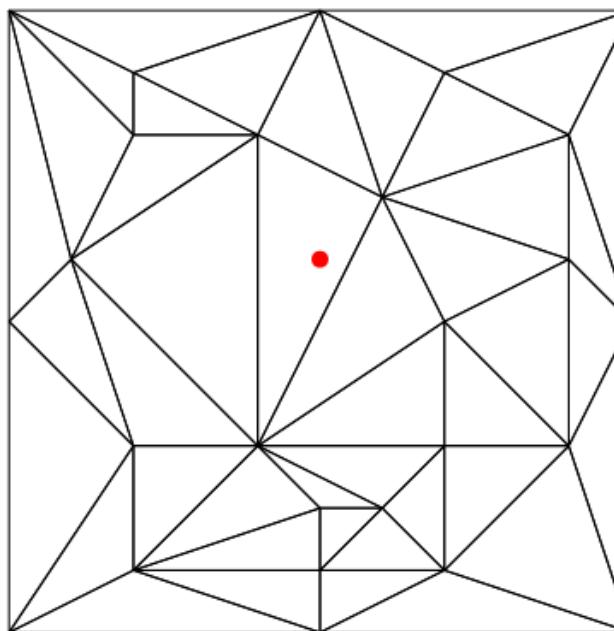
- A triangulation of a set of points \mathcal{P} in the plane such that no triangle's circumcircle contains any other point in the set.
- Maximizes the minimum angle over all other triangulations of \mathcal{P} .

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.

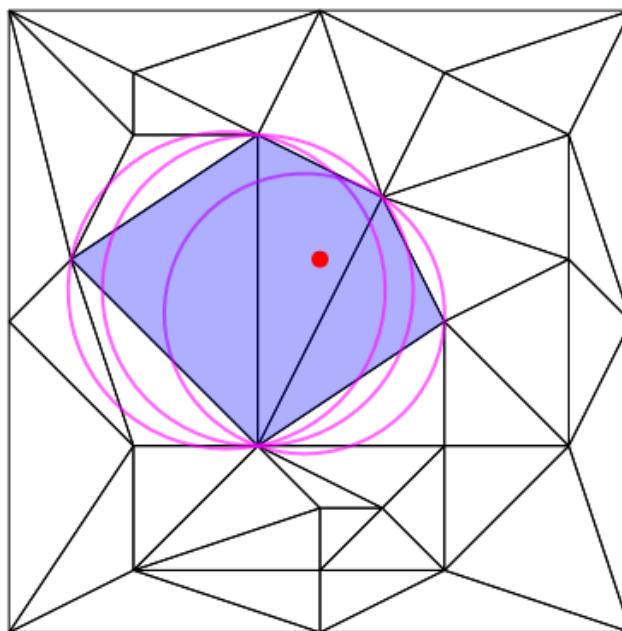
Bowyer-Watson Algorithm

- Triangulations are computed incrementally.



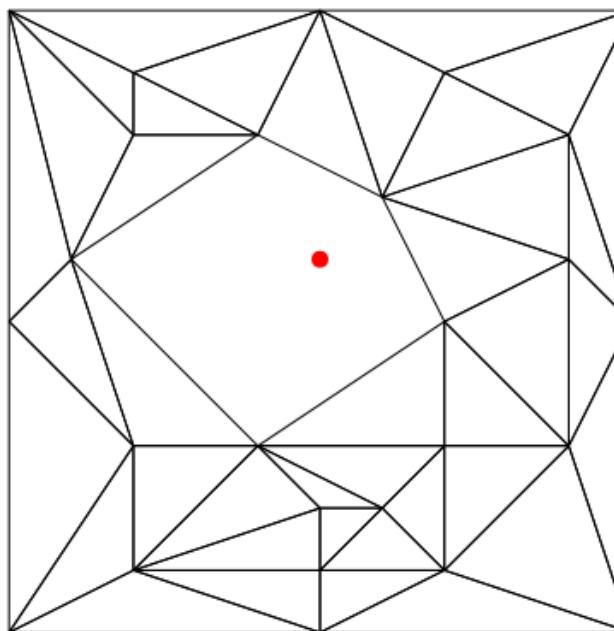
Bowyer-Watson Algorithm

- Triangulations are computed incrementally.



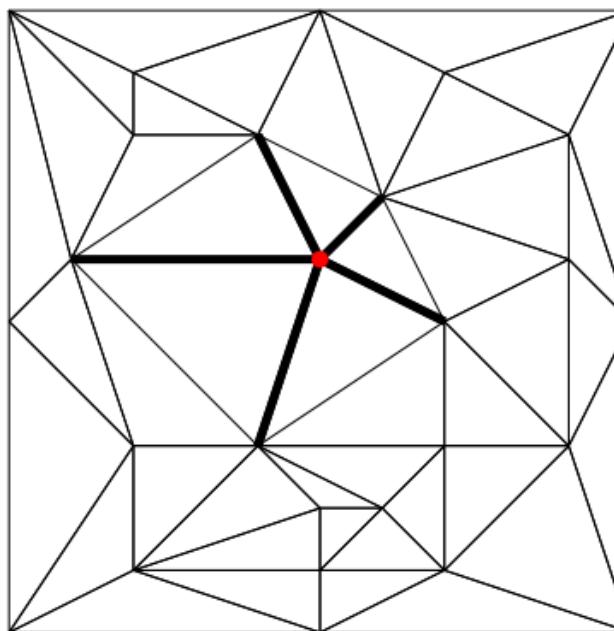
Bowyer-Watson Algorithm

- Triangulations are computed incrementally.



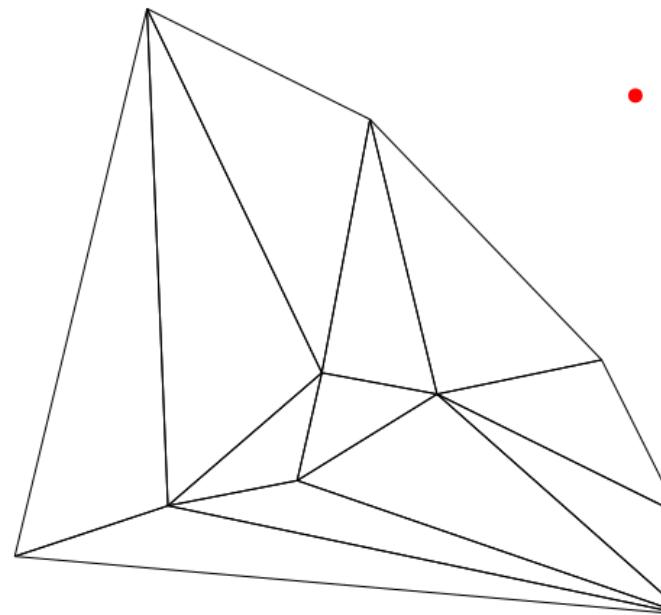
Bowyer-Watson Algorithm

- Triangulations are computed incrementally.



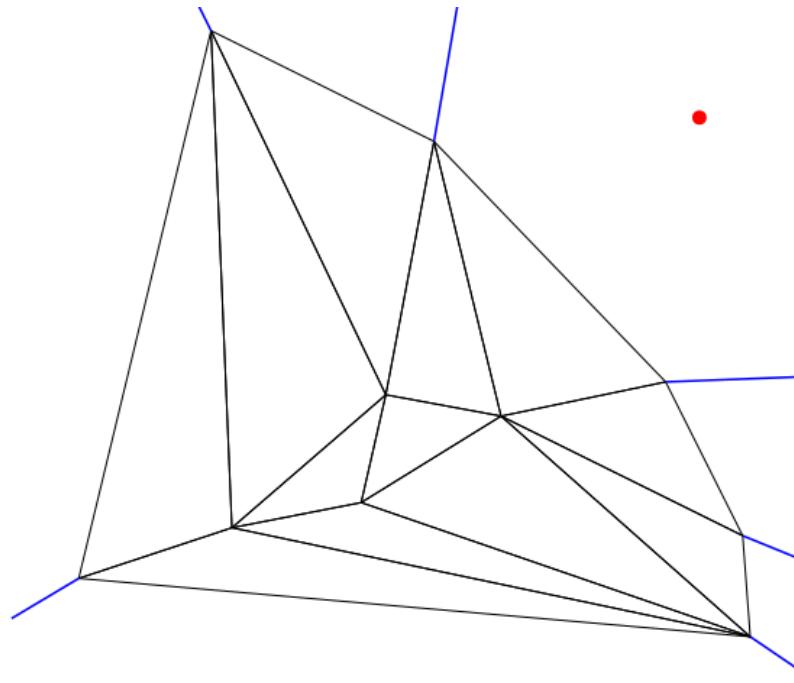
Ghost Vertices

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



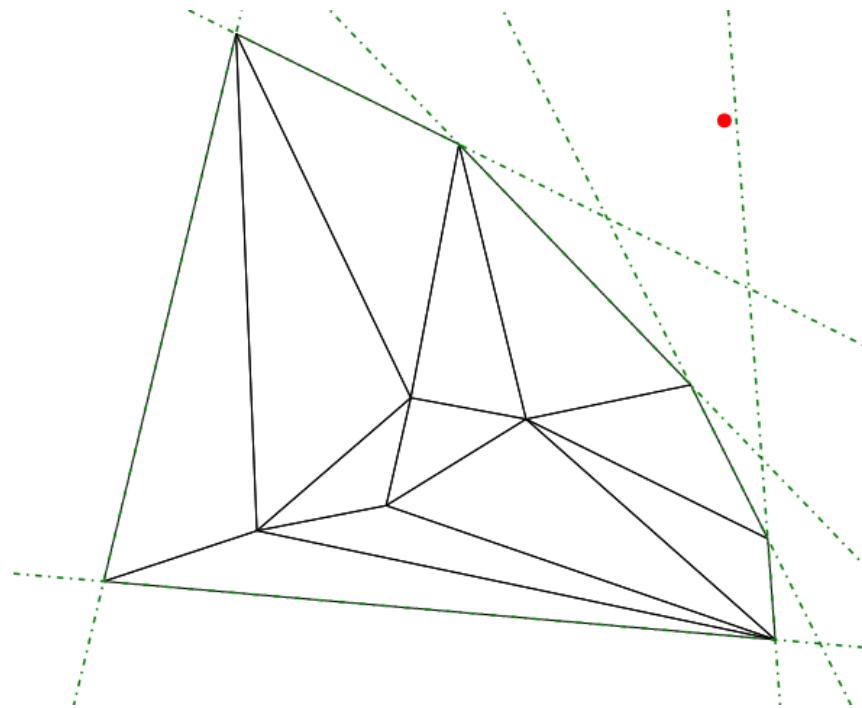
Ghost Vertices

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



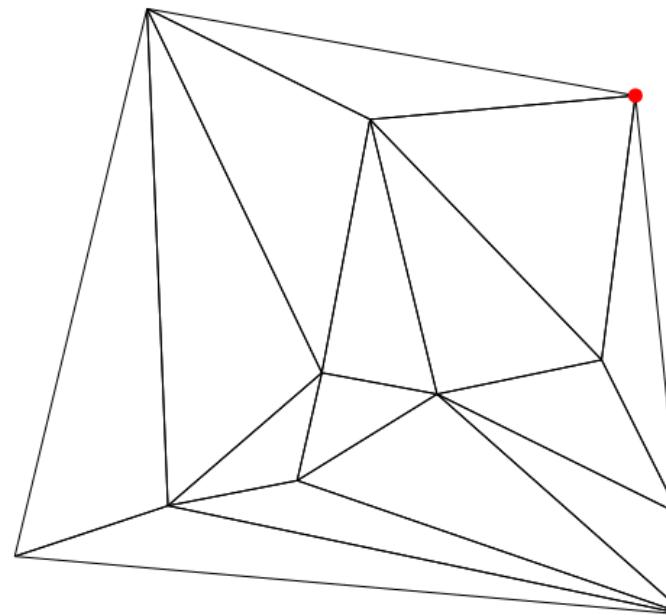
Ghost Vertices

- 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

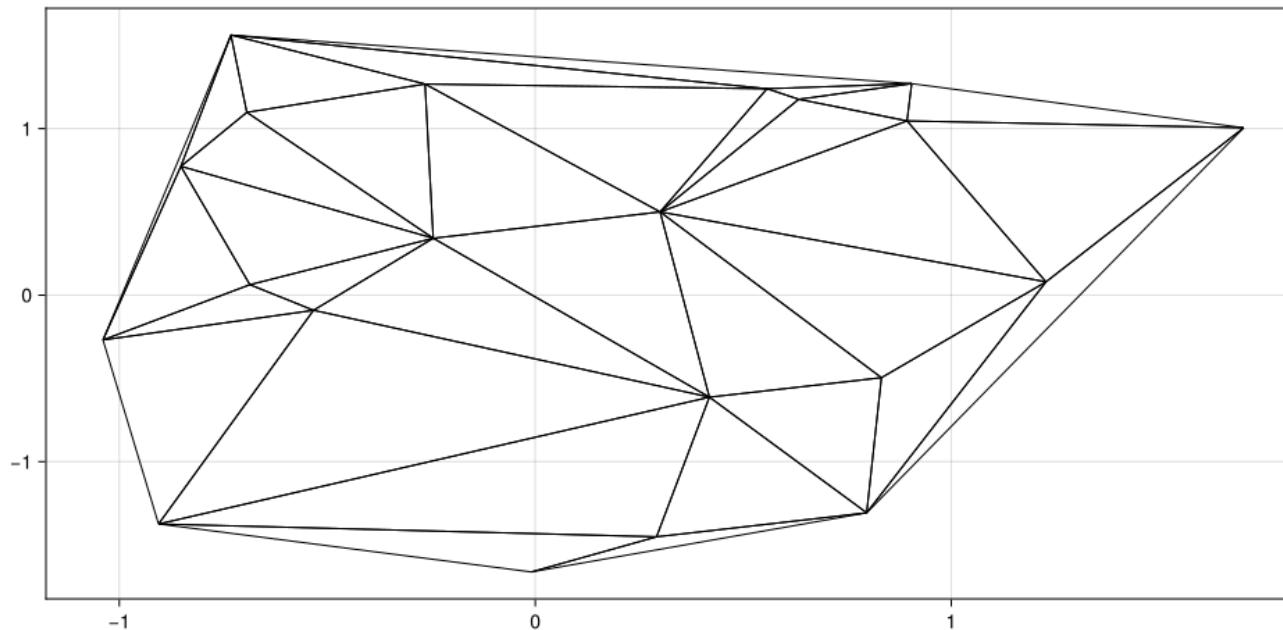
```
1 using DelaunayTriangulation  
2 points = [randn(2) for _ in 1:20]  
3 tri = triangulate(points)
```

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)
```

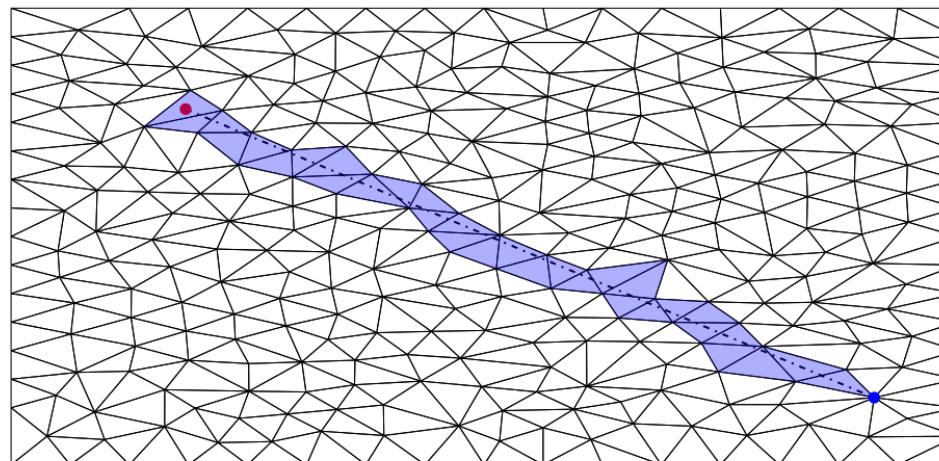
Using DelaunayTriangulation.jl

```
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))
```

(339, 410, 376)

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))
```

(269, 166, -1)

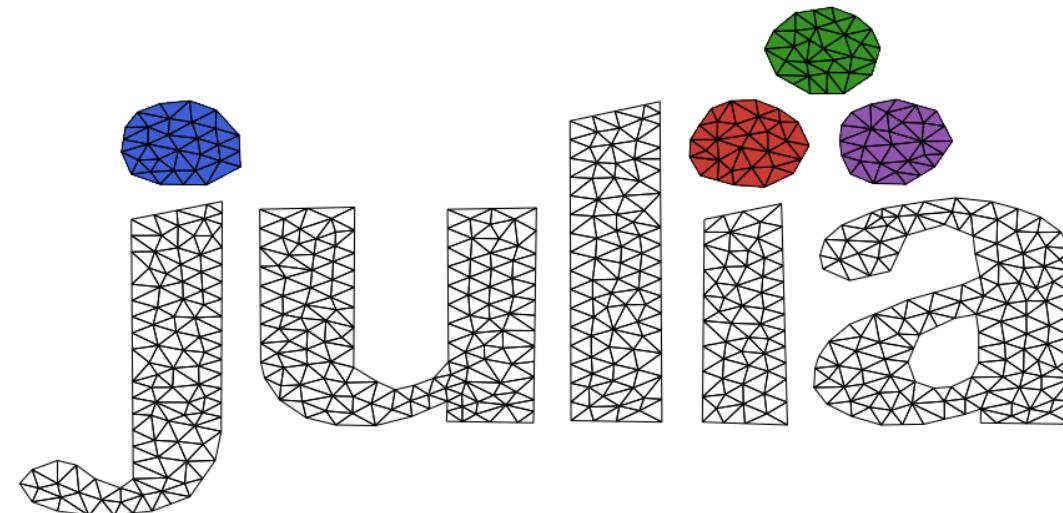
Using DelaunayTriangulation.jl

```
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)
```

(376, 127, 182)

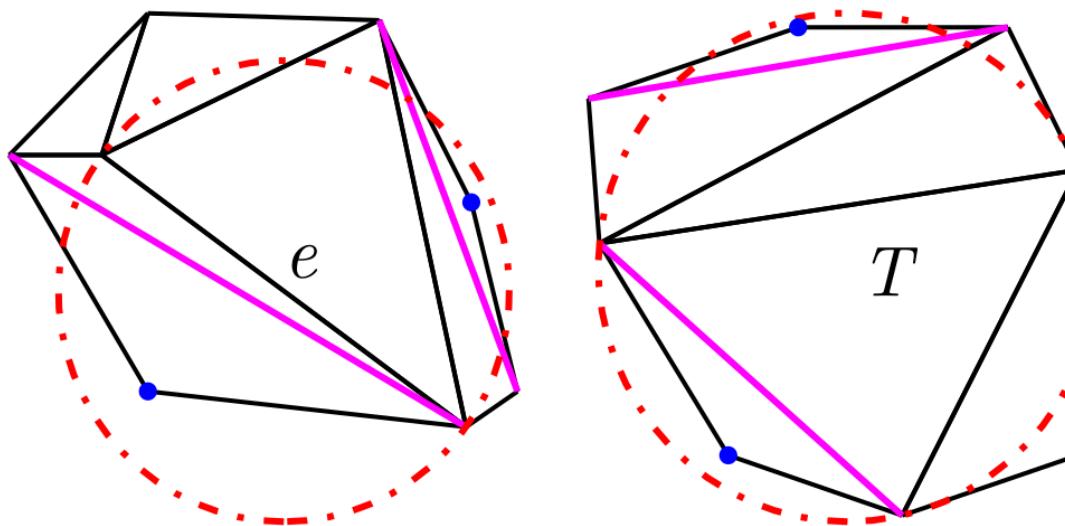
Constrained Triangulations

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



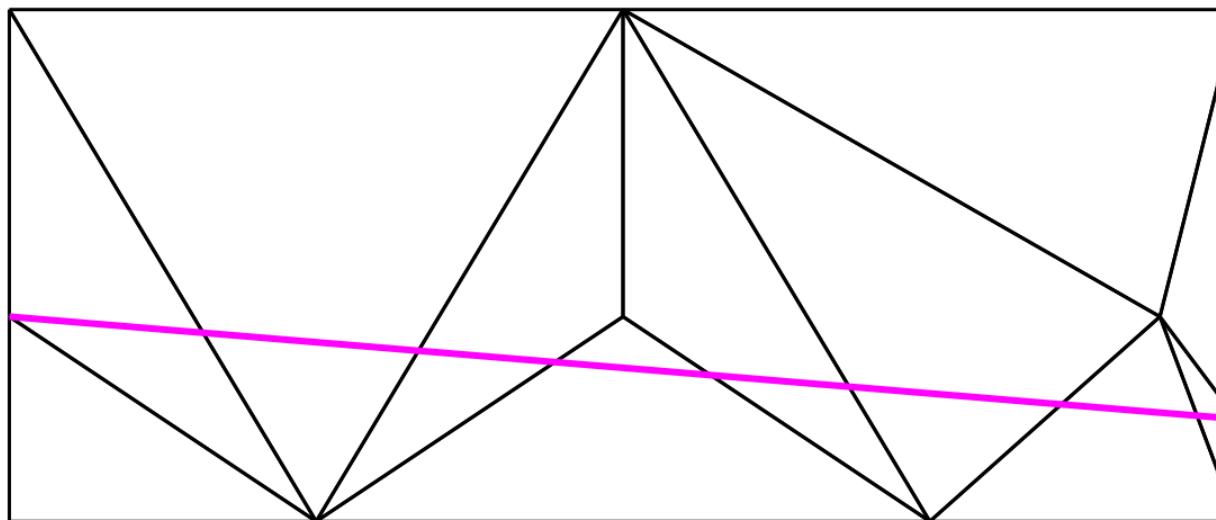
Constrained Delaunay Property

- 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.



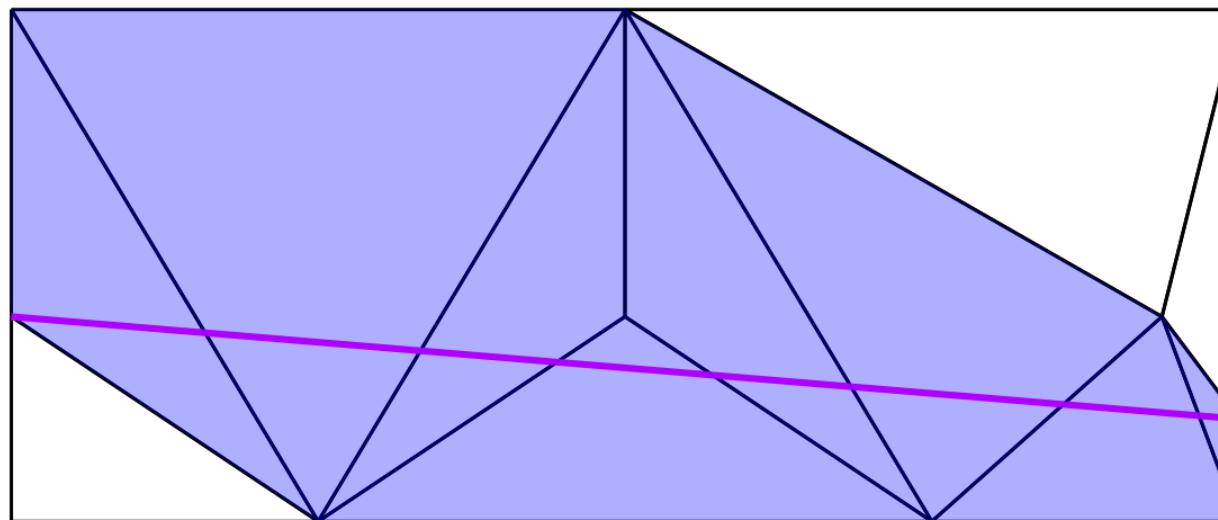
Inserting Segments

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



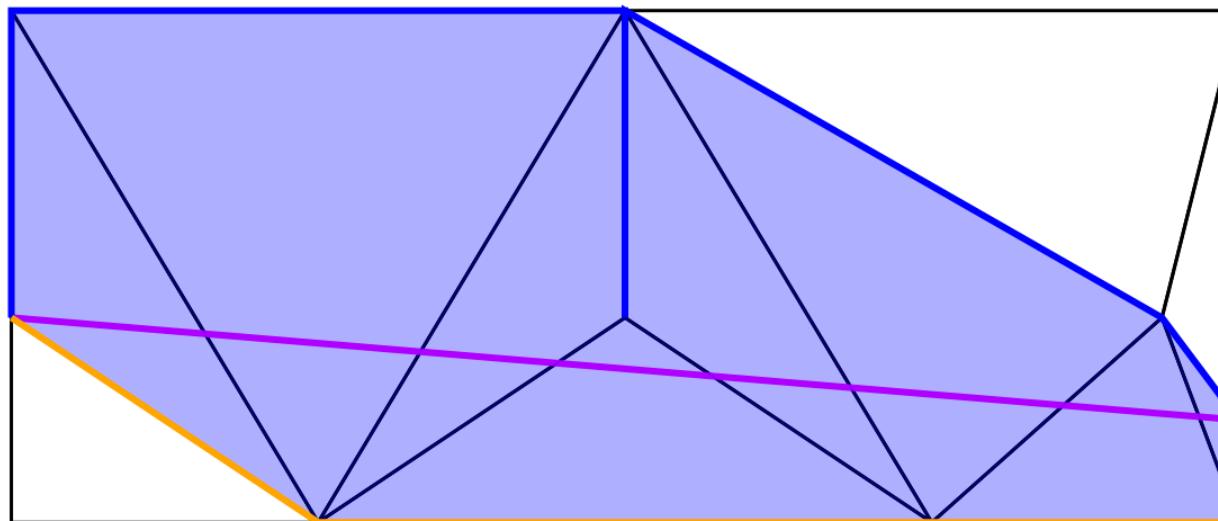
Inserting Segments

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



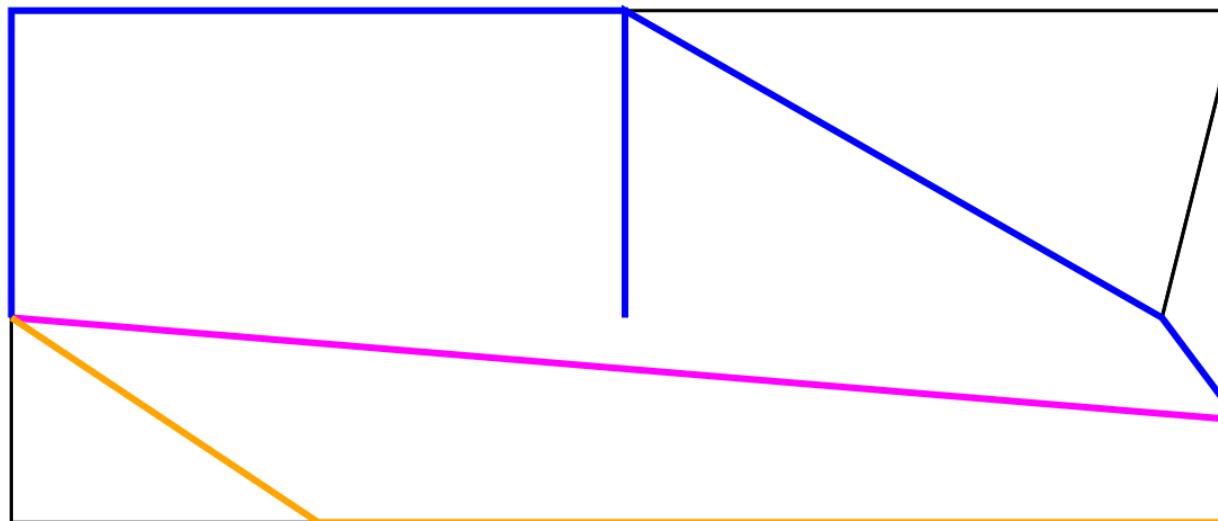
Inserting Segments

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



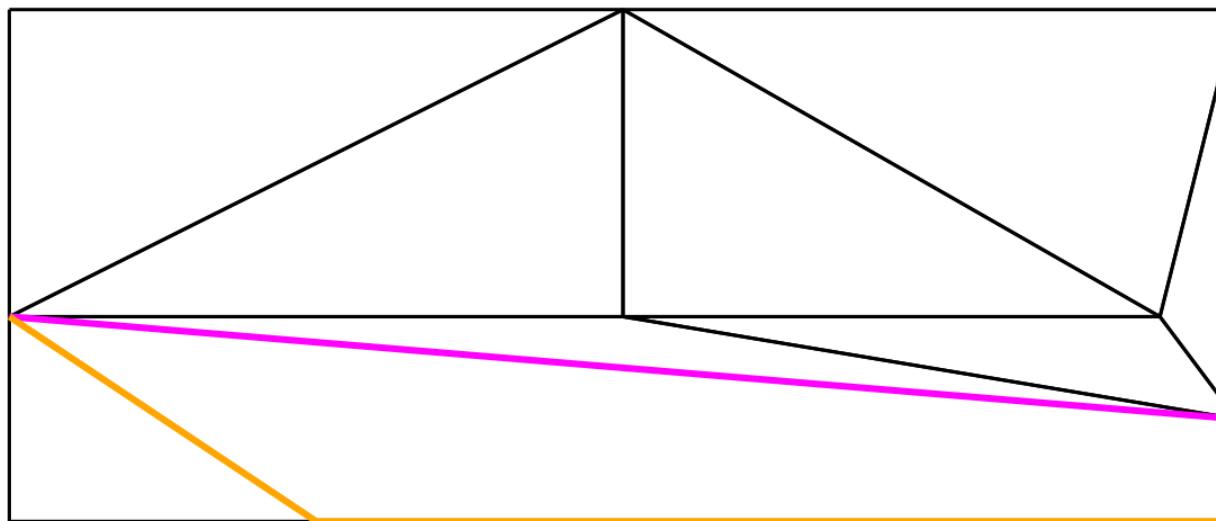
Inserting Segments

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



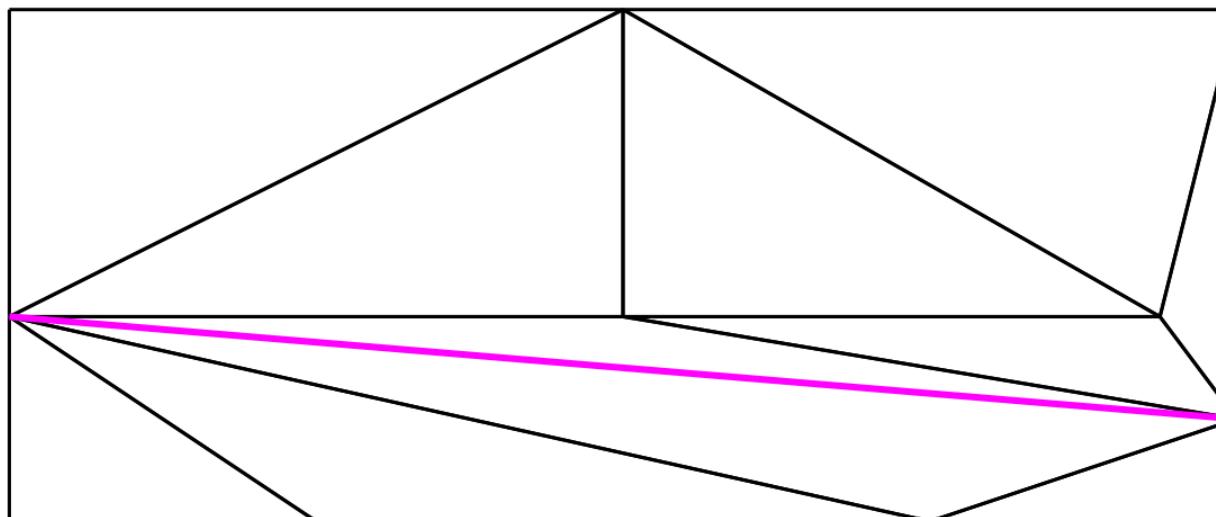
Inserting Segments

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



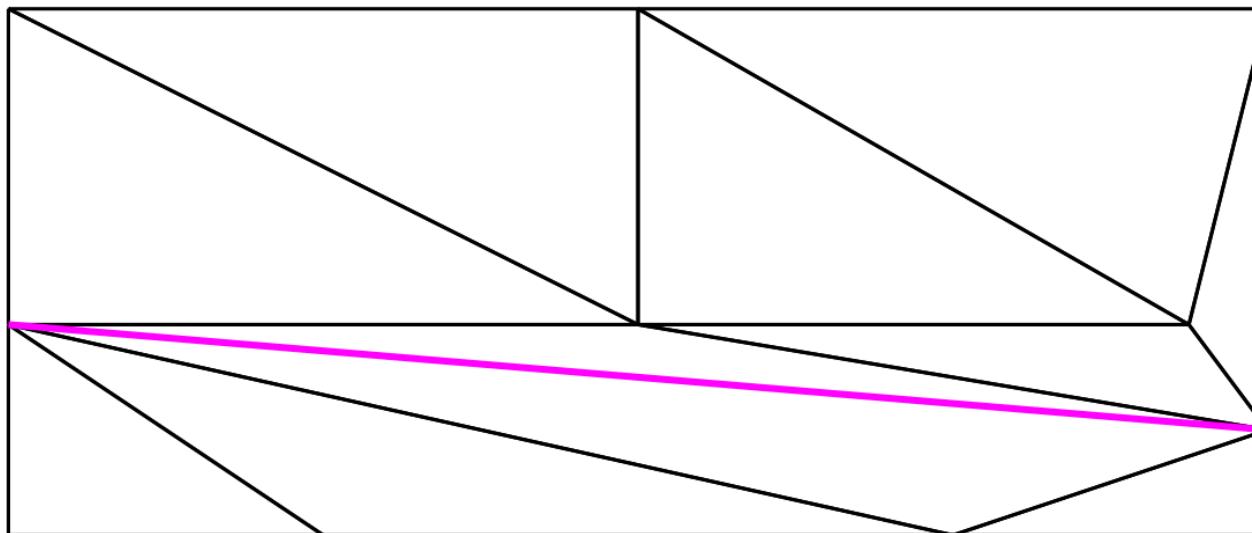
Inserting Segments

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



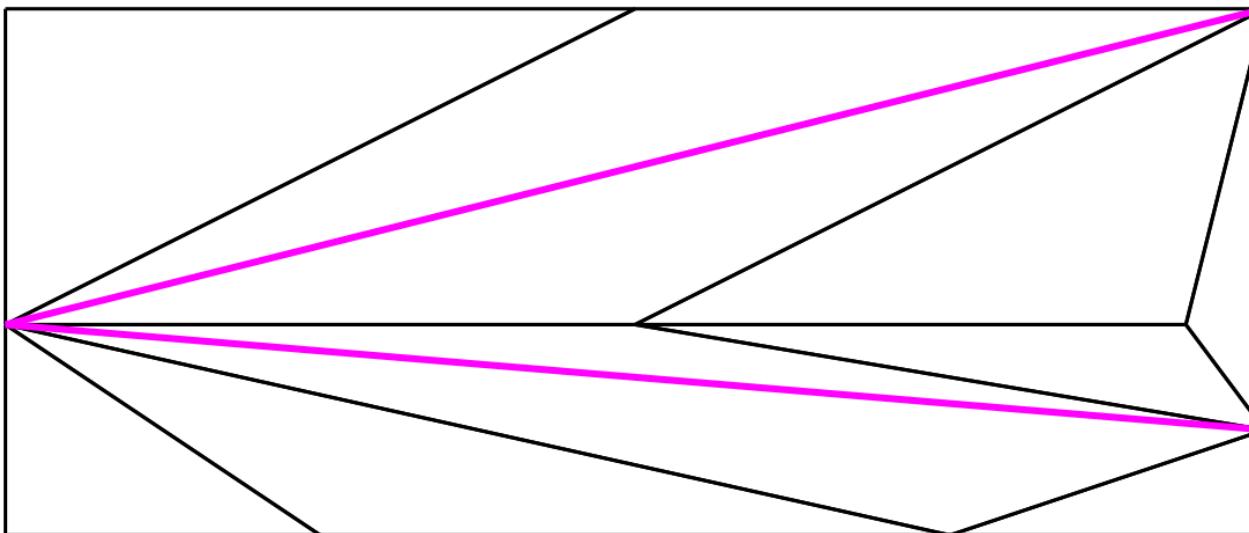
Using DelaunayTriangulation.jl

```
1 using DelaunayTriangulation, CairoMakie
2 points = [(0.0, 0.0), (0.0, 1.0), (0.0, 2.5), (2.0, 0.0),
3     (6.0, 0.0), (8.0, 0.0), (8.0, 0.5), (7.5, 1.0),
4     (4.0, 1.0), (4.0, 2.5), (8.0, 2.5)]
5 tri = triangulate(points; segments=Set([(2, 7)]))
6
7 triplot(tri)
```



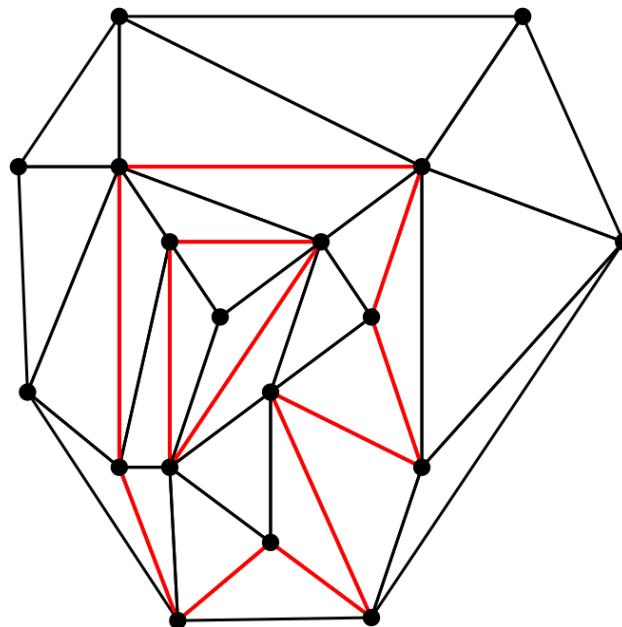
Using DelaunayTriangulation.jl

```
1 using DelaunayTriangulation, CairoMakie
2 points = [(0.0, 0.0), (0.0, 1.0), (0.0, 2.5), (2.0, 0.0),
3             (6.0, 0.0), (8.0, 0.0), (8.0, 0.5), (7.5, 1.0),
4             (4.0, 1.0), (4.0, 2.5), (8.0, 2.5)]
5 tri = triangulate(points; segments=Set([(2, 7)]))
6 add_segment!(tri, 2, 11)
7 triplot(tri)
```



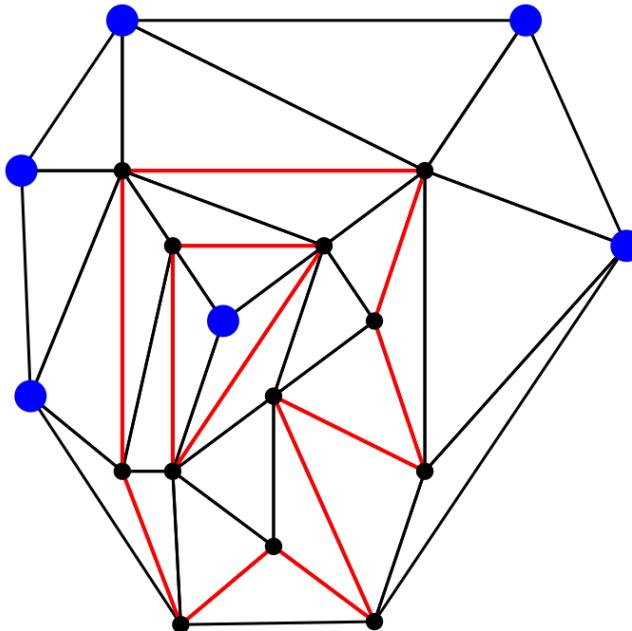
Boundaries

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



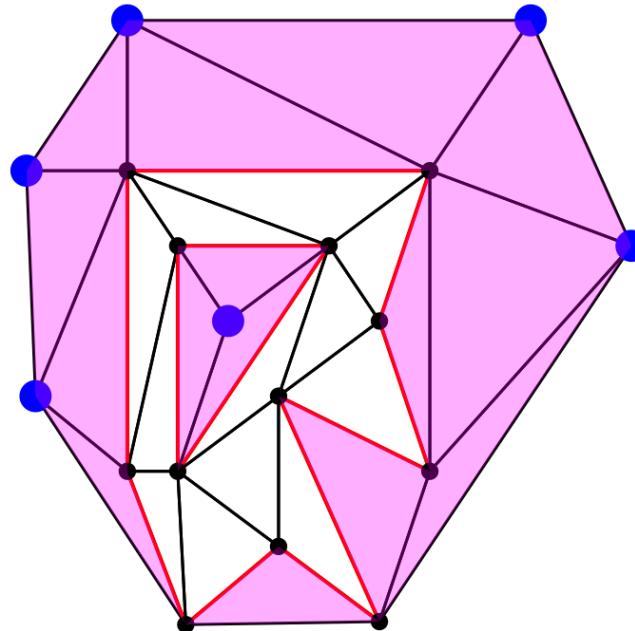
Boundaries

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



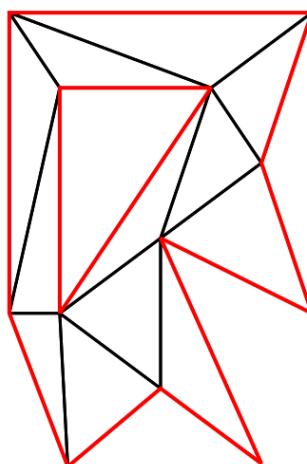
Boundaries

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



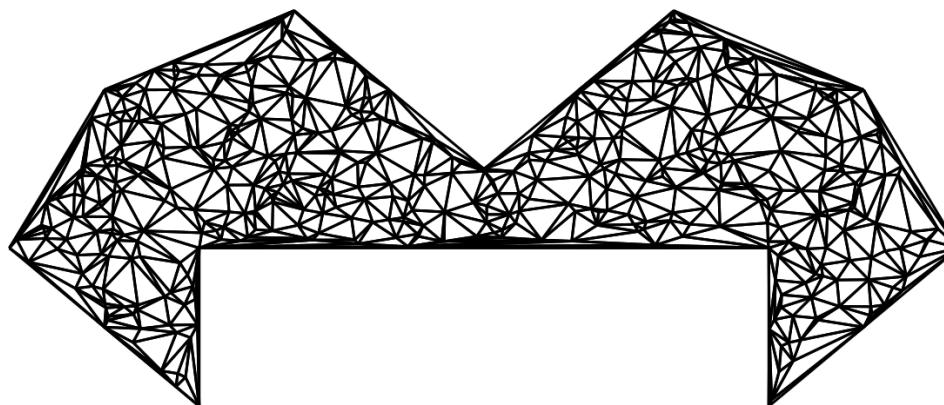
Boundaries

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



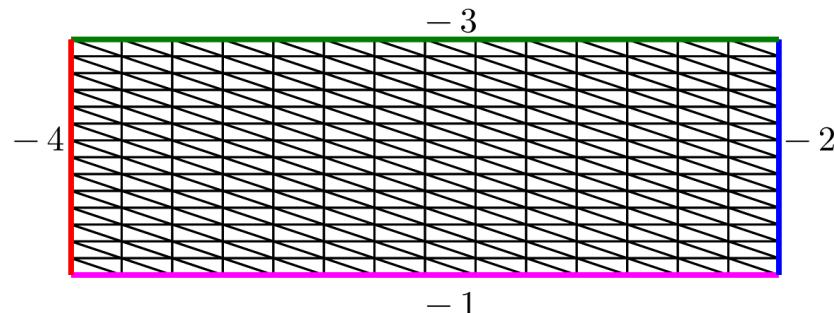
Using DelaunayTriangulation.jl

```
1 using DelaunayTriangulation, CairoMakie
2 boundary = [(0.0, 0.0), (0.1, -0.1), (0.2, -0.2), (0.2, 0.0),
3              (0.8, 0.0), (0.8, -0.2), (0.9, -0.1), (1.0, 0.0),
4              (0.9, 0.2), (0.7, 0.3), (0.6, 0.2), (0.5, 0.1),
5              (0.4, 0.2), (0.3, 0.3), (0.1, 0.2), (0.0, 0.0)]
6 points = [(rand(), 0.5rand() - 0.2) for _ in 1:1000]
7 boundary_nodes, points = convert_boundary_points_to_indices(
8    boundary; existing_points=points)
9 tri = triangulate(points; boundary_nodes)
10 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

```
1 using DelaunayTriangulation, CairoMakie
2 section_1 = [(0.0, 0.0), (0.1, -0.1)]
3 section_2 = [(0.1, -0.1), (0.2, -0.2), (0.2, 0.0),
4               (0.8, 0.0), (0.8, -0.2)]
5 section_3 = [(0.8, -0.2), (0.9, -0.1), (1.0, 0.0)]
6 section_4 = [(1.0, 0.0), (0.9, 0.2), (0.7, 0.3), (0.6, 0.2), (0.5, 0.1),
7               (0.4, 0.2), (0.3, 0.3), (0.1, 0.2), (0.0, 0.0)]
8 boundary = [section_1, section_2, section_3, section_4]
9 points = [(rand(), 0.5rand() - 0.2) for _ in 1:1000]
10 boundary_nodes, points = convert_boundary_points_to_indices(
11     boundary; existing_points=points)
12 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)
```

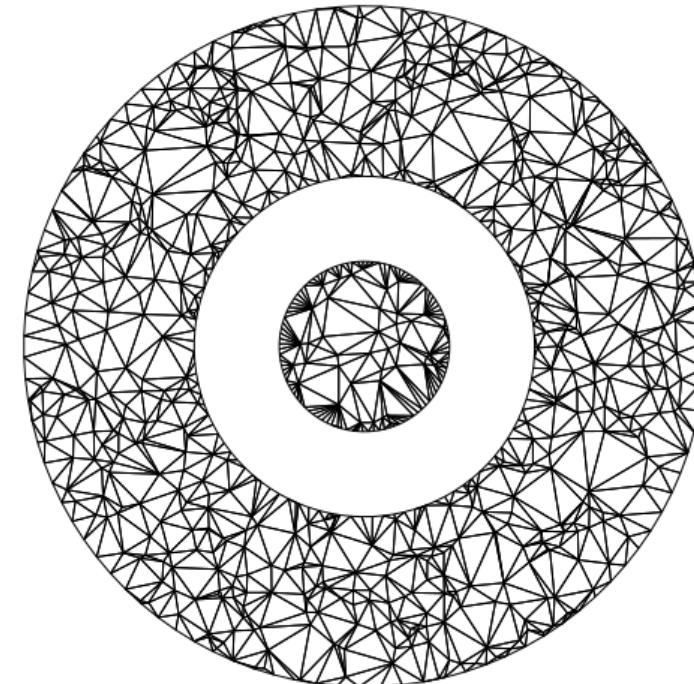
Set{Int64} with 5 elements:

```
1005
1002
1006
1003
1004
```

Using DelaunayTriangulation.jl

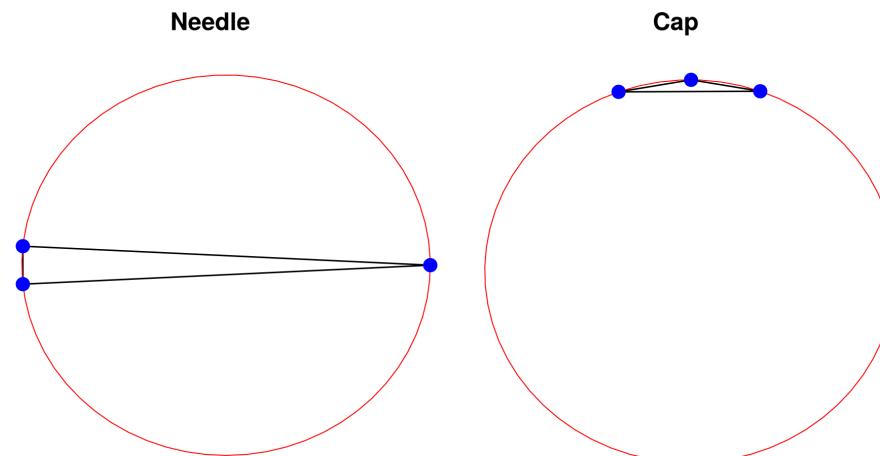
- Disjoint domains and holes follow the same representation.

```
1 using DelaunayTriangulation
2 using CairoMakie
3 θ = LinRange(0, 2pi, 100);
4 θ = [θ[1:99]; θ[1]];
5 θr = reverse(θ)
6 otrx, otry = 10cos.(θ), 10sin.(θ)
7 innx, inny = 5cos.(θr), 5sin.(θr)
8 inn2x, inn2y = 2.5cos.(θ), 2.5sin.(θ)
9 x = [[otrx], [innx], [inn2x]]
10 y = [[otry], [inny], [inn2y]]
11 points = [(20rand() - 10, 20rand() - 10)
12           for _ in 1:1000]
13 boundary_nodes, points =
14     convert_boundary_points_to_indices(
15         x, y; existing_points=points)
16 tri = triangulate(points;
17                   boundary_nodes)
18 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

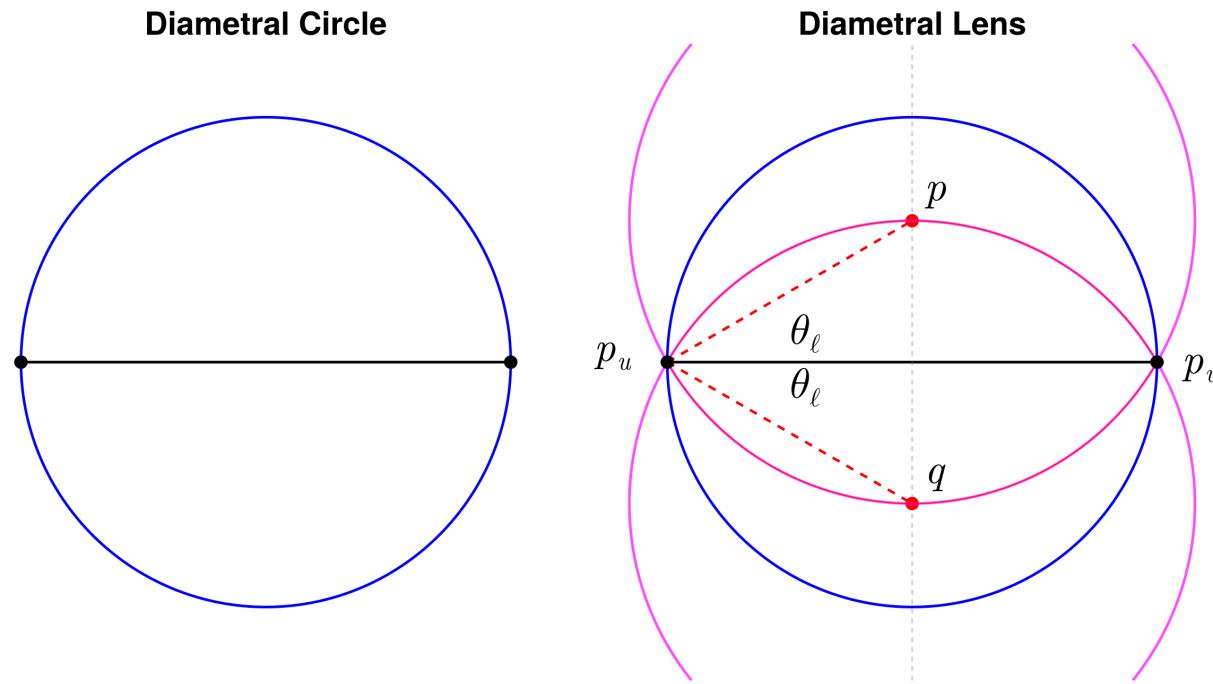
- Assess triangles using the *radius-edge ratio* $\rho = R/\ell_{\min}$.
- $1/\sqrt{3} \leq \rho < \infty$, with $1/\sqrt{3}$ attained for equilateral triangles.
- Related to the smallest angle by

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

Thus, controlling ρ controls the smallest angle.

Mesh Encroachment

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



Ruppert's Algorithm

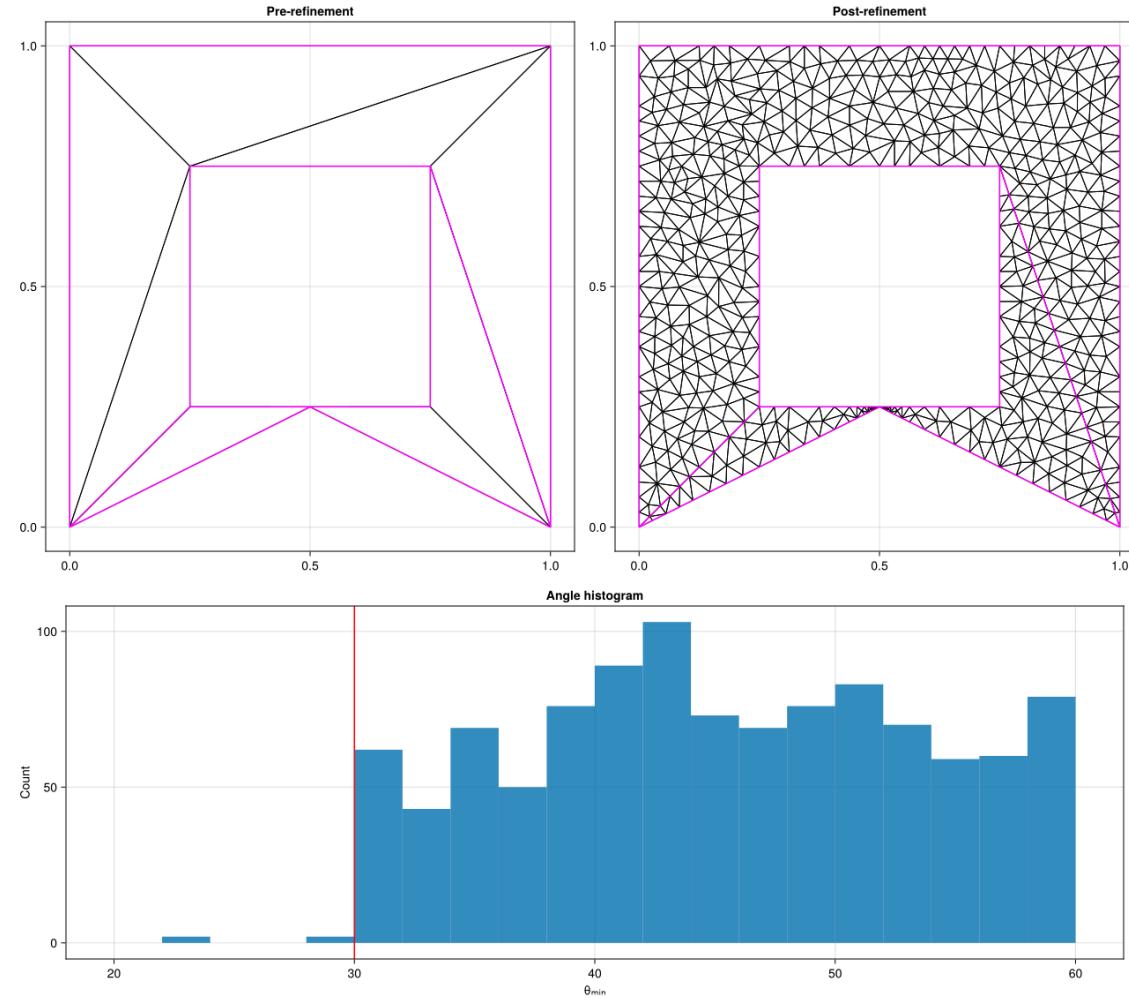
```
1  function refine(tri)
2      split_all_encroached_segments!(tri)
3      while has_bad_triangles(tri)
4          T = get_bad_triangle(tri)
5          insert_circumcenter!(tri, T)
6          if circumcenter_encroaches(tri)
7              undo_insertion!(tri, T)
8              split_all_encroached_segments!(tri)
9          else
10             for V in new_triangles(tri)
11                 p, A = radius_edge_ratio(V), area(V)
12                 ( $p > p_{\max}$  ||  $A > A_{\max}$ ) && mark_bad!(tri, V)
13             end
14         end
15     end
16 end
```

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

Using DelaunayTriangulation.jl

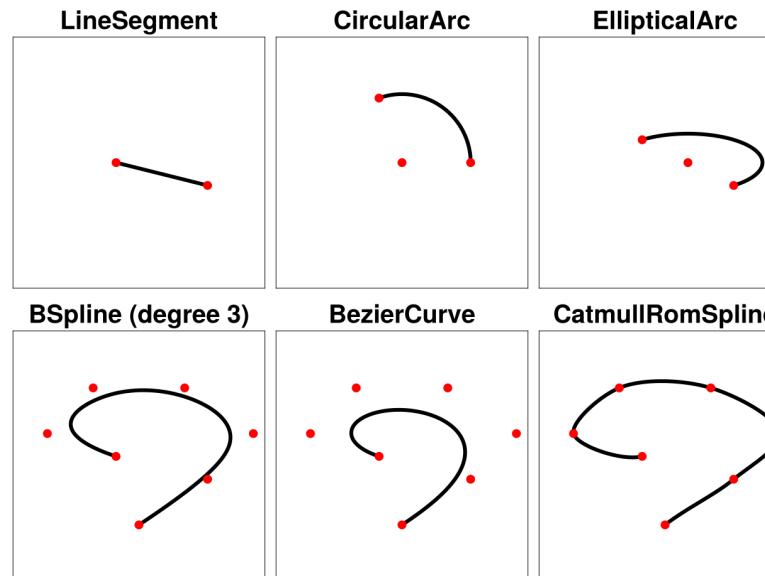
```
1 using DelaunayTriangulation, CairoMakie
2 outer = [(0.0, 0.0), (0.5, 0.2499), (1.0, 0.0), (1.0, 1.0), (0.0, 1.0), (0.0, 0.0)
3 inner = [(0.25, 0.25), (0.25, 0.75), (0.75, 0.75), (0.75, 0.25), (0.25, 0.25)]
4 boundary = [[outer], [inner]]
5 boundary_nodes, points = convert_boundary_points_to_indices(boundary)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

Using DelaunayTriangulation.jl



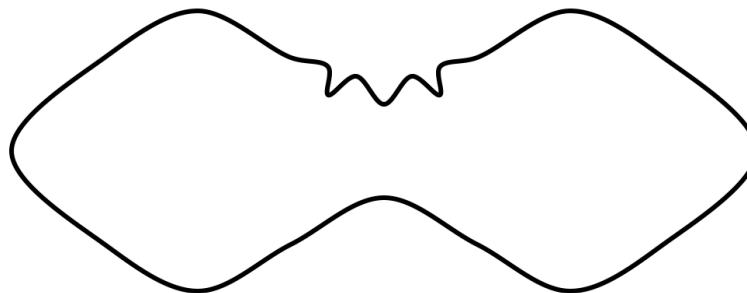
Curve-Bounded Domains

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



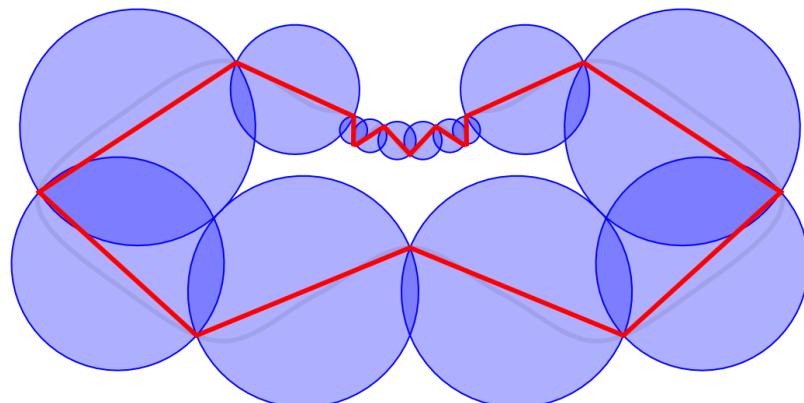
Boundary Enrichment

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



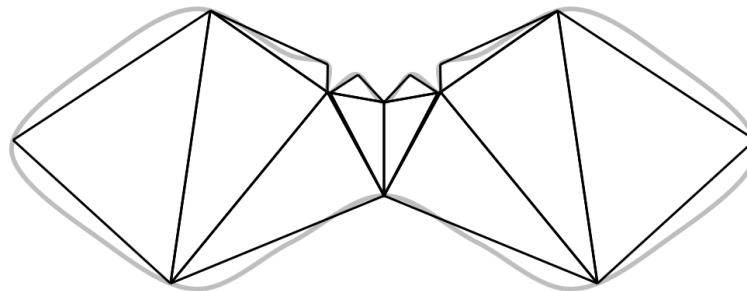
Boundary Enrichment

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



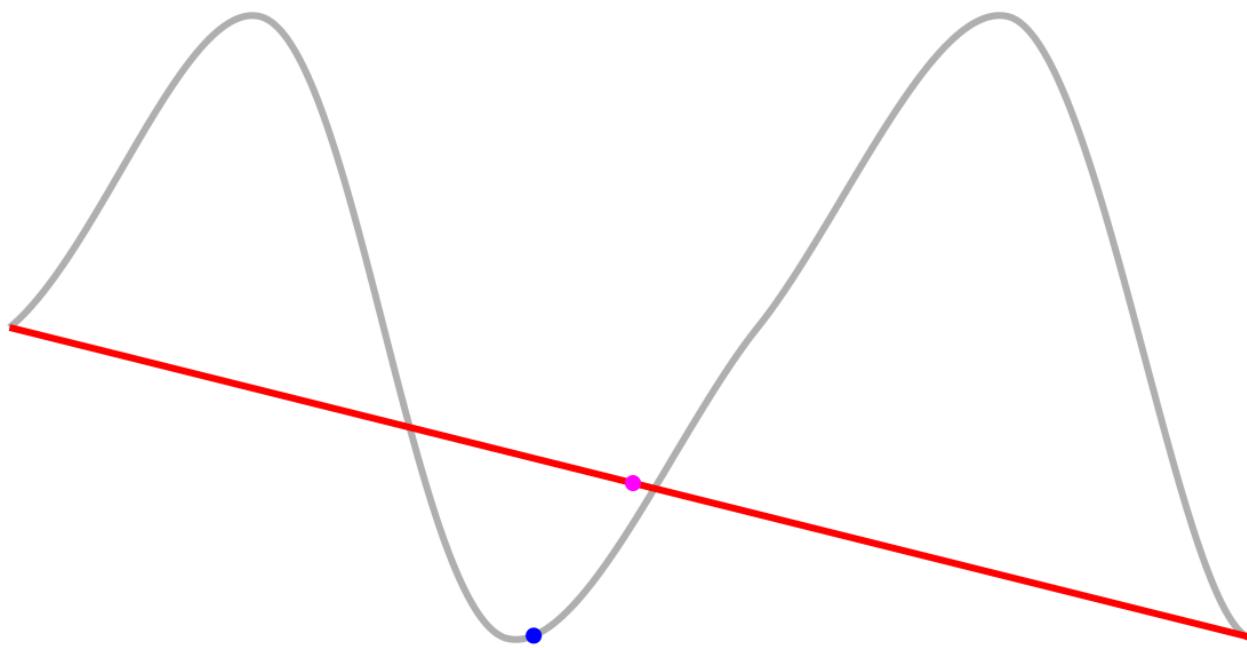
Boundary Enrichment

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



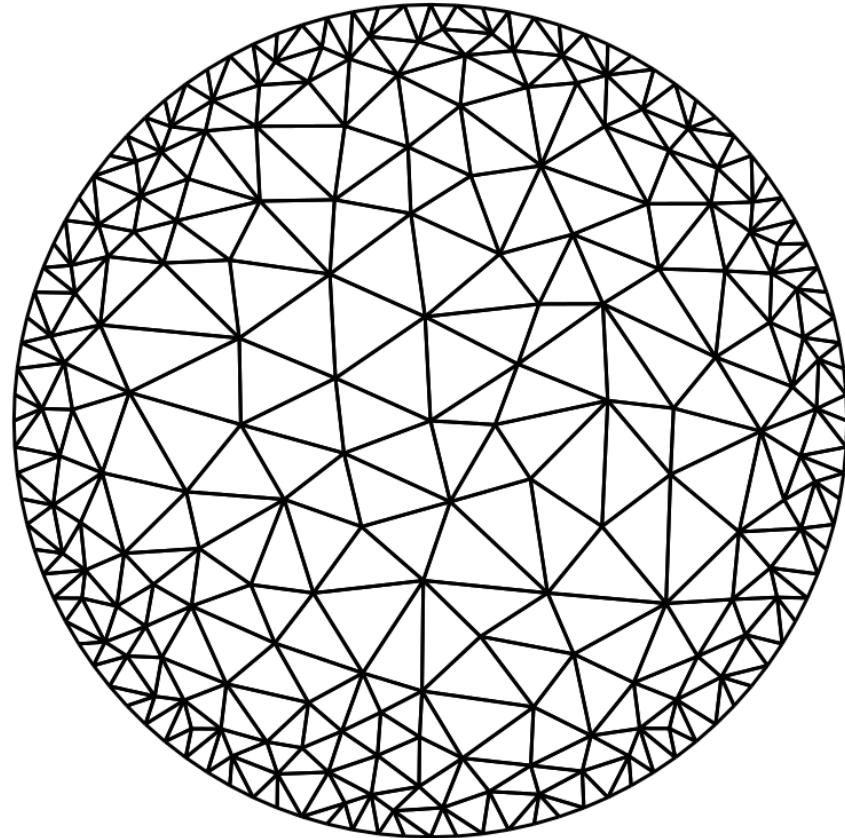
Curve-Bounded Refinement

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

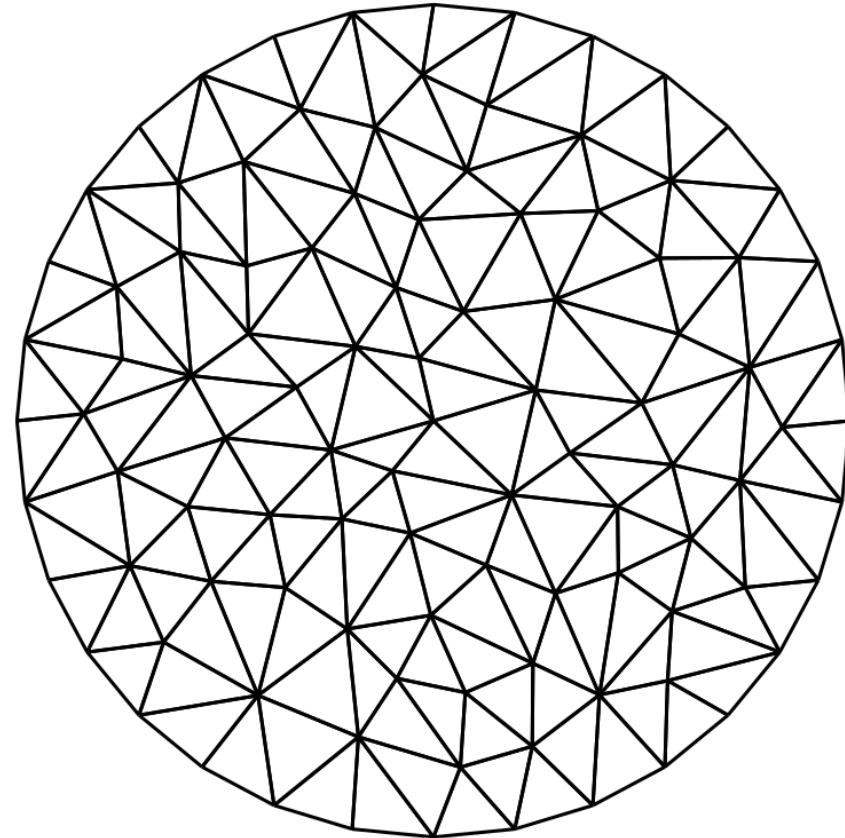


Comparison with Discretisation

Manual

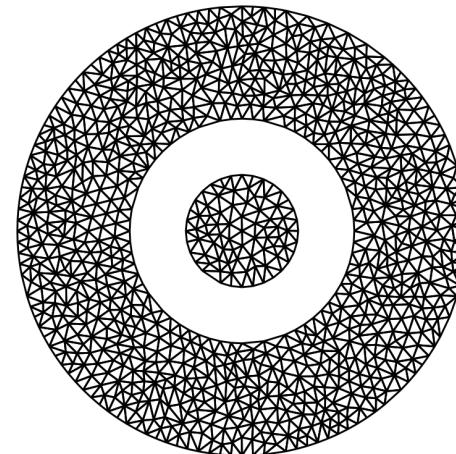


CircularArc



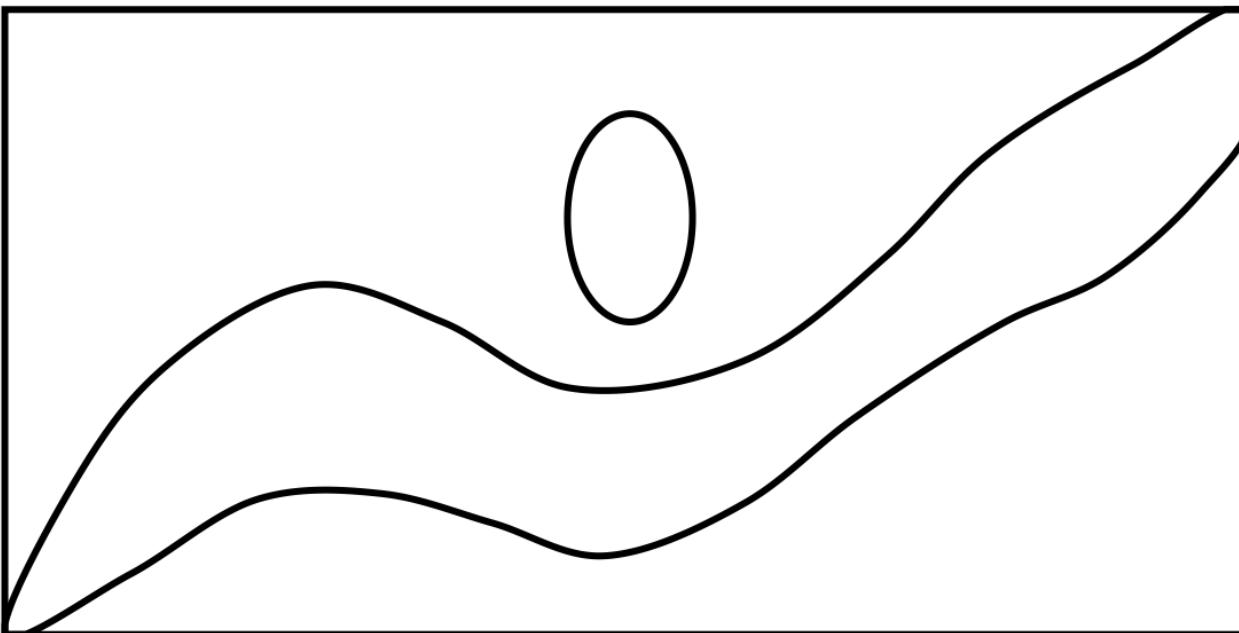
Using DelaunayTriangulation.jl

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



Heterogenous Constraints

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

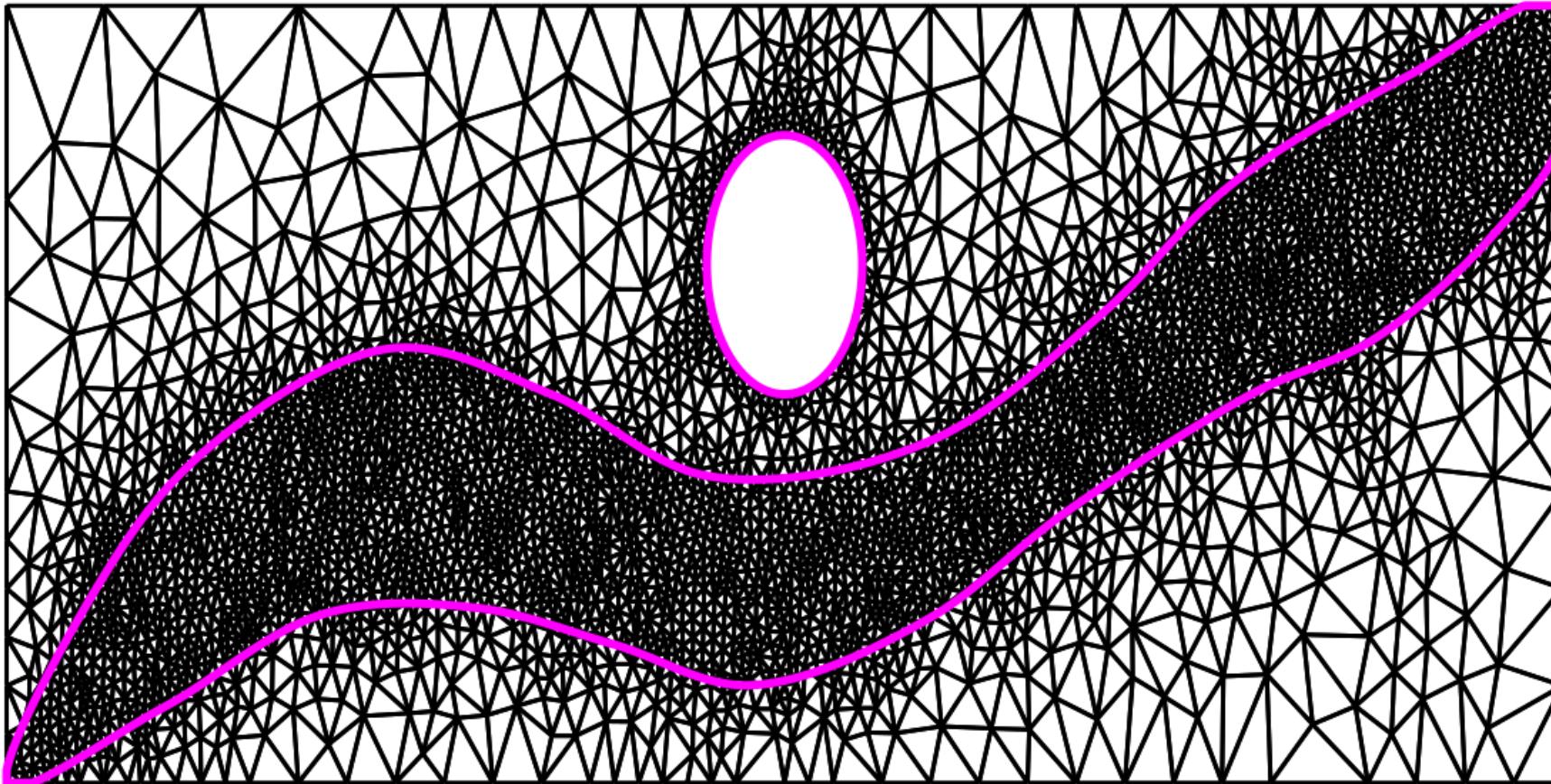


Heterogenous Constraints

- Use `custom_constraint`.

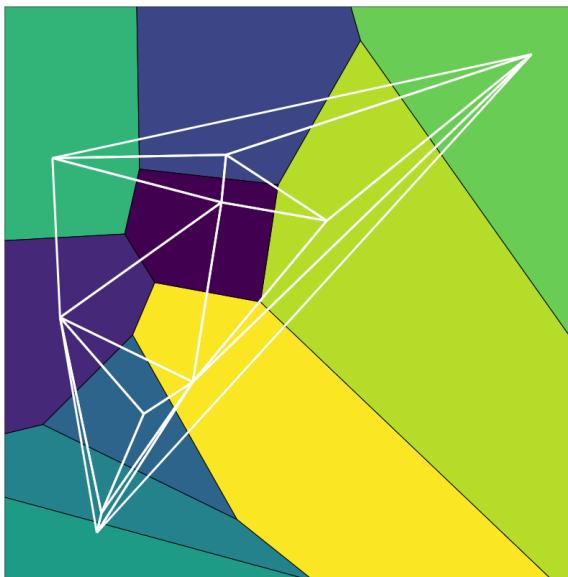
```
1 domain = [A, B, C, D, A]
2 river = [A, N, O, P, Q, R, S, T, U, V, W, Z, C, M, L, K, J, I, H, G, F, E, A]
3 A1, B1 = (50.0, 20.0), (50.0, 25.0)
4 cir = CircularArc(B1, B1, A1, positive=false)
5 boundary_nodes, points = convert_boundary_points_to_indices(domain)
6 river_spl = CatmullRomSpline(river)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

Heterogenous Constraints



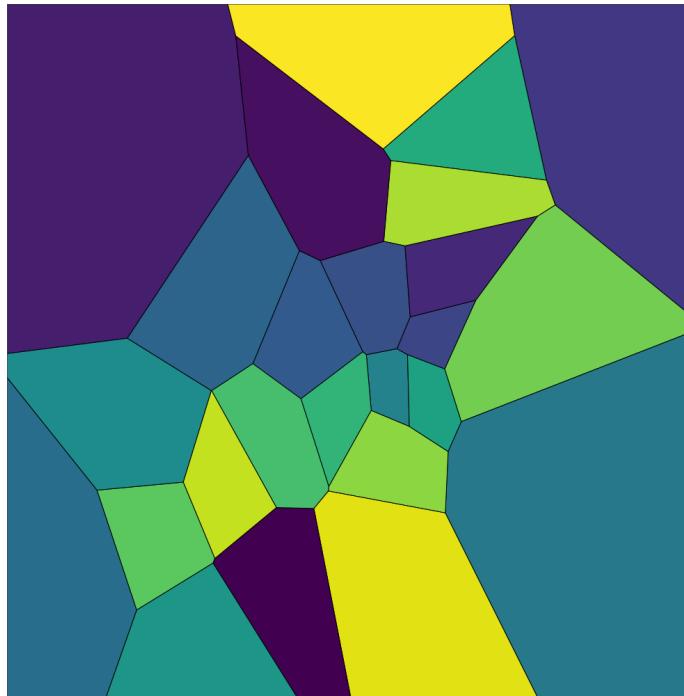
Voronoi Tessellations

- 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

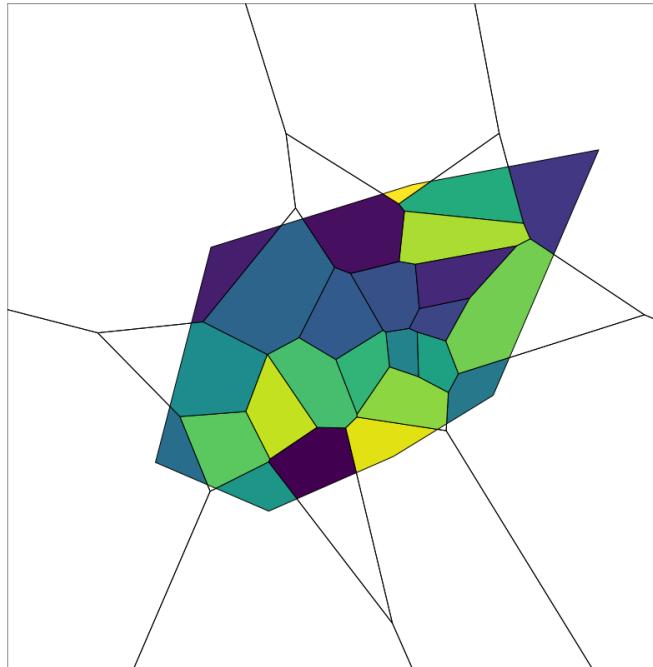
```
1 using DelaunayTriangulation, CairoMakie  
2 points = randn(2, 25)  
3 tri = triangulate(points)  
4 vorn = voronoi(tri)  
5 voronoiplot(vorn)
```



Clipped Voronoi Tessellations

- Support for clipping to convex polygons.

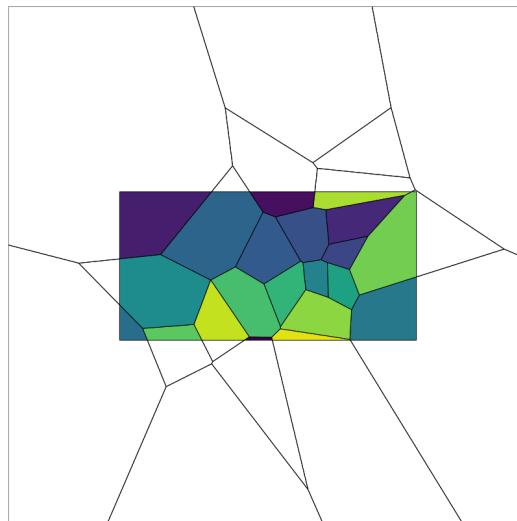
```
1 vorn = voronoi(tri; clip=true) # clips to the convex hull  
2 voronoiplot(vorn)
```



Clipped Voronoi Tessellations

- Support for clipping to convex polygons.

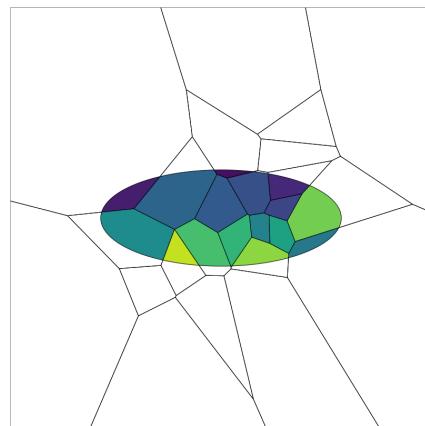
```
1 clip_points = [(-2.0, -1.0), (2.0, -1.0), (2.0, 1.0), (-2.0, 1.0)]
2 clip_vertices = [1, 2, 3, 4, 1]
3 vorn = voronoi(tri; clip=true,
4     clip_polygon=(clip_points, clip_vertices)) # clips to the square
5 voronoiplot(vorn)
```



Clipped Voronoi Tessellations

- Support for clipping to convex polygons.

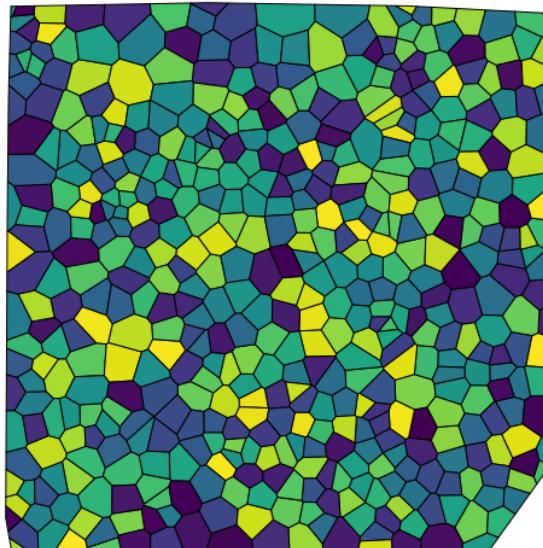
```
1 ellip = EllipticalArc((2.0, 0.0), (2.0, 0.0), (0.0, 0.0), 2.0, 0.8, 0.0)
2 t = LinRange(0, 1, 100)
3 clip_points = ellip.(t)[1:end-1]
4 clip_vertices = [1:length(clip_points); 1]
5 vorn = voronoi(tri; clip=true,
6     clip_polygon=(clip_points, clip_vertices)) # clips to the ellipse
7 voronoiplot(vorn)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



Centroidal Voronoi Tessellations

- Support for centroidal Voronoi tessellations.

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



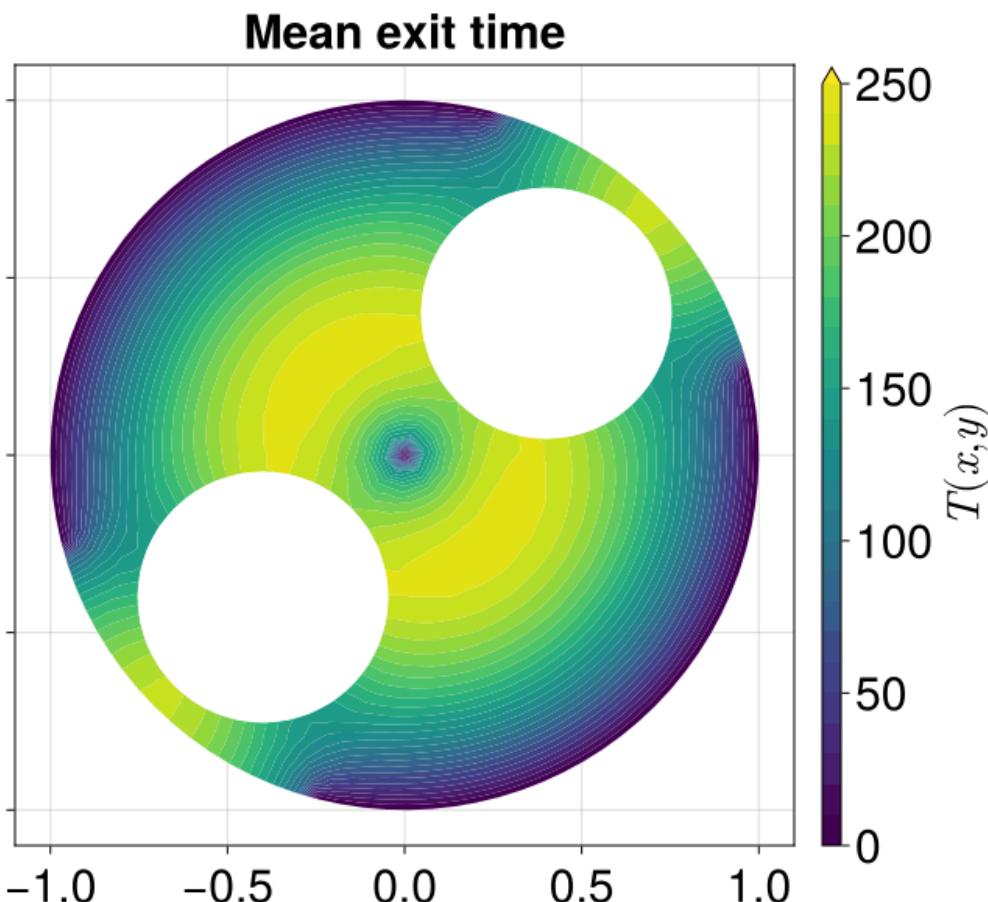
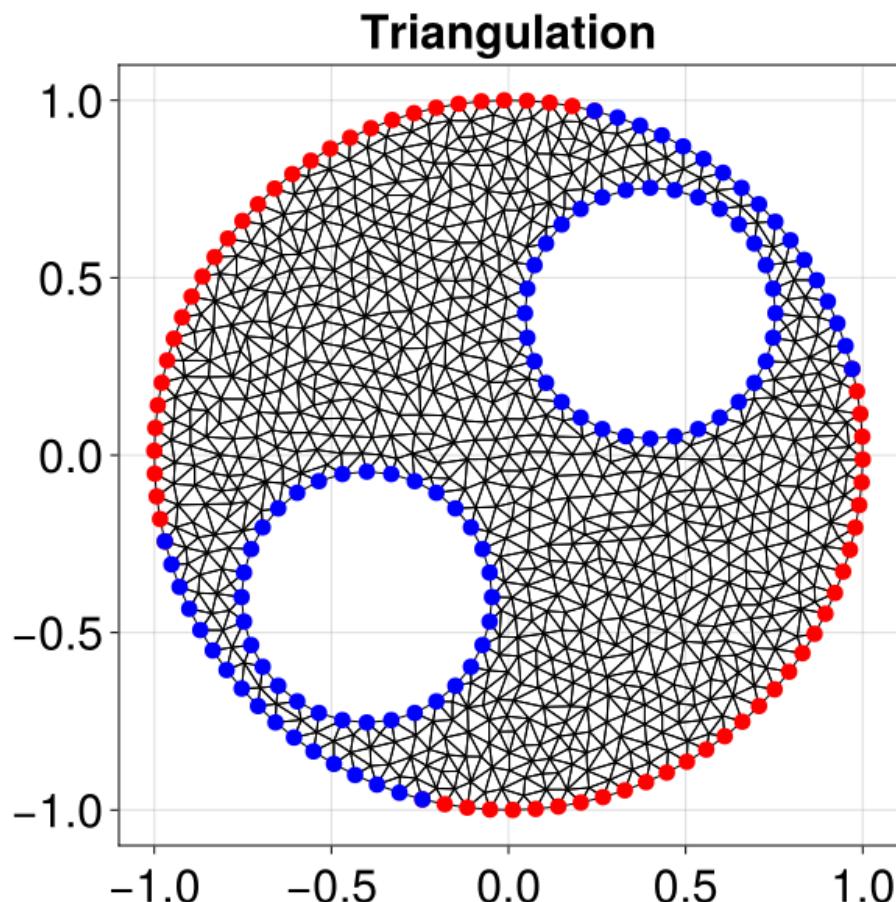
FiniteVolumeMethod.jl

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

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

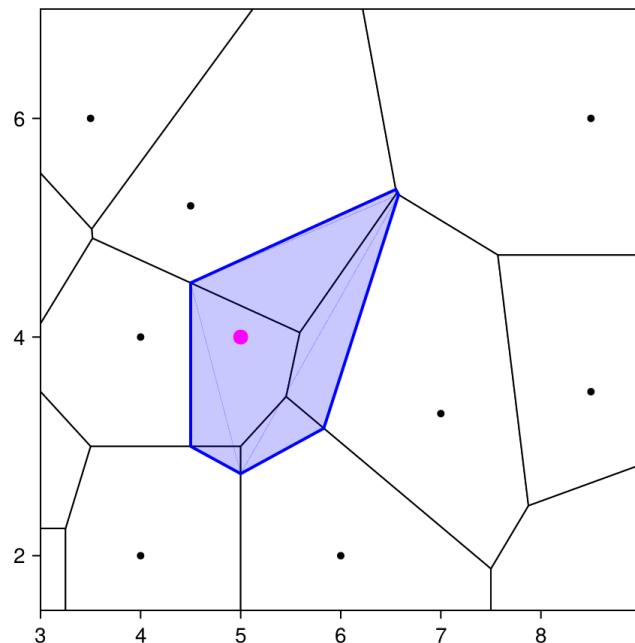
- Hooks into the SciML interface to support a wide range of solvers.

FiniteVolumeMethod.jl



NaturalNeighbours.jl

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



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.