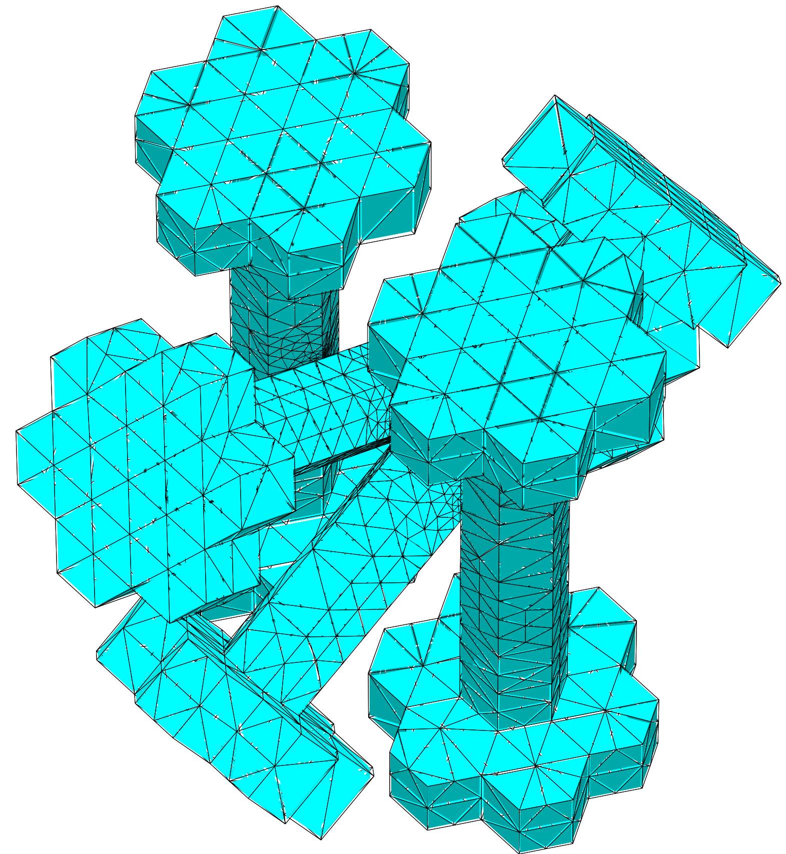
Dynamic Mesh Refinement

Benoît Hudson

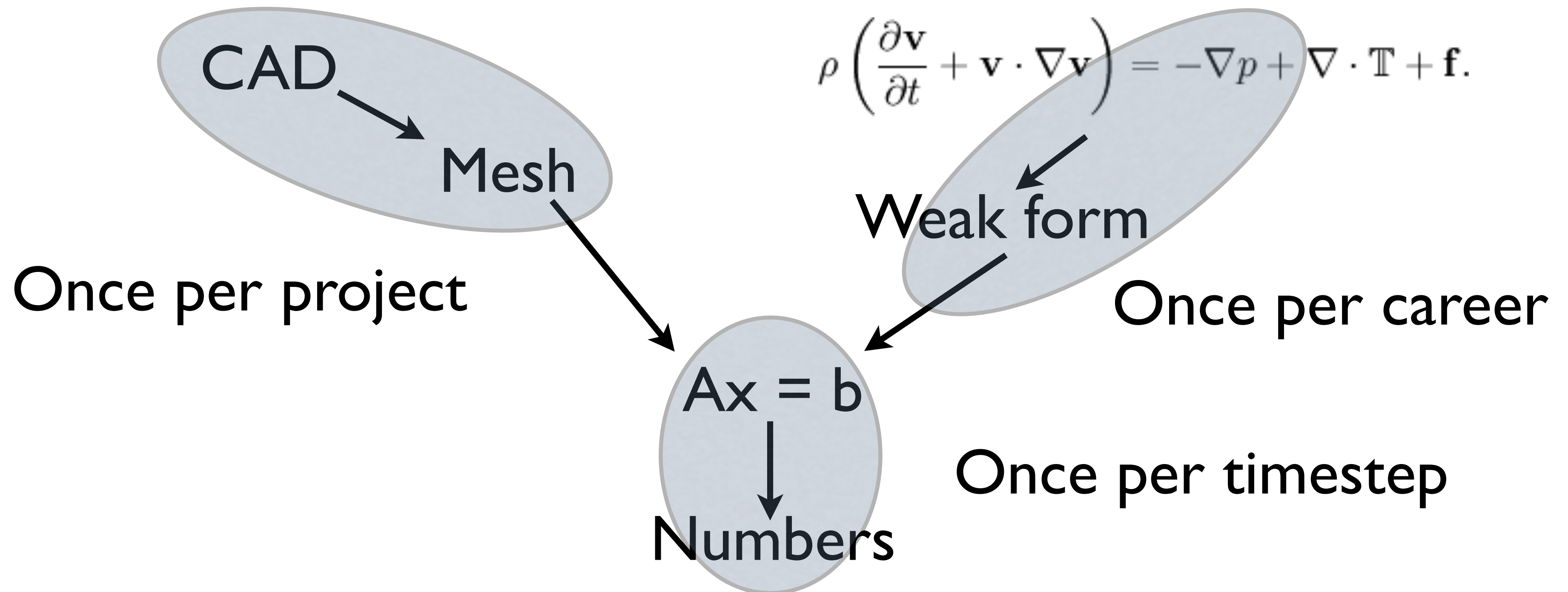
Joint work with Umut A. Acar, Gary Miller, Todd Phillips

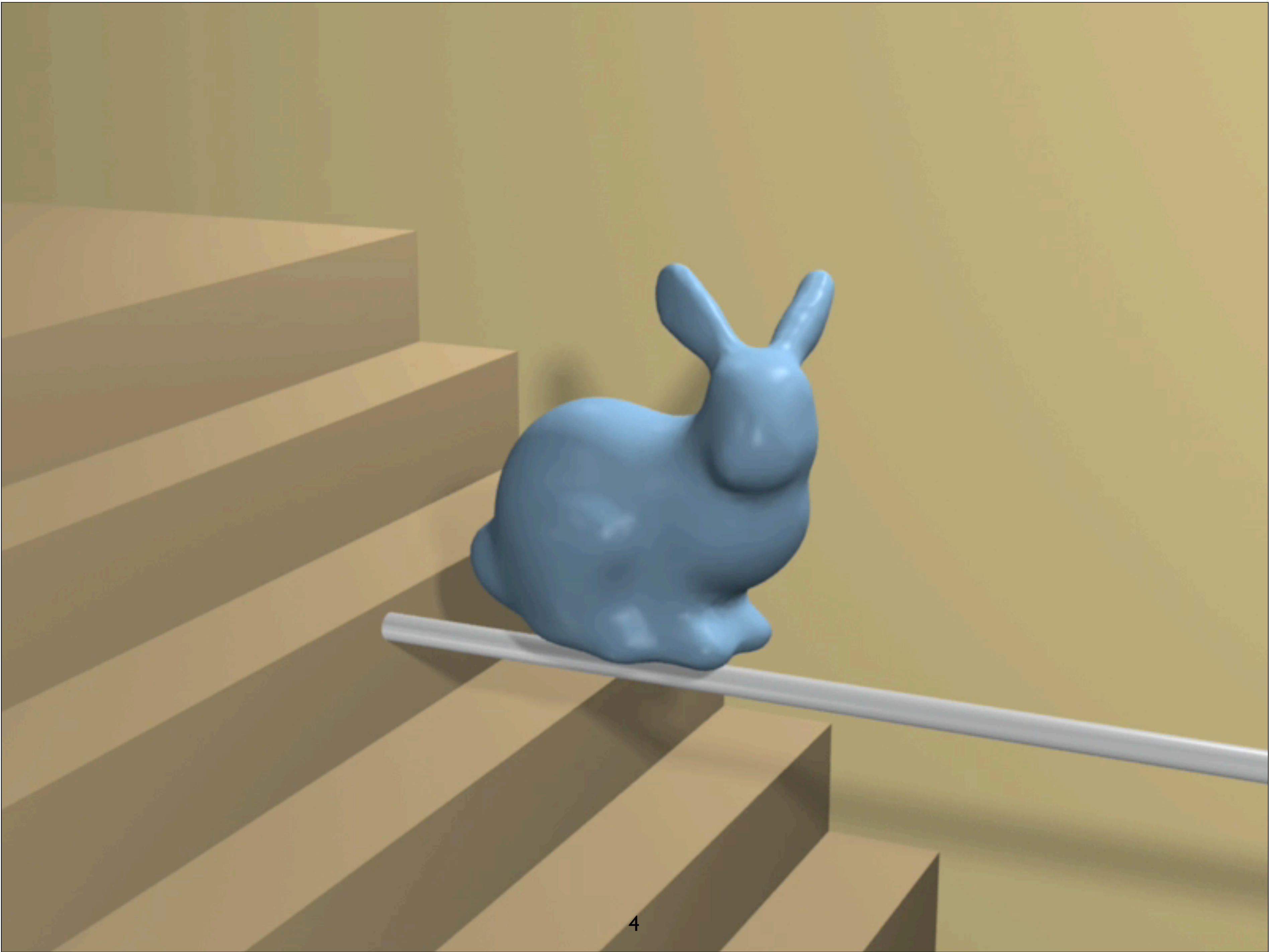


Why Mesh?

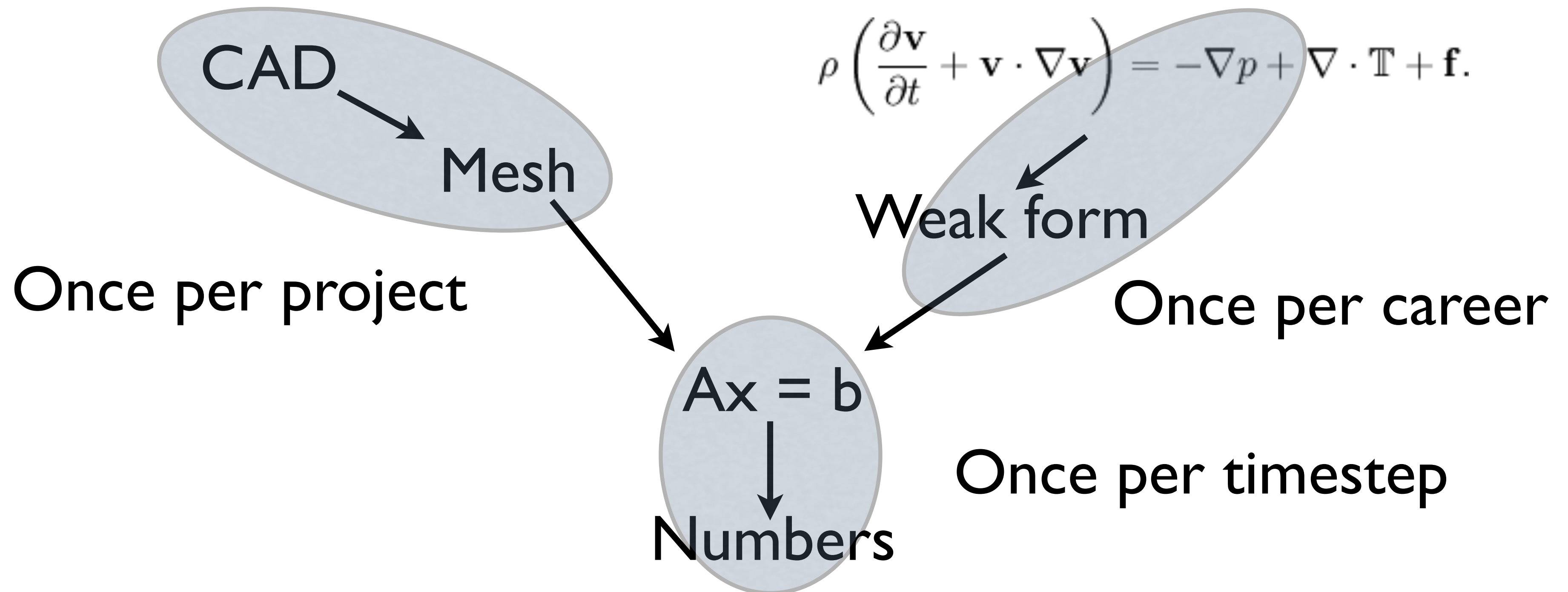
- Geometry is continuous
- Computers are discrete
- Mesh for any geometric processing:
  - Graphics
  - Scientific Computing
  - Vision, AI, ...

Scientific computing: historically

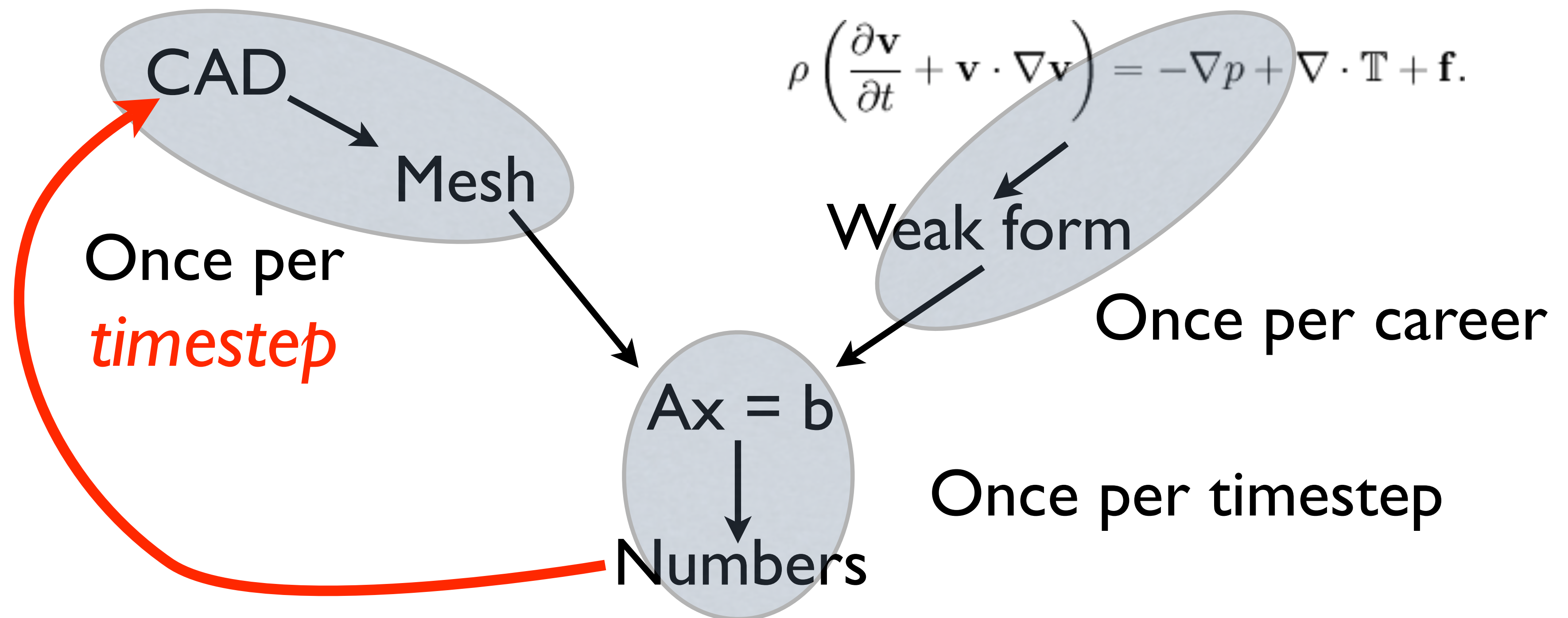




Scientific computing: historically



Scientific computing: modern



Claims of the Thesis

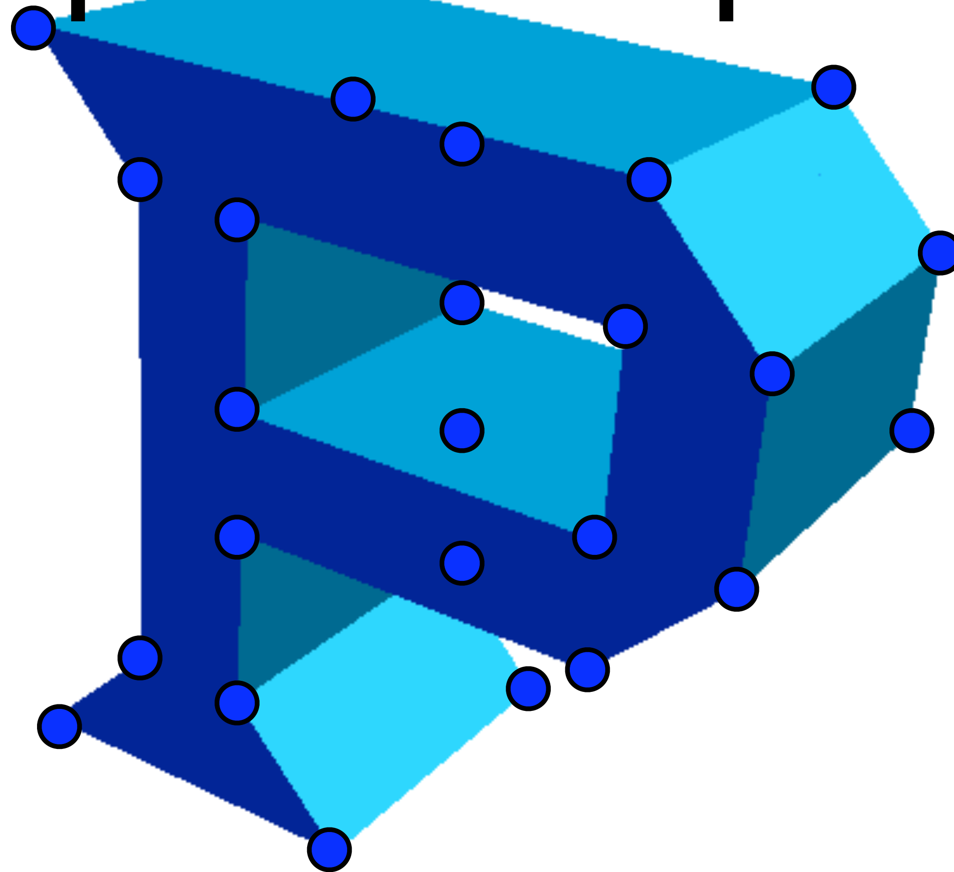
- The fastest static meshing code
- A framework with lots of explicit freedom for point placement
- The first dynamic meshing algorithm

Outline in four parts

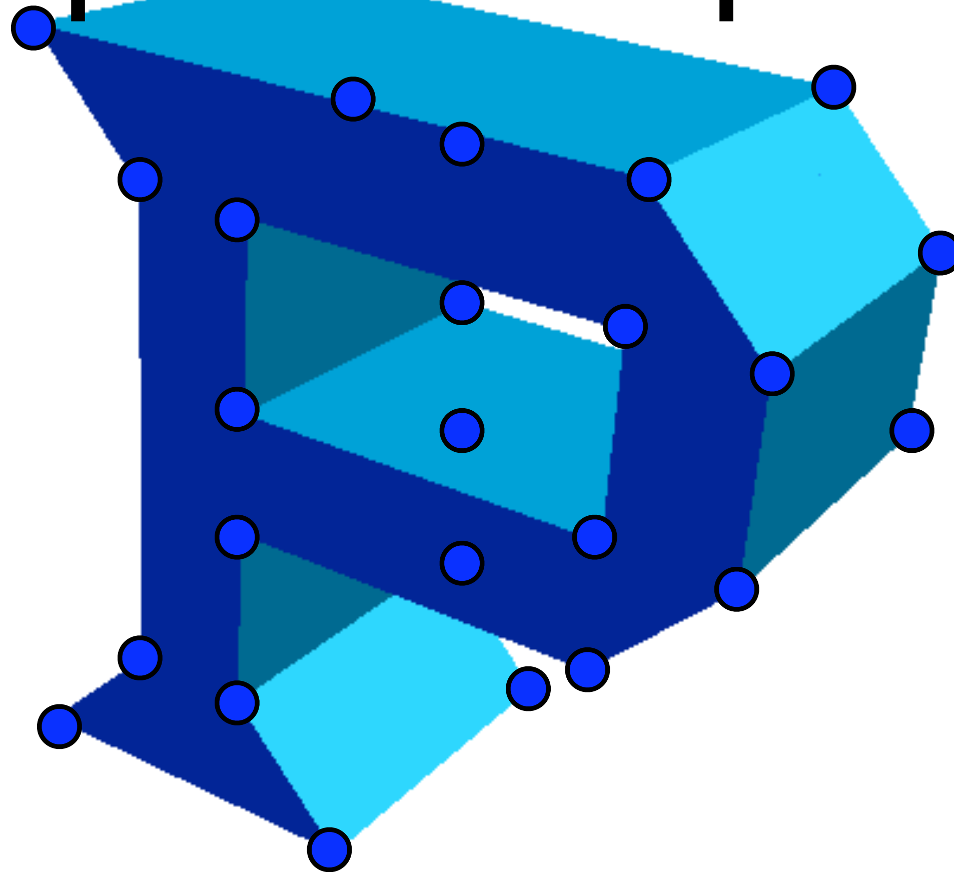
Goal of the talk:
Enough pictures to understand main points
Enough main points to understand thesis

- What makes a good mesh?
- Experimental results
- General conceptual algorithm
- Sub-linear dynamic mesh refinement

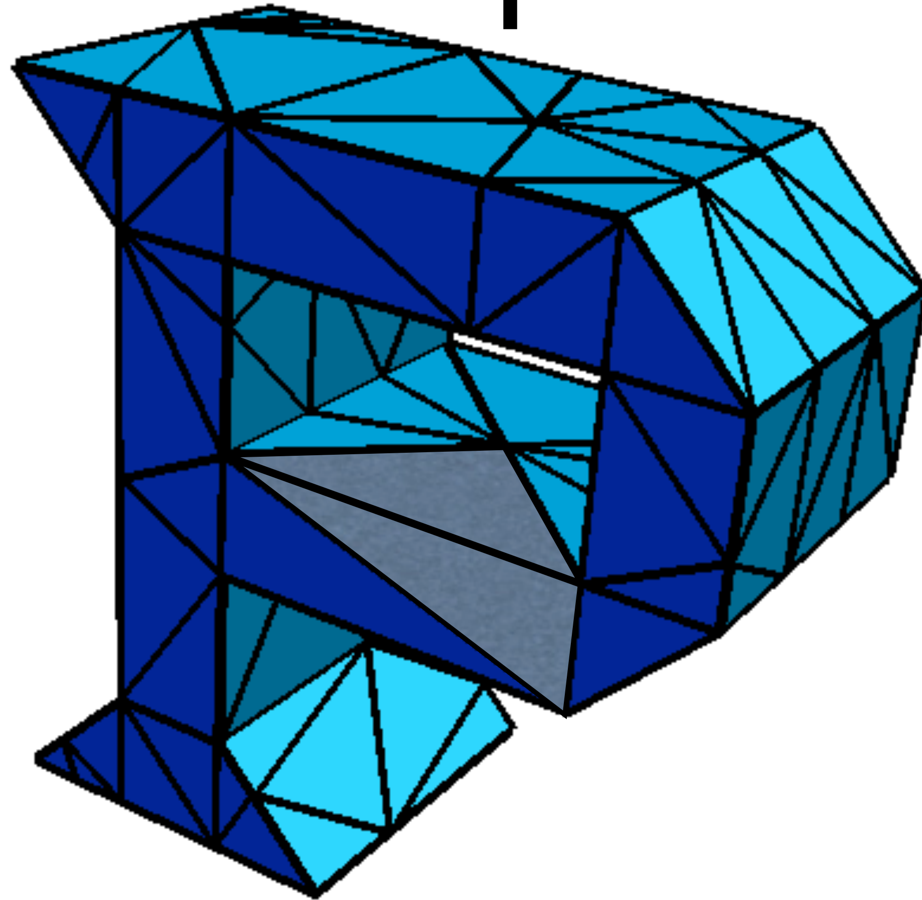
Input Description



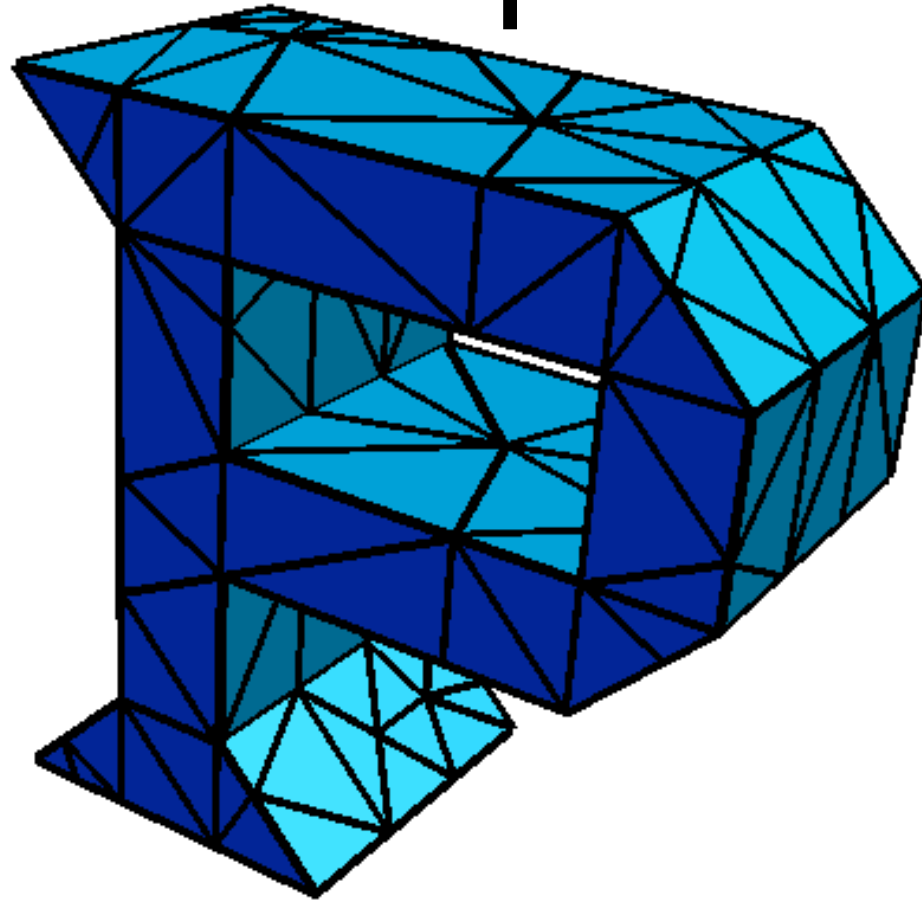
Input Description



Output

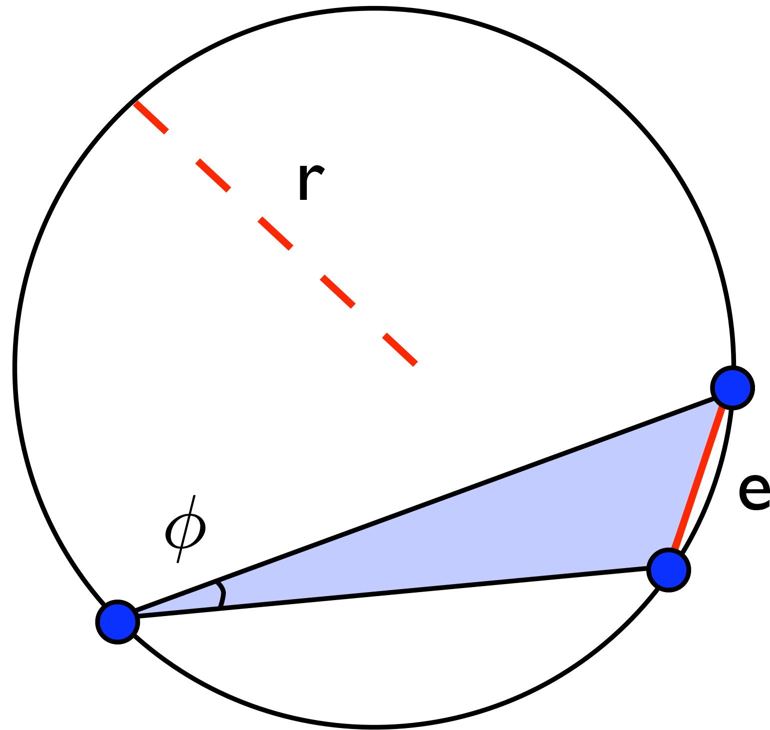


Output



Quality measure: radius/edge

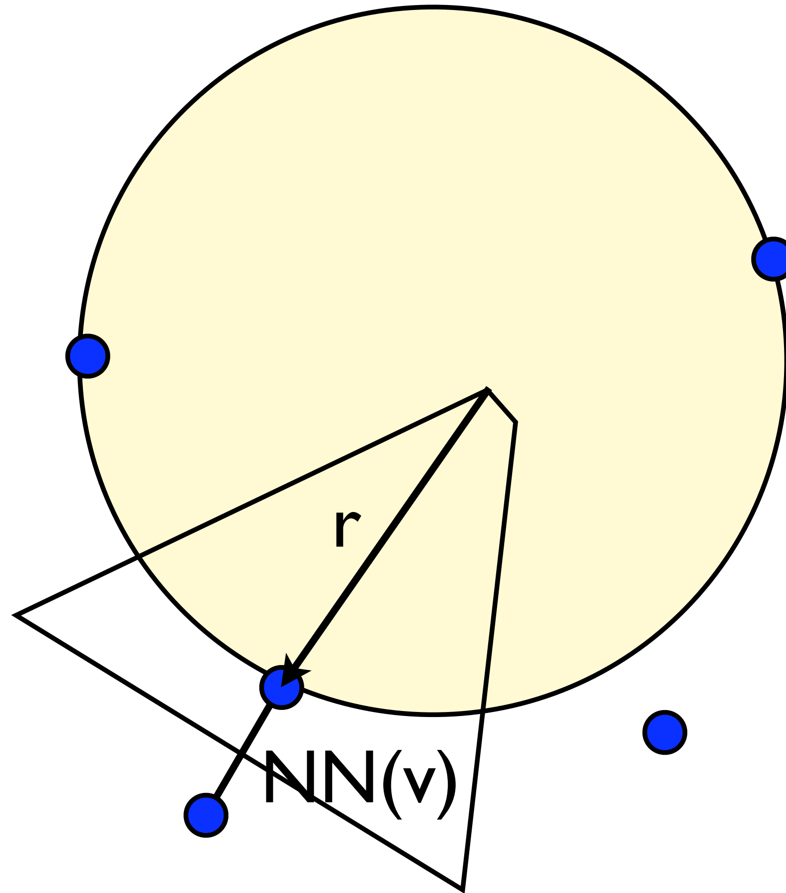
$r/e = 1/2\sin(\phi)$



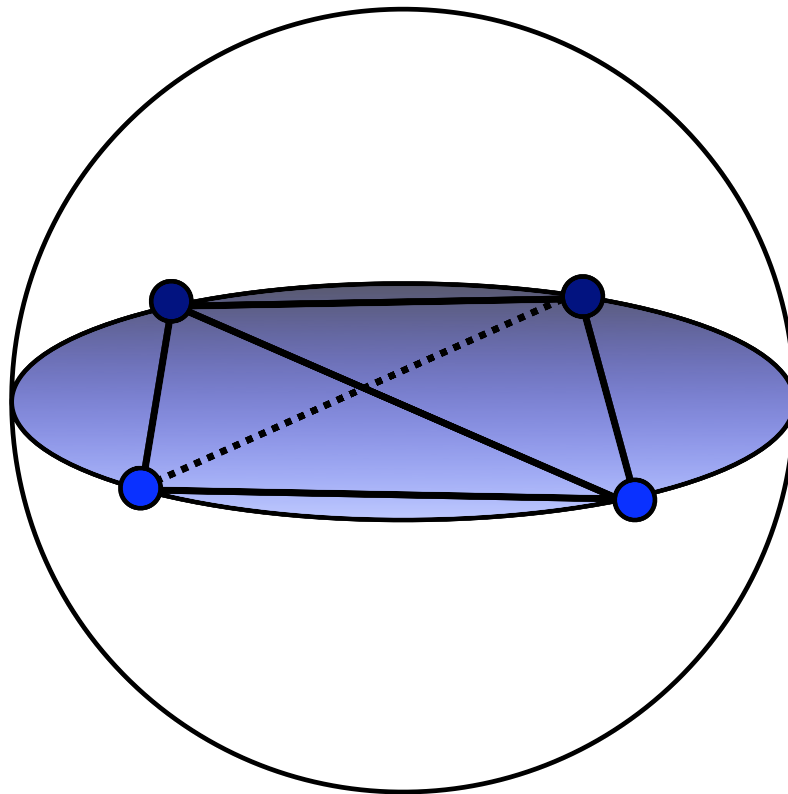
$\phi > \alpha$
$r/e < \rho$

Voronoi aspect ratio

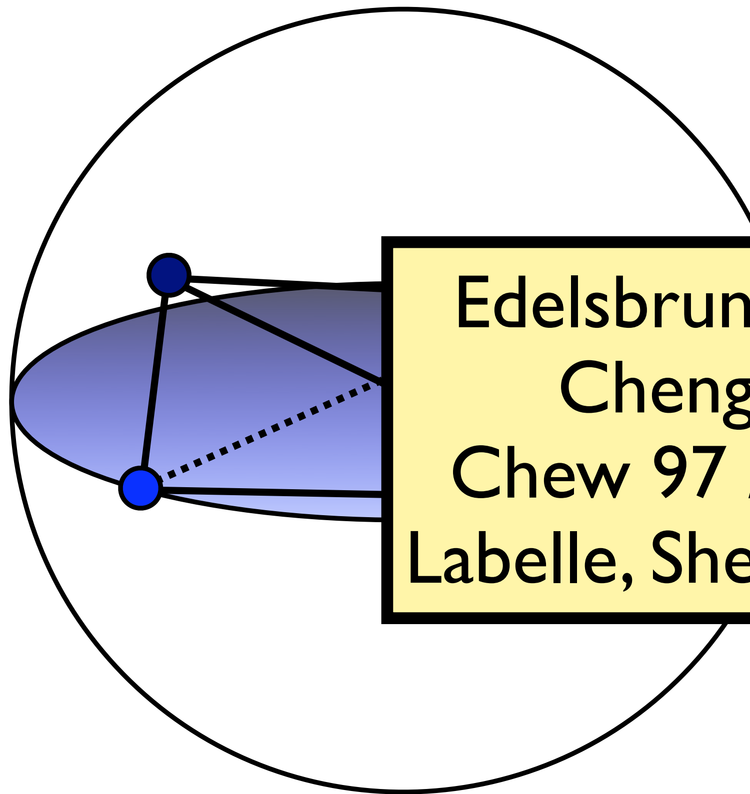
if r/NN(v) < ρ
then r/e < ρ



3d: Slivers



3d: Slivers



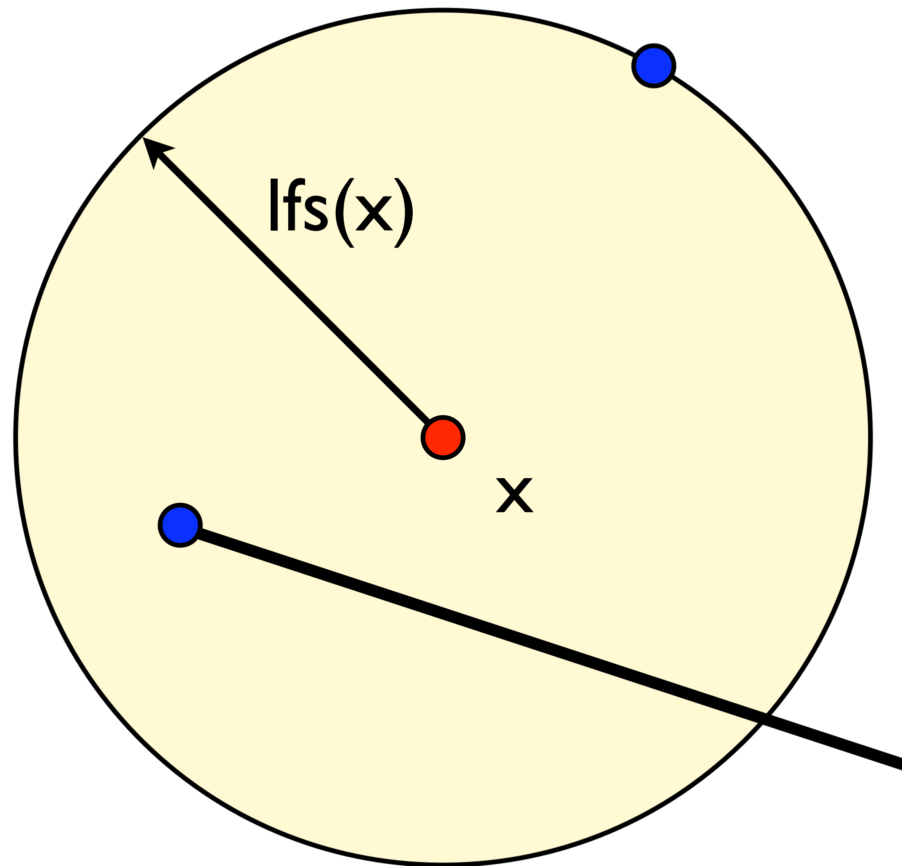
0° angles
$r/e = 1/\sqrt{2}$

Edelsbrunner *et al* 00
Cheng *et al* 00
Chew 97 / Li-Teng 01
Labelle, Shewchuk 06/07

Simulation Runtime

- Runtime: O(# triangles) or O(# triangles$^{3/2}$)
  - Don't create too many elements
- Timestep length: O(shortest edge)
  - Don't create tiny elements

local feature size



lfs and mesh size

- NN(v) $\in \Omega$(lfs(v)) at every vertex v:

  Size-conforming

- Then #vertices $\in O(\int \mathrm{lfs}^{-d}(x)dx)$
- Any no-small-angle mesh is $\Omega(\int \mathrm{lfs}^{-d}(x)dx)$

Requirements

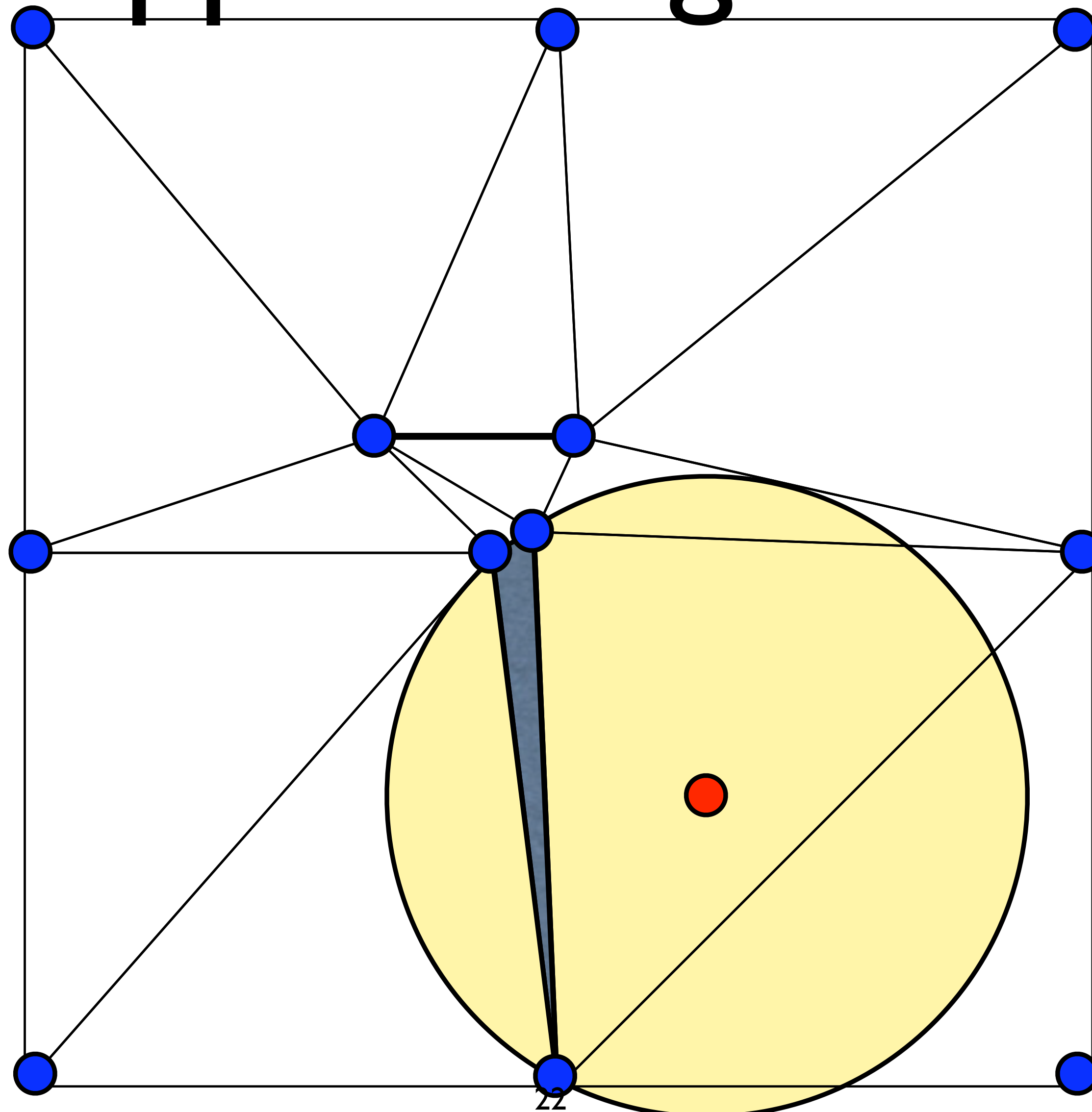
- *Quality*: no b… …ut
- *Respect*: t… …he output
- *Size-conforming*: spacing in output ~ in input

Fast

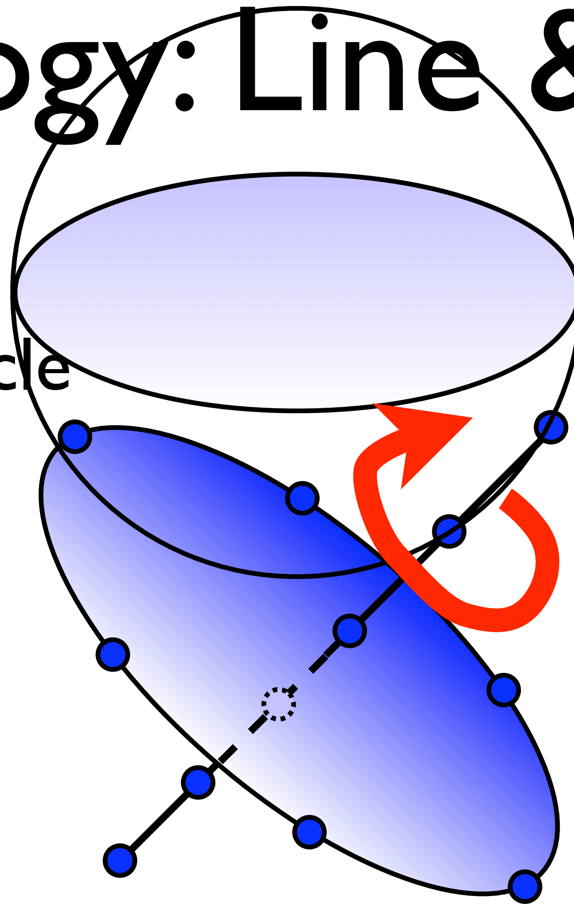
Outline in four parts

- What makes a good mesh?
- **Experimental results**
- General conceptual algorithm
- Sub-linear dynamic mesh refinement

Ruppert's algorithm

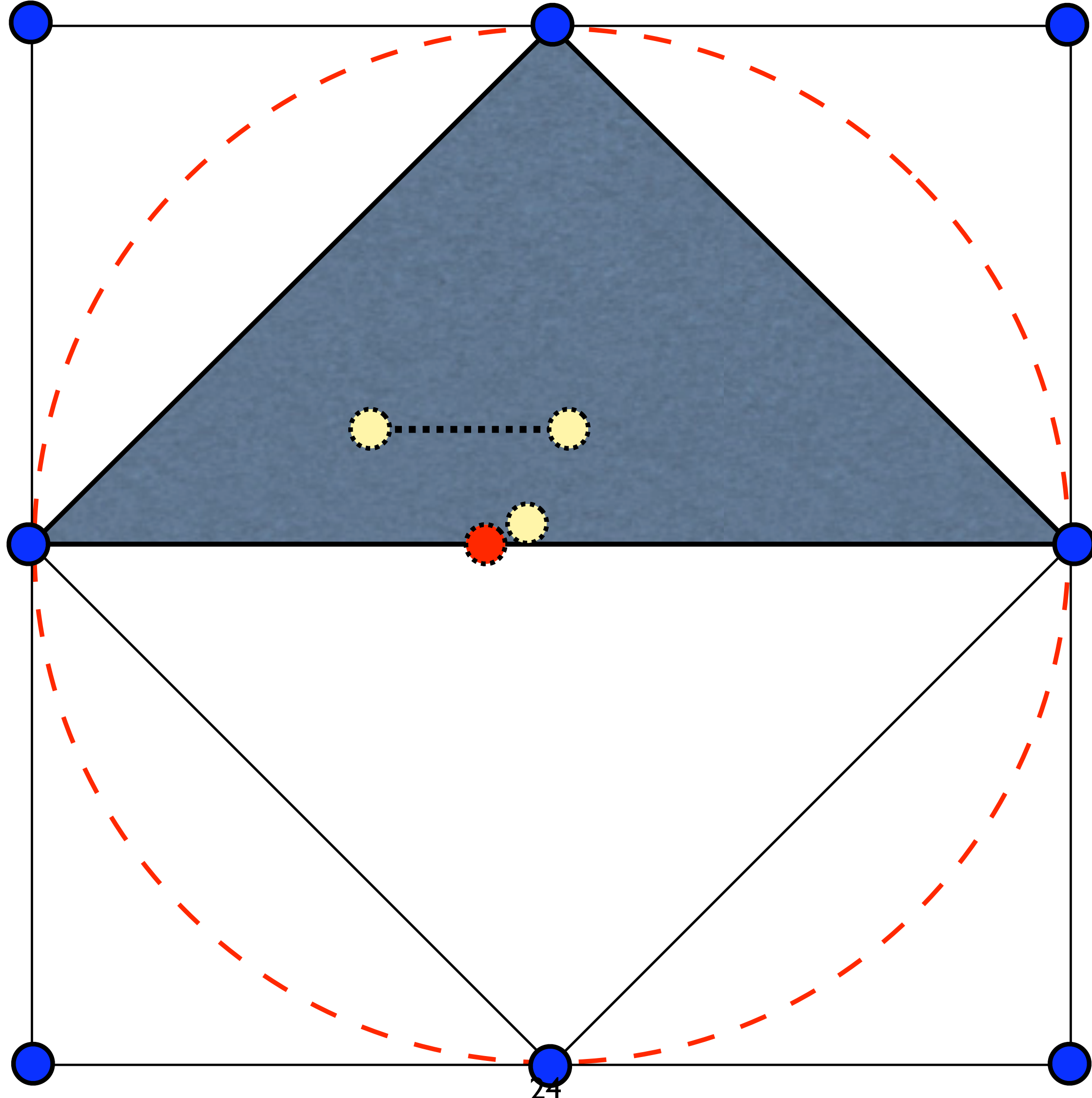


Pathology: Line & Circle

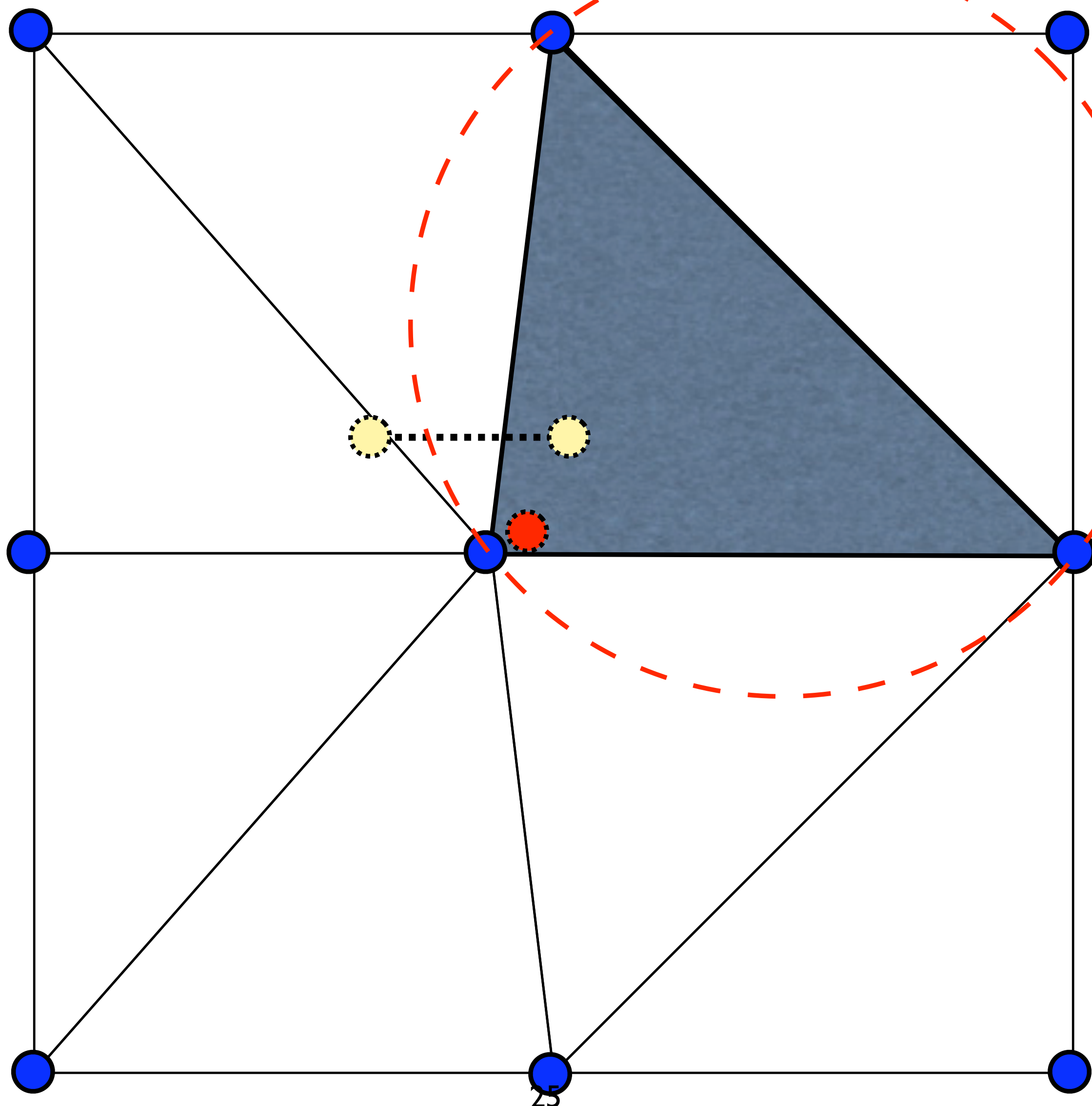


n^2 Delaunay triangles!

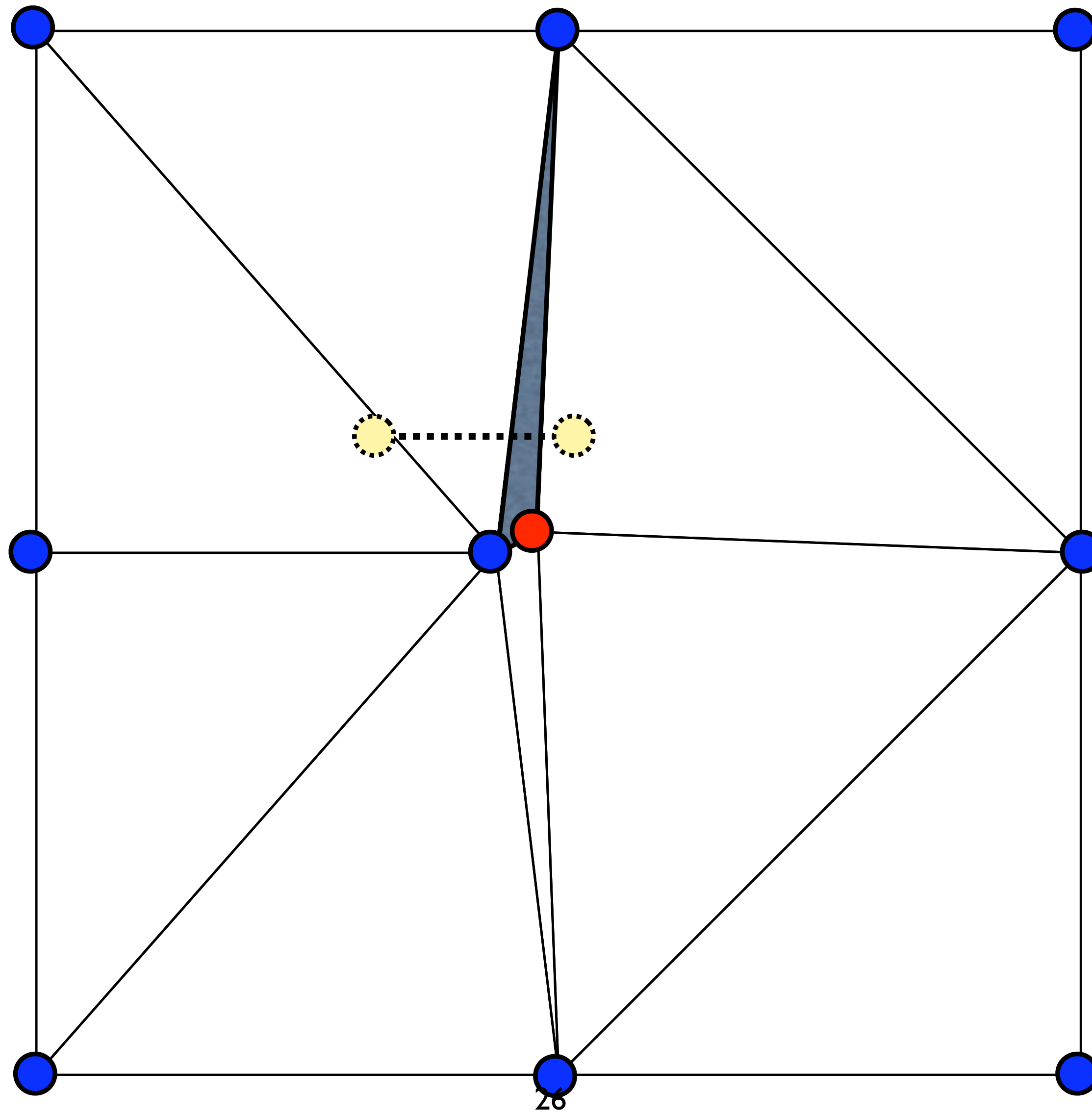
SVR



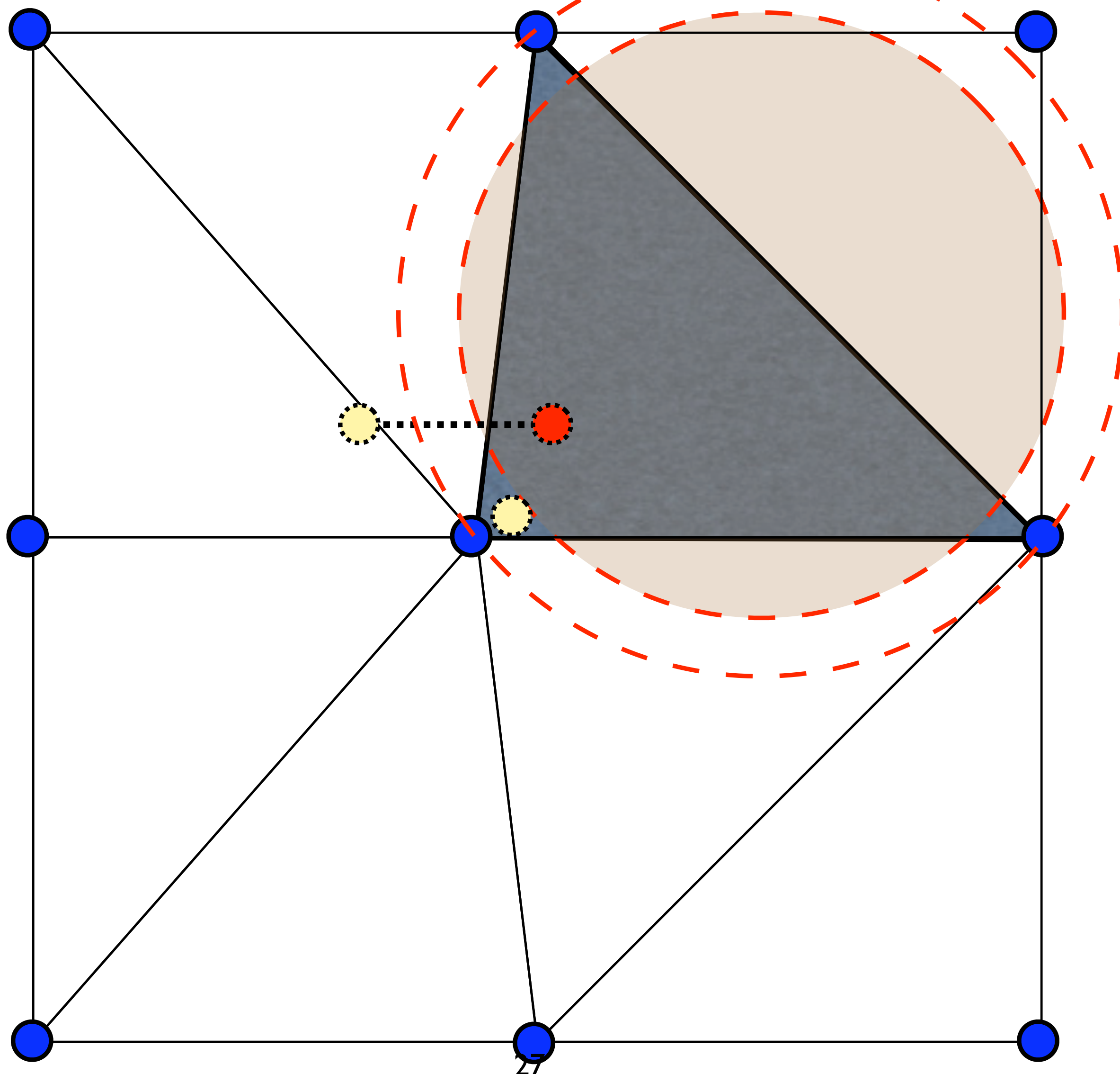
SVR



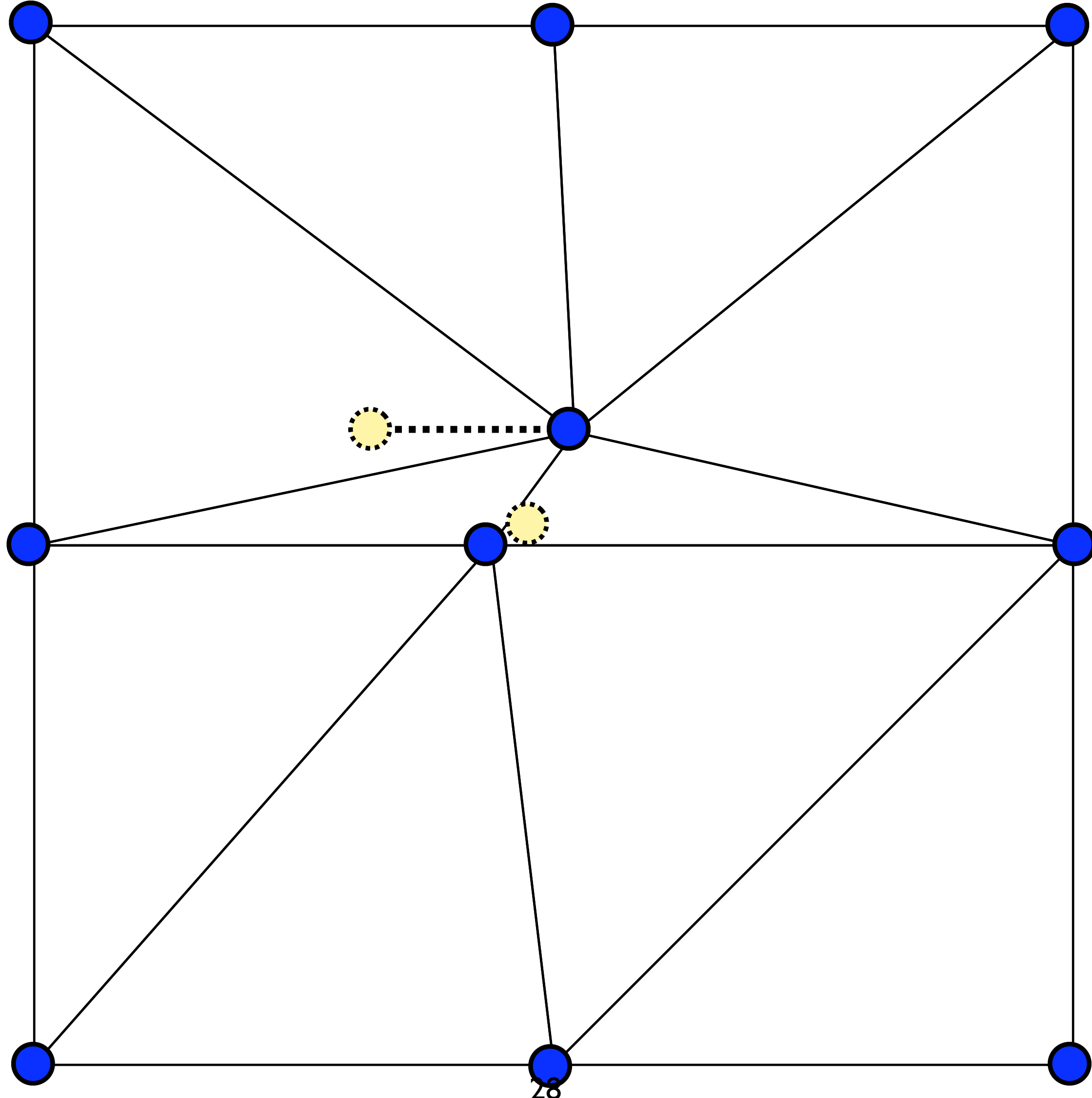
SVR



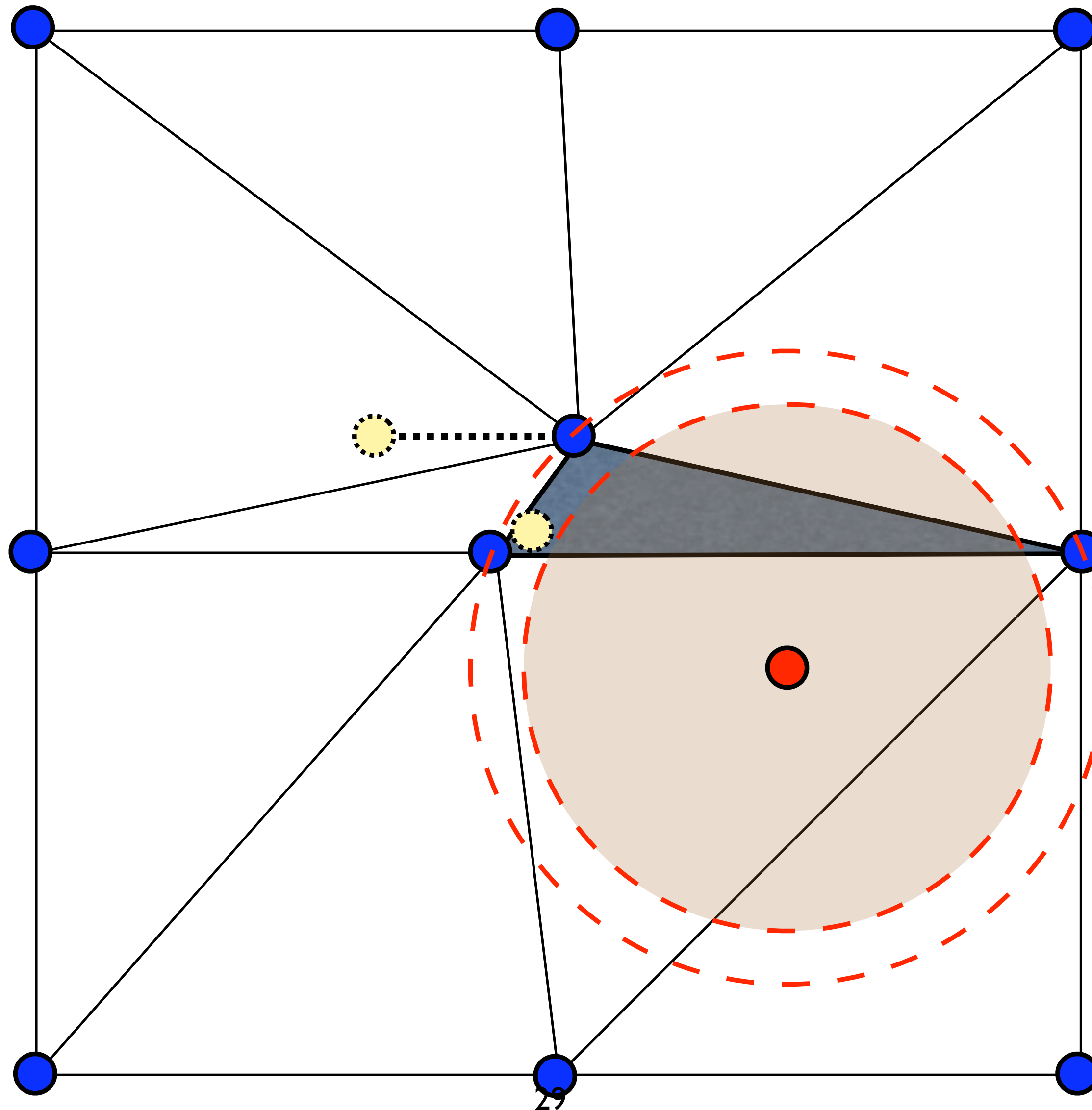
SVR



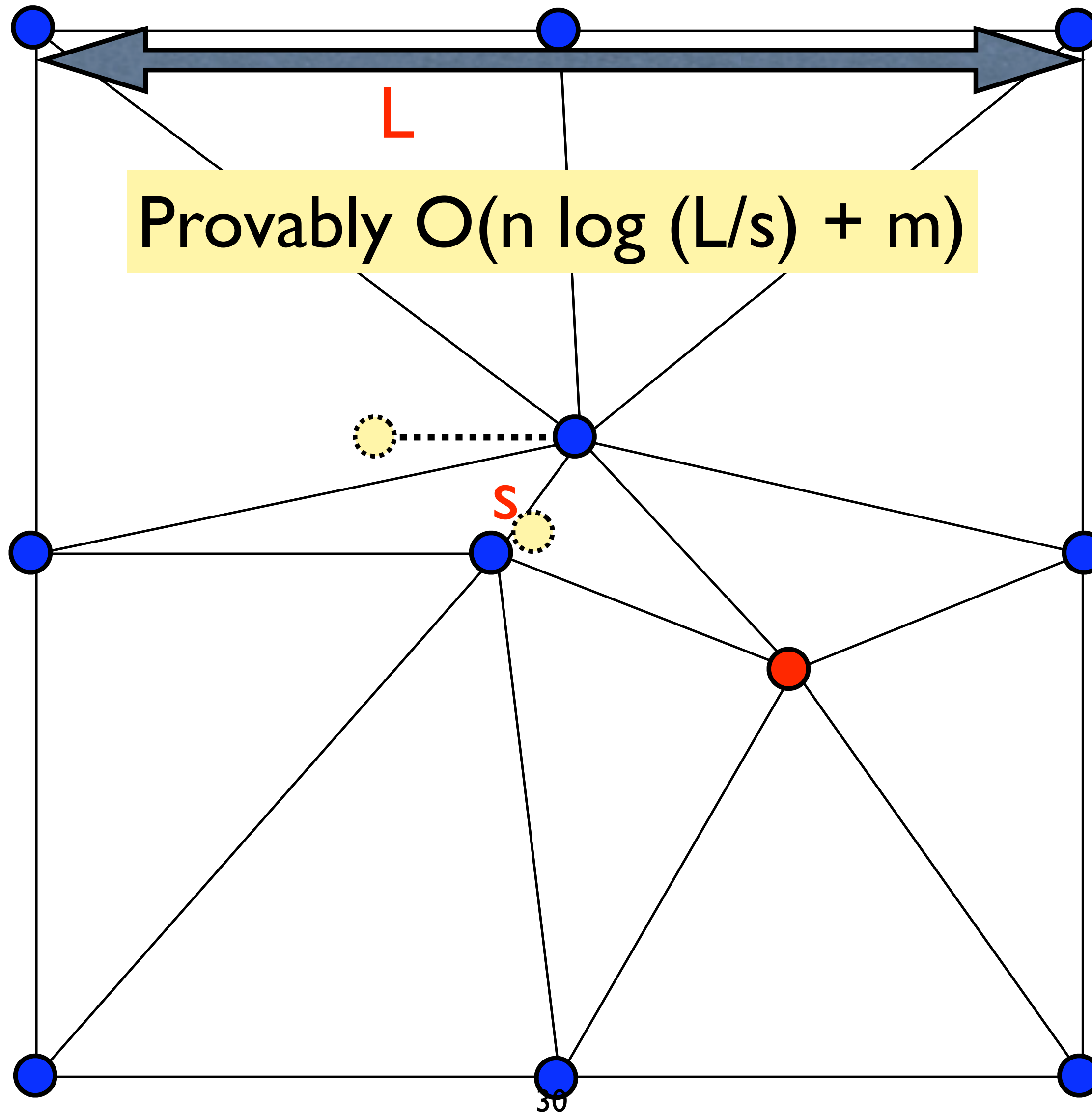
SVR



SVR



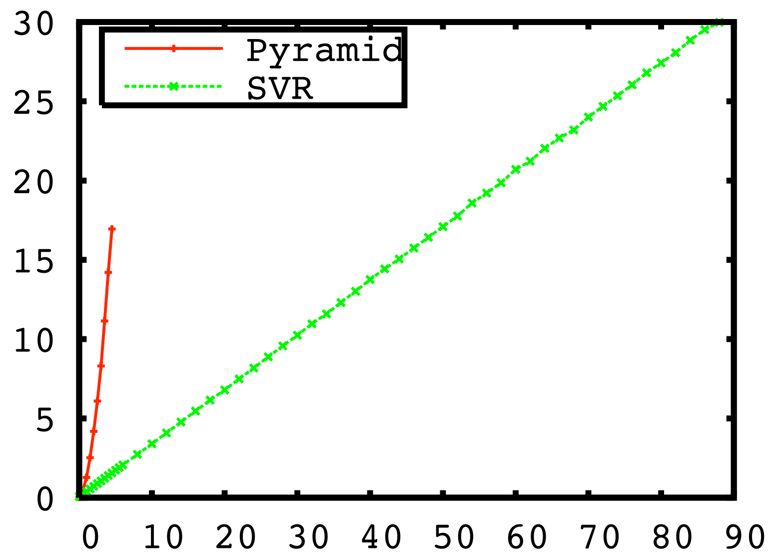
SVR

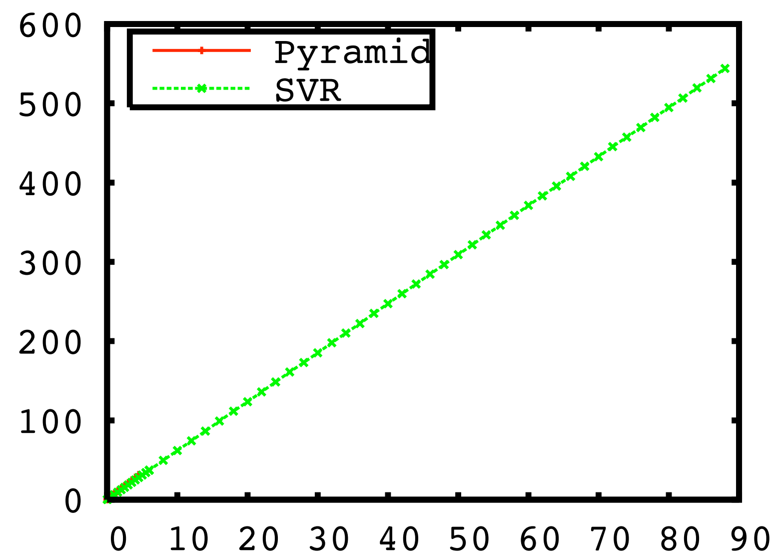


Comparative Results: Line & Circle

seconds

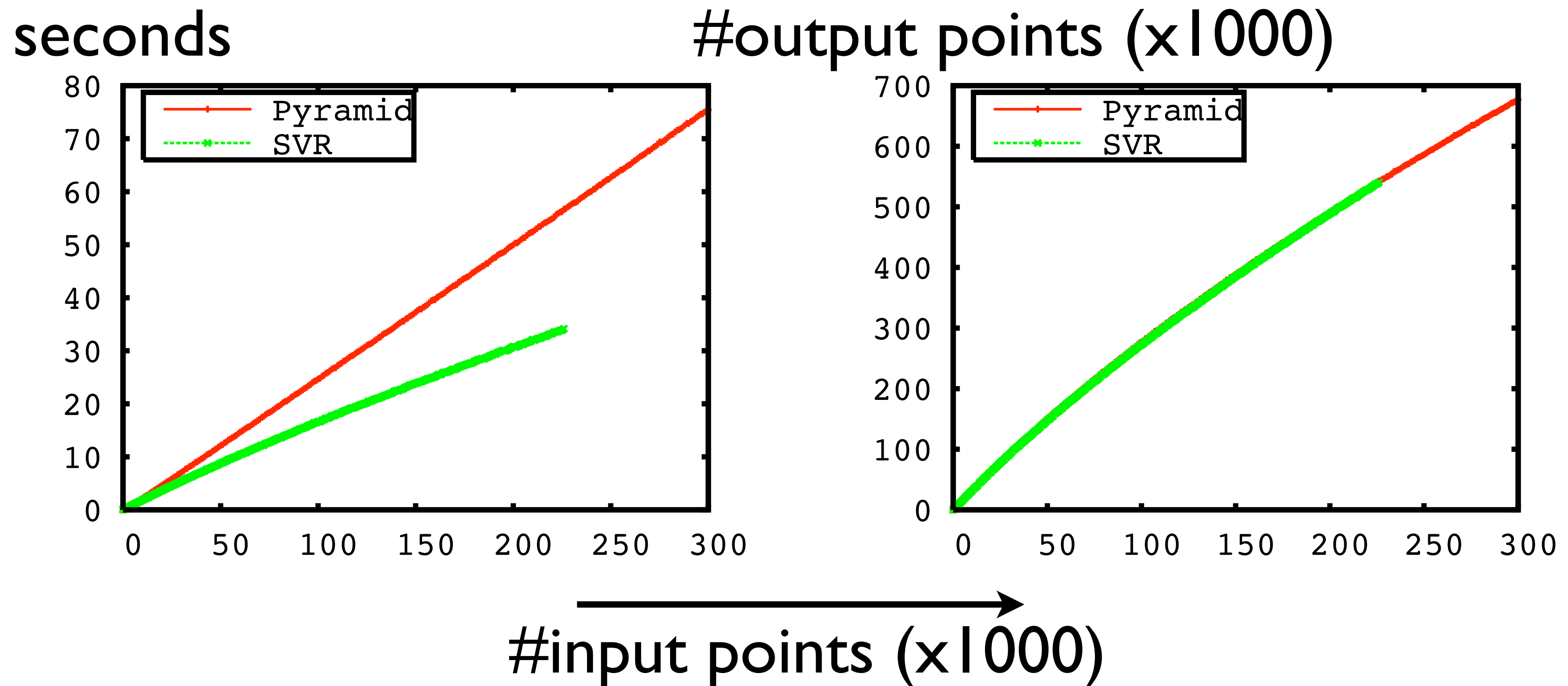
#output points (x1000)





#input points (x1000)

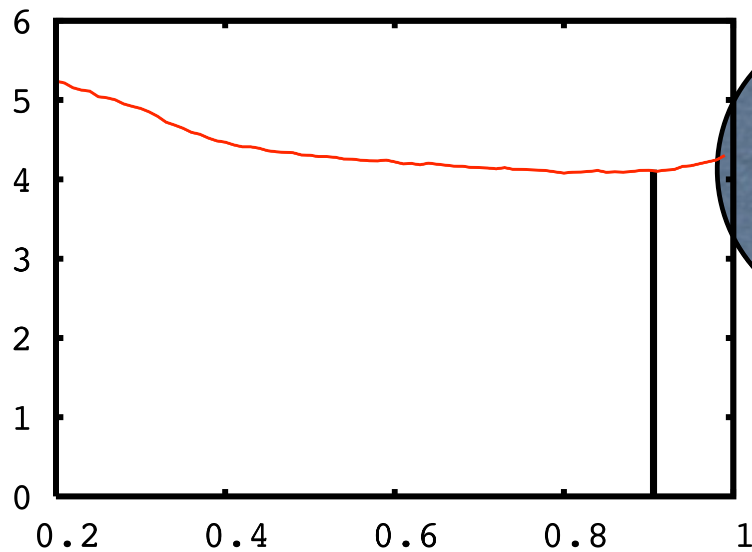
Comparative Results: 27 Stanford Bunnies

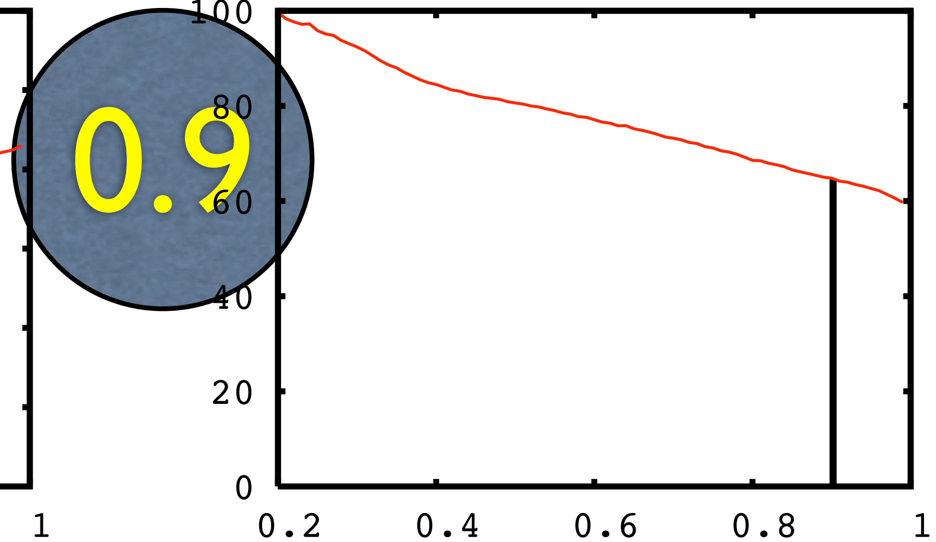


How much to shrink?

seconds

#output points (x1000)

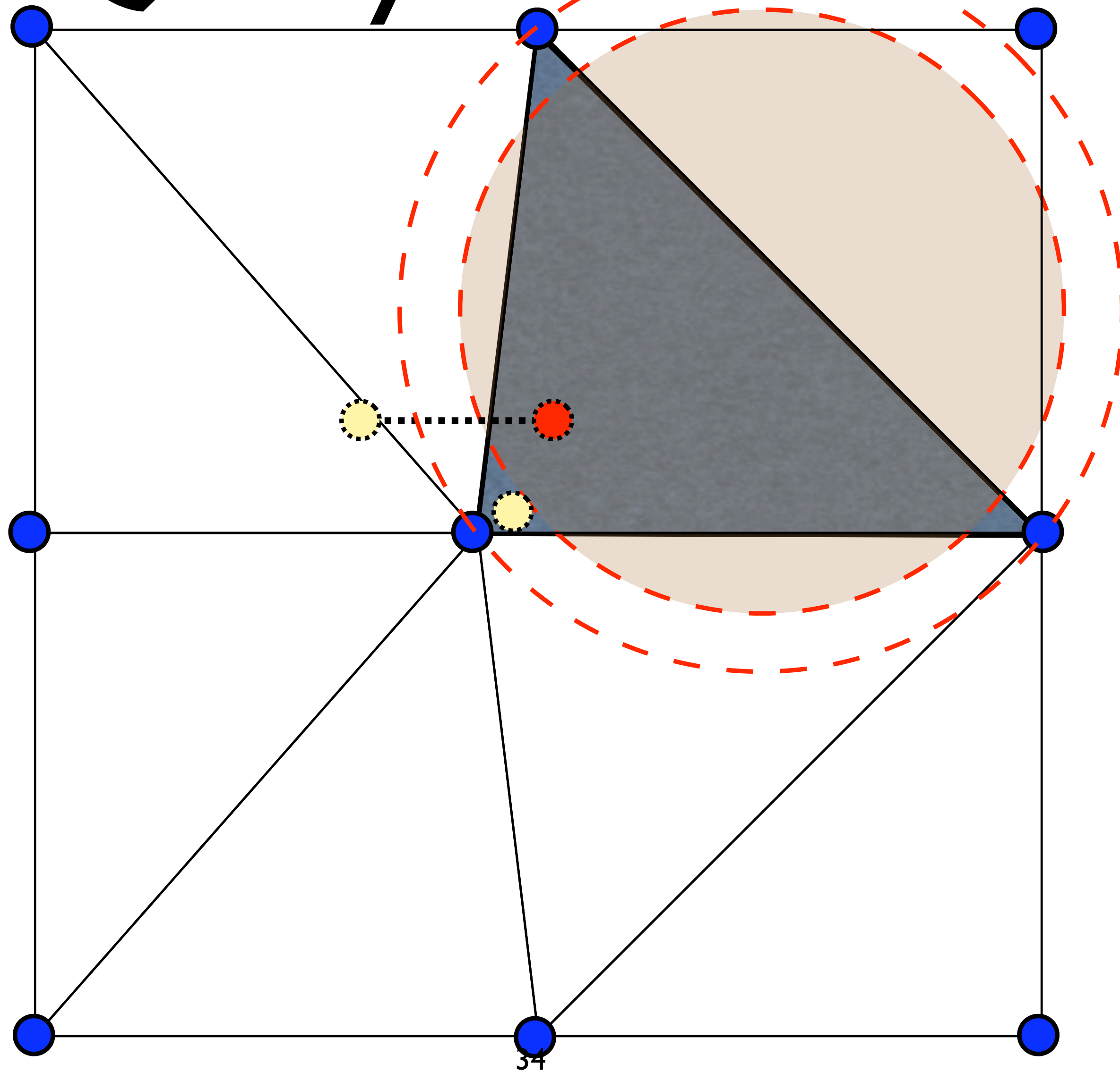




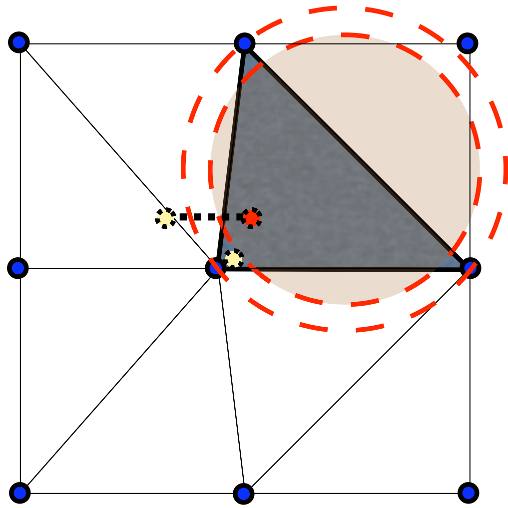
shrink to

shrink to

Query Structure

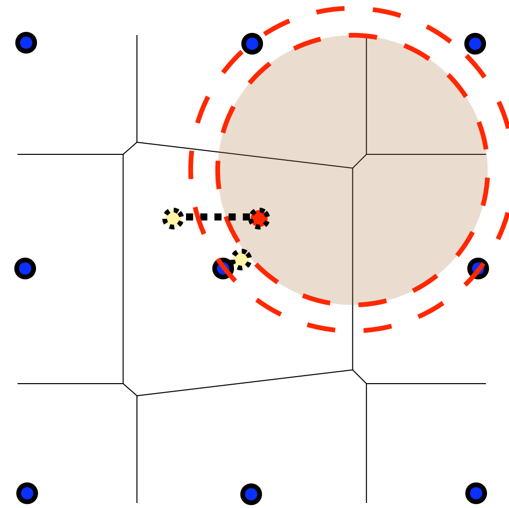


Query Structure



By simplex:

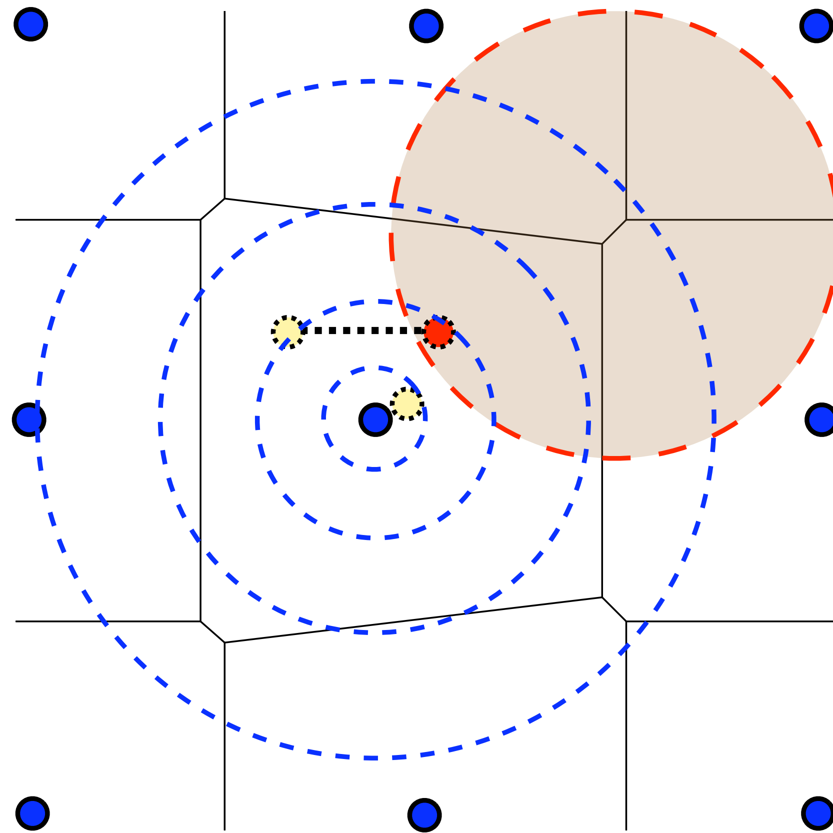
+ easy, traditional
+ cheap query
- expensive update
- non-robust

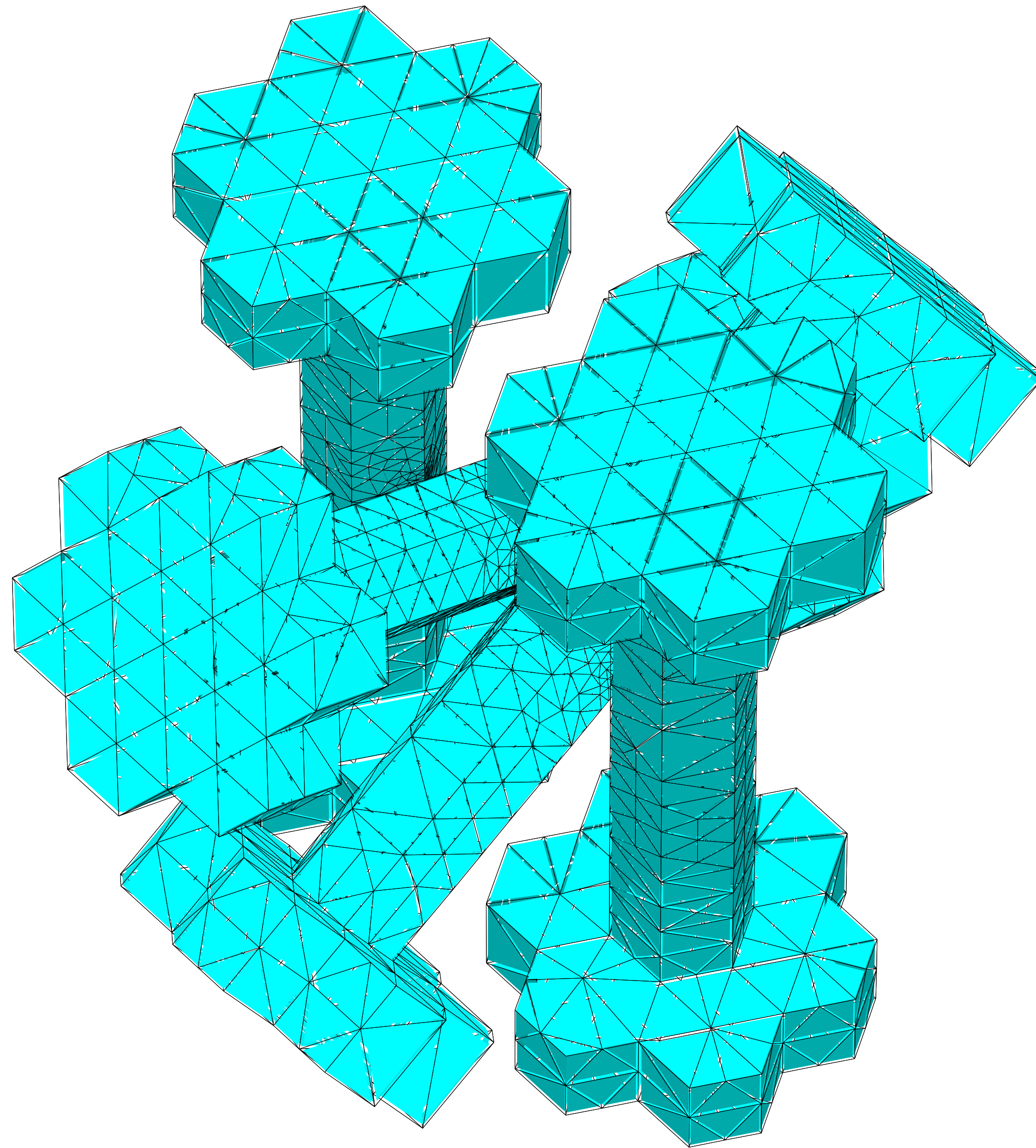


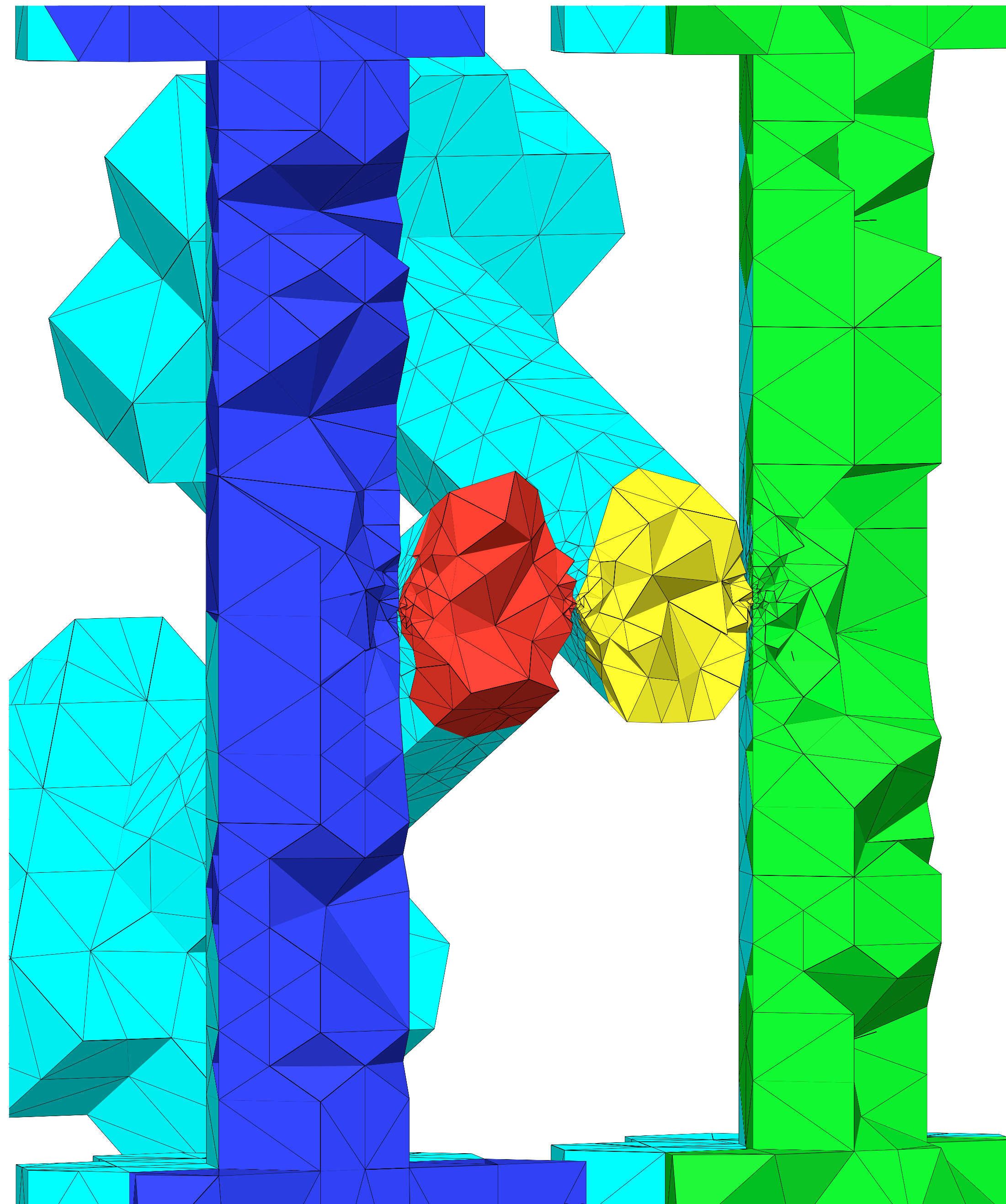
By Voronoi:

+ easy
+ fast update
+ robust
- bigger query

Speeding up queries







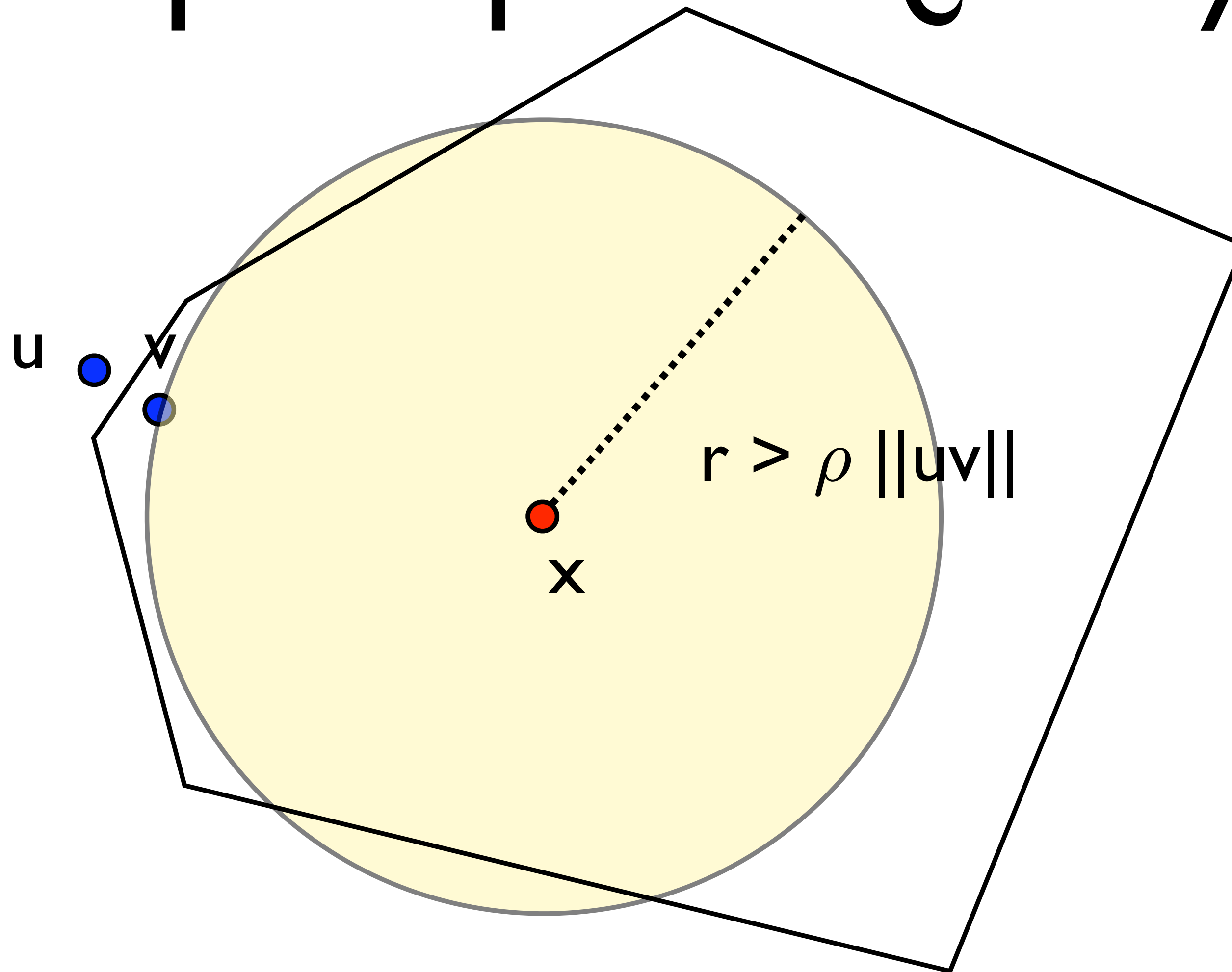
Outline in four parts

- What makes a good mesh?
- Experimental results
- **General conceptual algorithm**
- Sub-linear dynamic mesh refinement

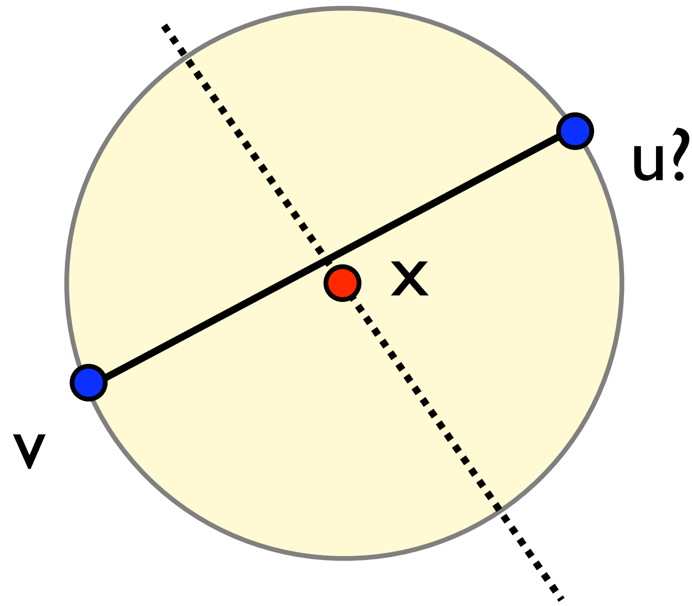
What makes a good Steiner point?

- Helps achieve *Quality*
- Helps achieve *Respect*
- Doesn't violate *Size-conforming*

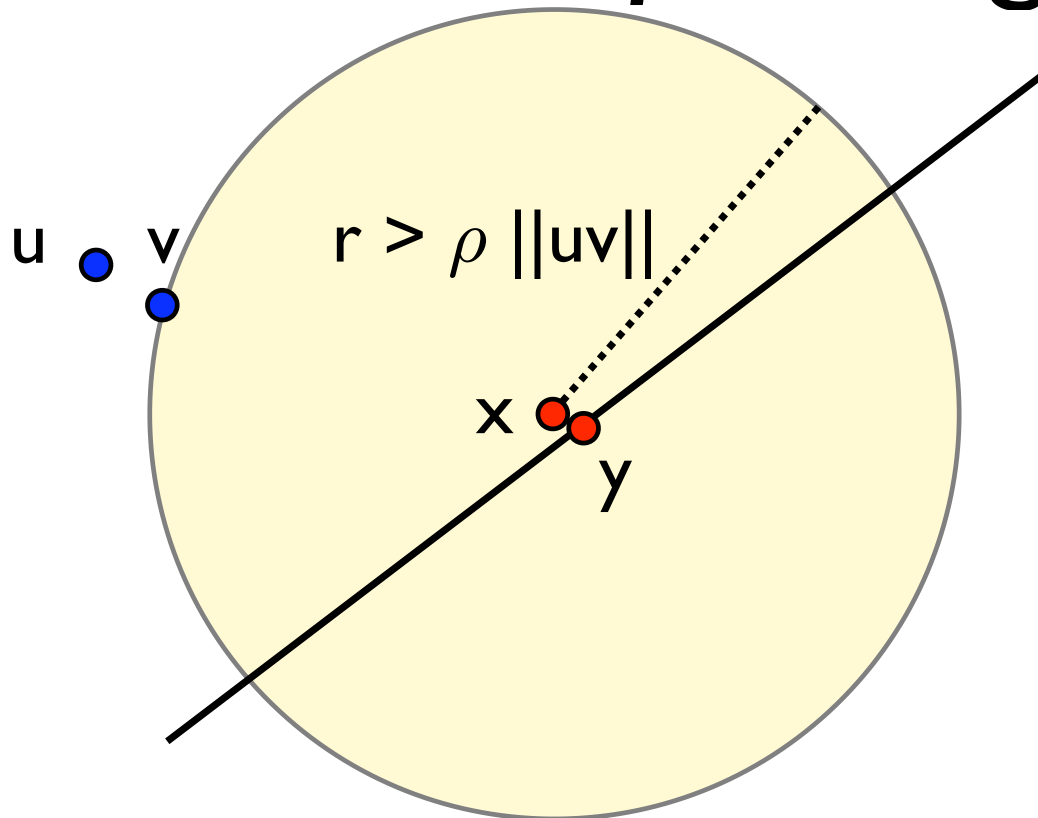
Helps improve *Quality*



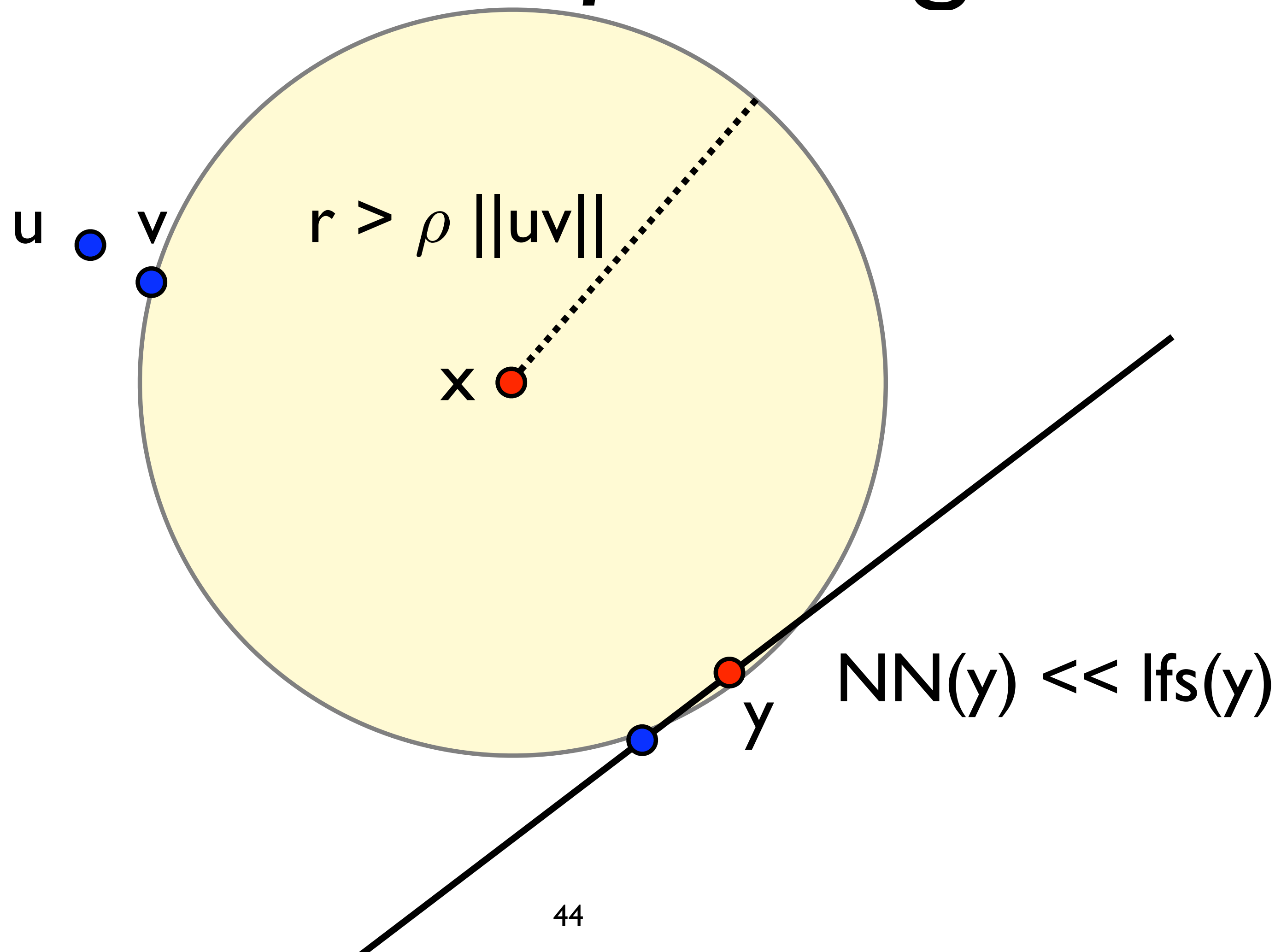
Helps improve *Respect*



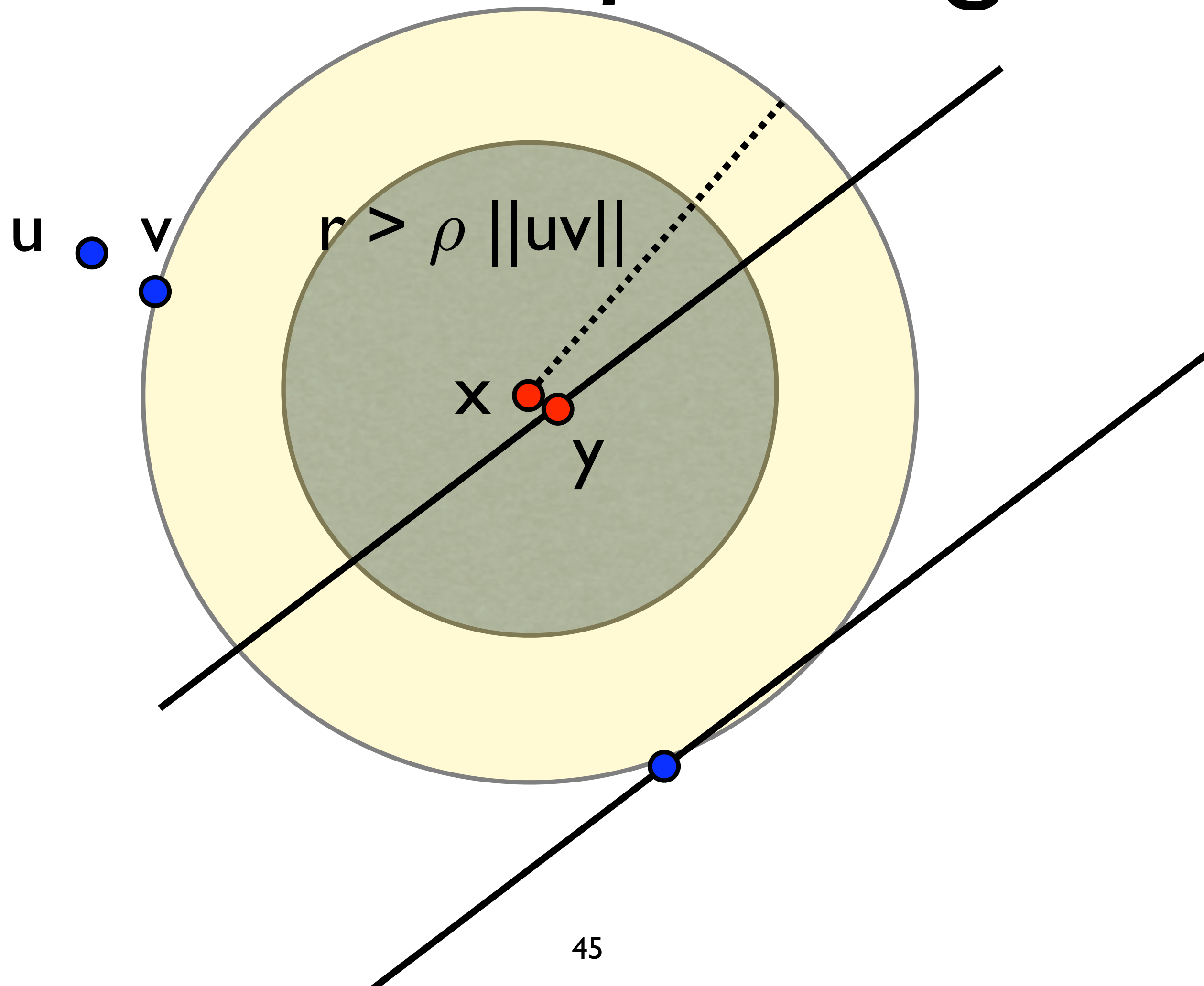
Without violating *size-conforming*



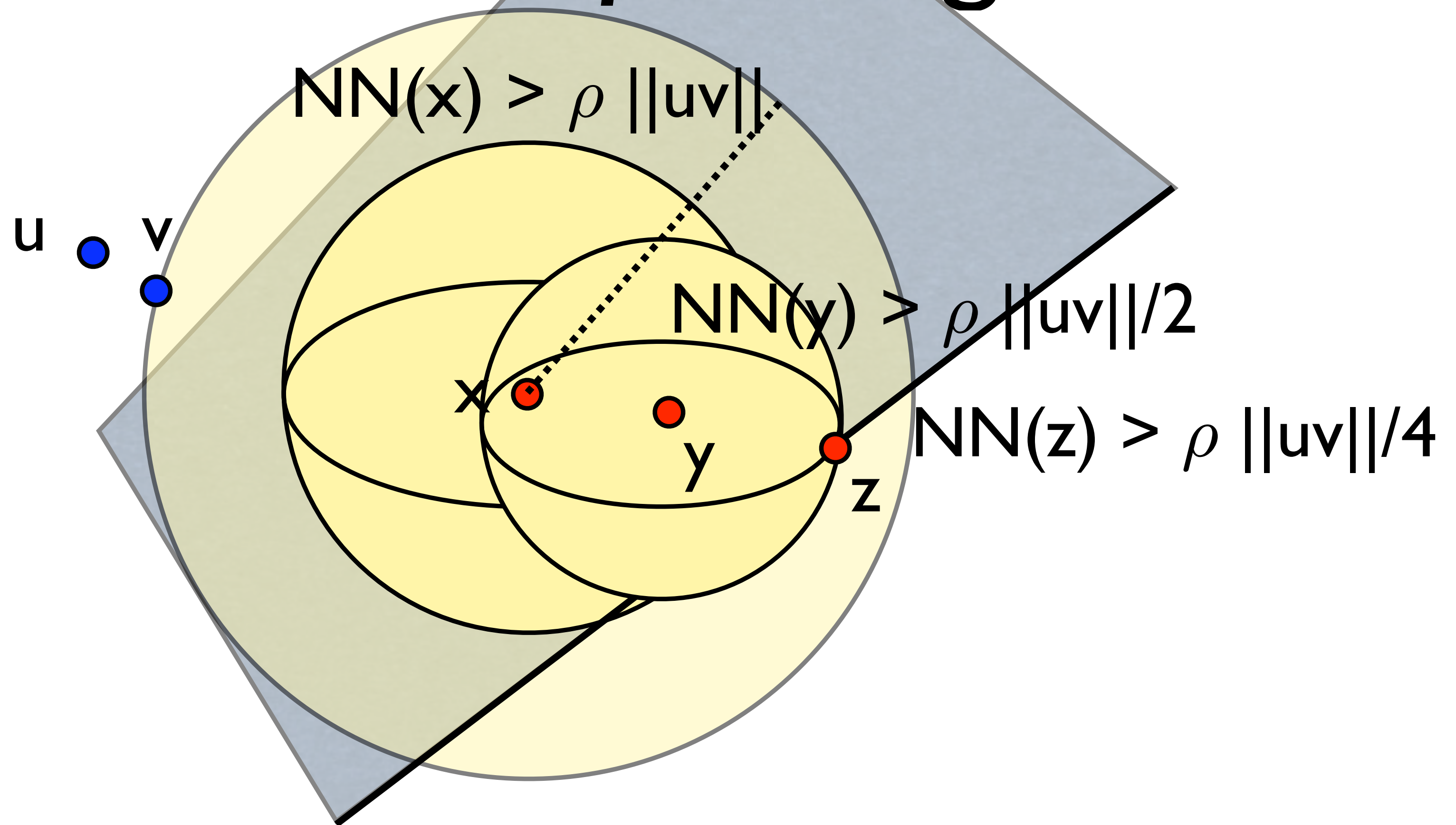
Without violating *size-conforming*



Without violating *size-conforming*

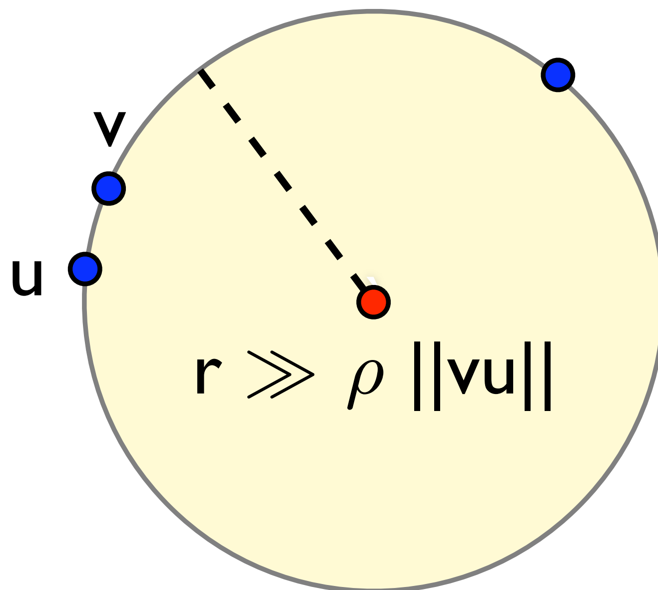


Without violating *size-conforming*

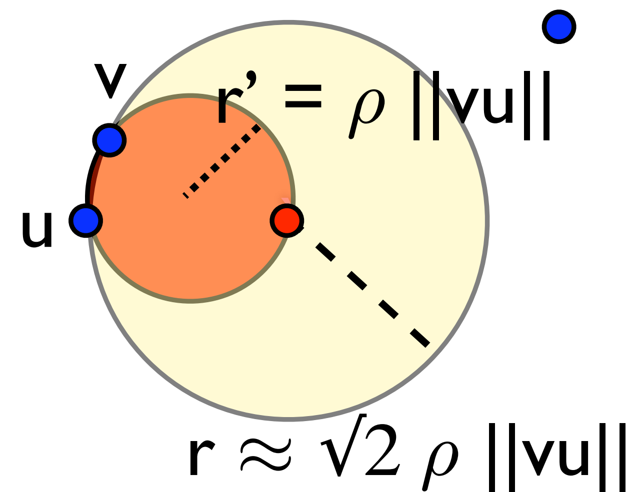


Related work

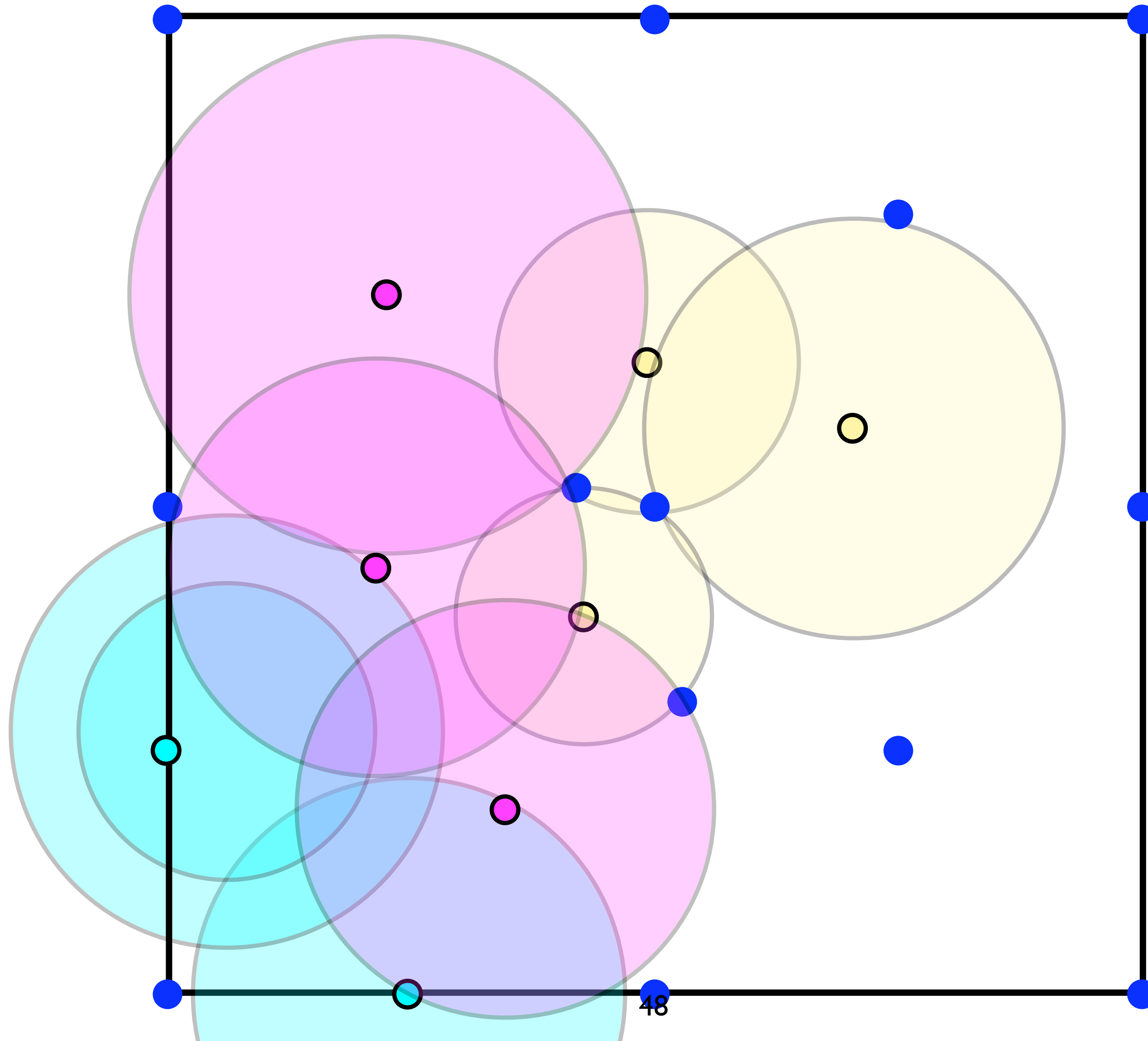
Ruppert, SVR:
Biggest possible



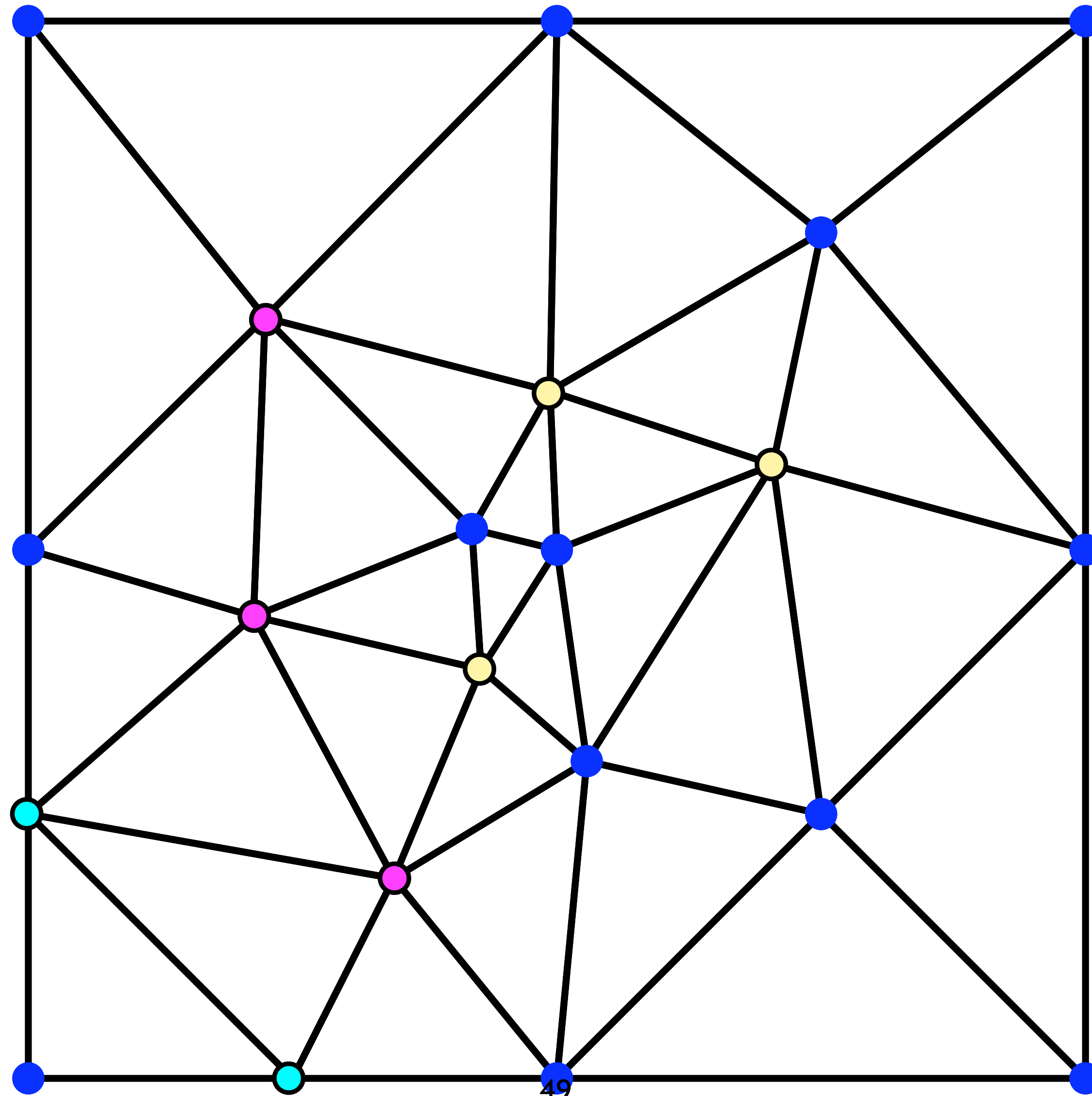
Ungor:
Just-right



Completing a mesh



A complete mesh is good

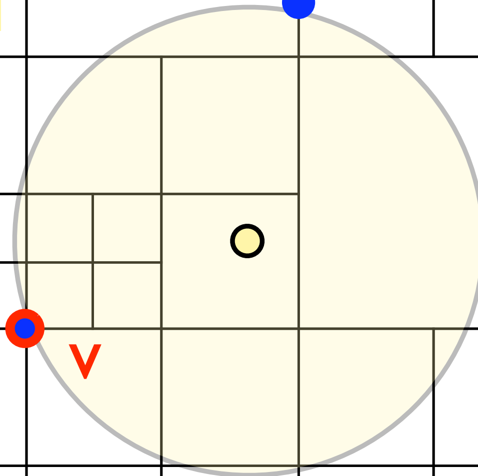


Outline in four parts

- What makes a good mesh?
- Experimental results
- General conceptual algorithm
- **Sub-linear dynamic mesh refinement**

Efficient Queries

How do I know this disc is empty?



Idea credit: Har-Peled, Ungor

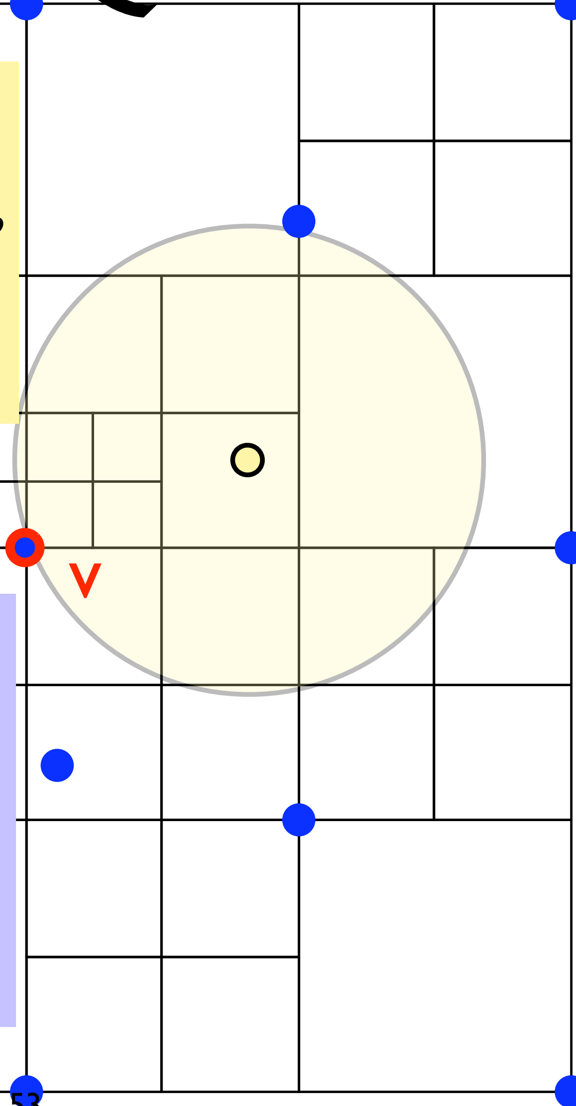
Efficient Algorithm

1. Build the quadtree
2. Add every p to Q, with key NN(p)
3. While Q is not empty
    1. v ← Delete-min(Q)
    2. Complete(v)
    3. Add neighbours of v to Q

Not strictly in order: bucket by $[l, \rho l)$

Efficient Queries

Lemma 3.3.3:
when processing bucket *b*, empty ball of radius r ~ b intersects O(1) squares



Proof sketch:
To intersect many, must intersect small squares.
But small squares are in small buckets ⇒ contain points.

Total runtime

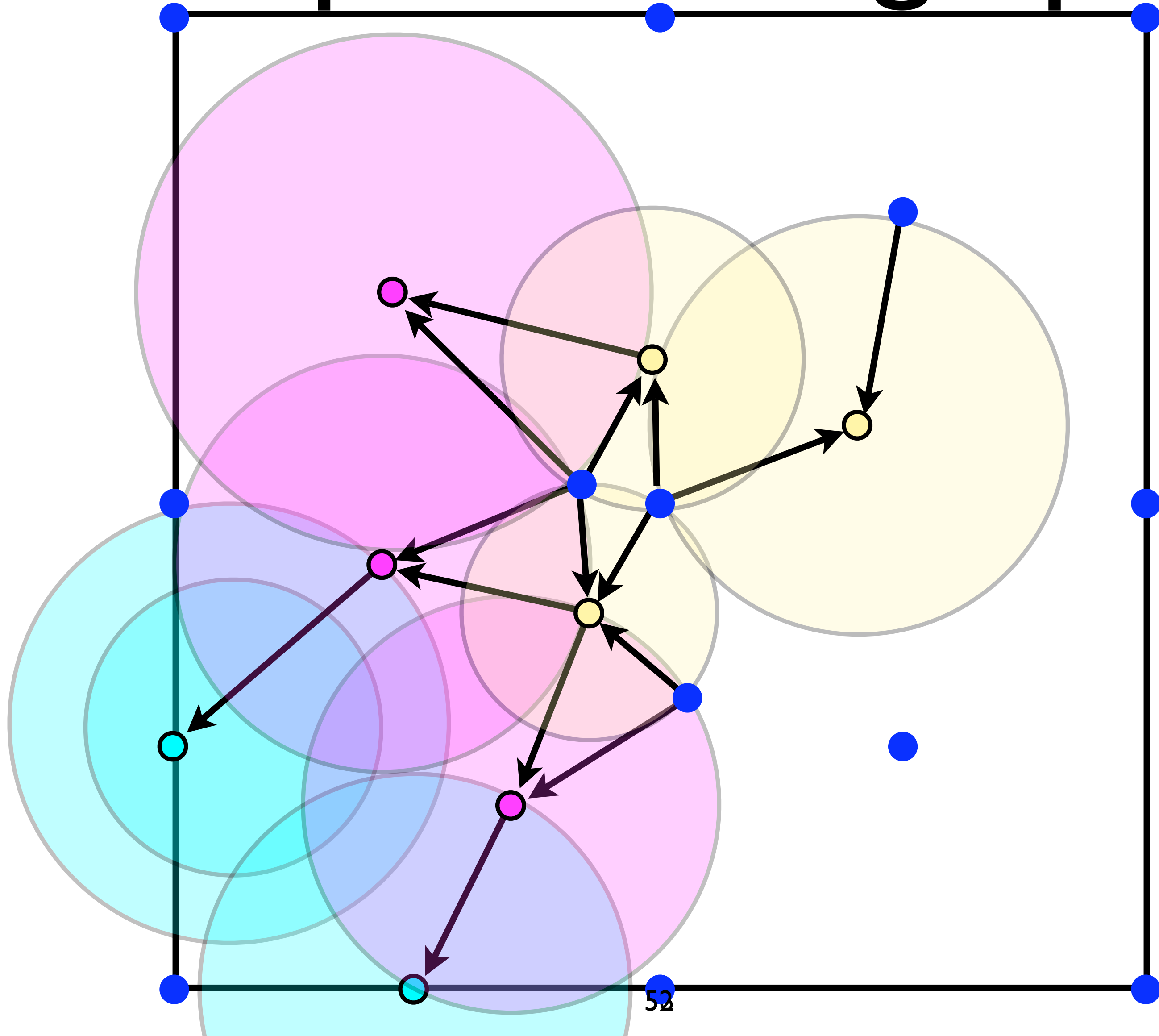
1. Build the quadtree O(n log L/s + m)
2. Add ever
3. While Q tions
   1. v ← Delete-min(Q) O(1) [bucketing]
   2. Complete(v) O(1) [fast queries]
   3. Add neighbours to Q O(1)

Size-conforming
Quality
Respecting
O(n log L/s + m) runtime

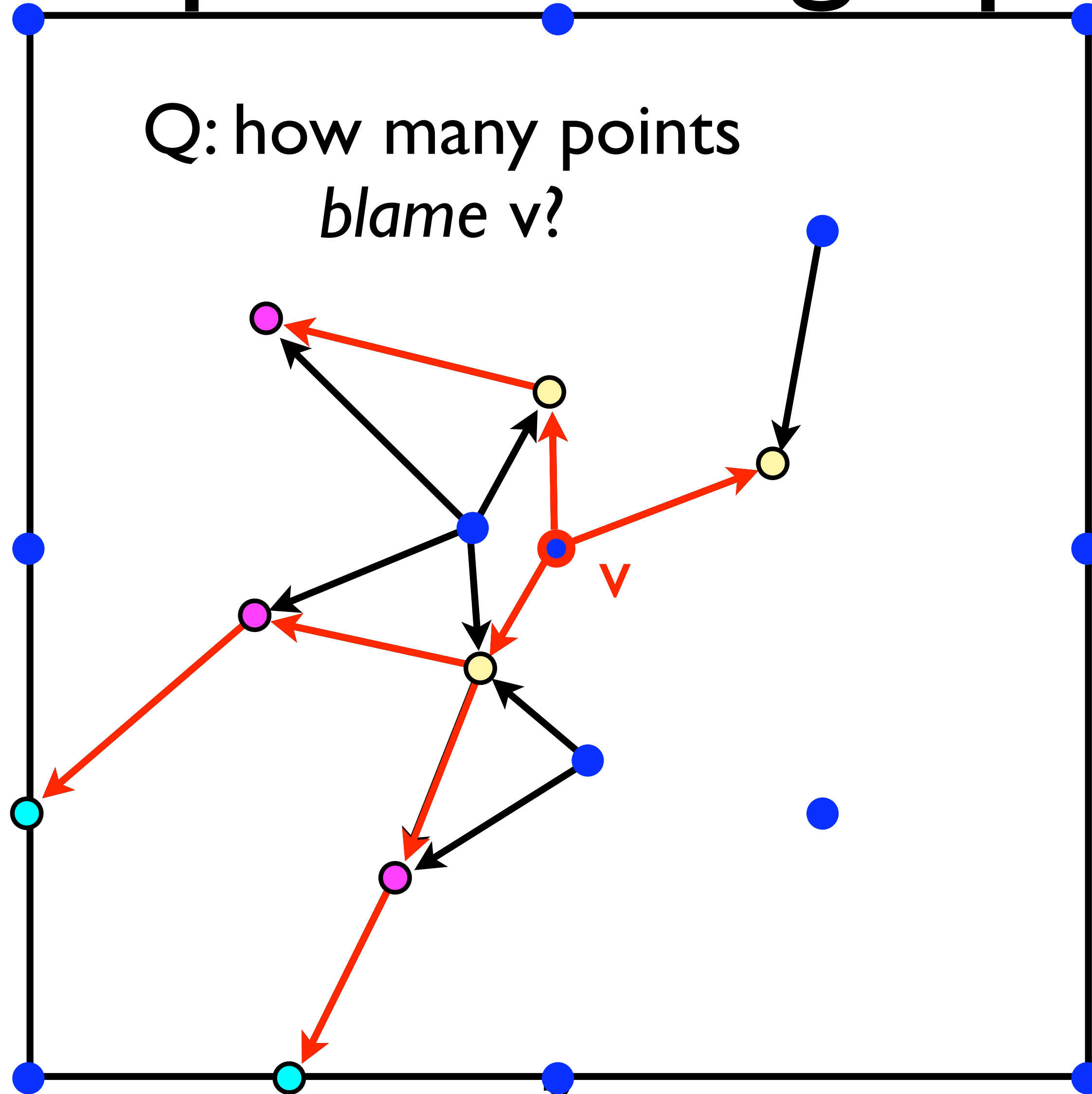
Remeshing

- Change $\Rightarrow$ Reinterpolation $\Rightarrow$ Error
- No change $\Rightarrow$ Why recompute it?

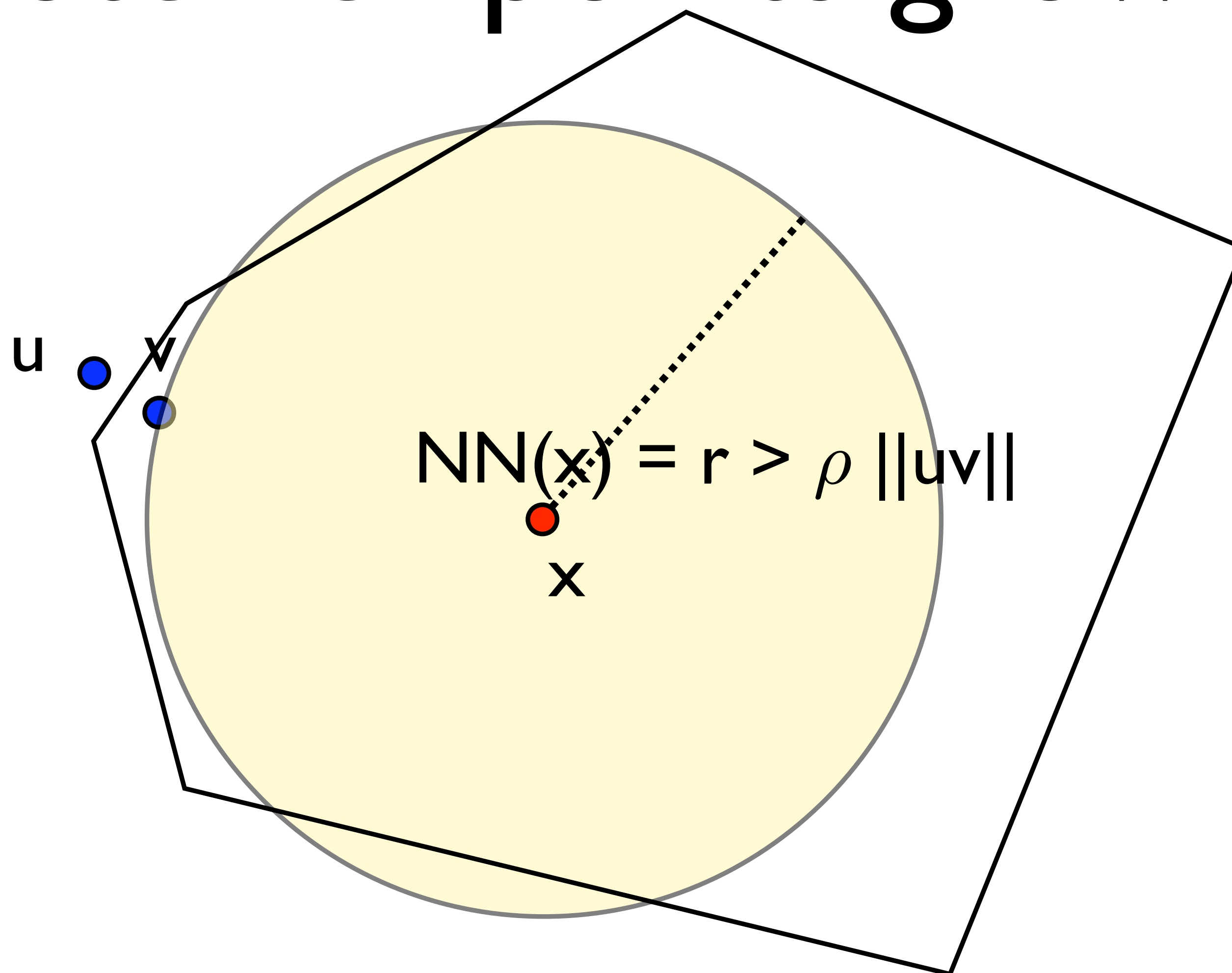
Dependence graph



Dependence graph



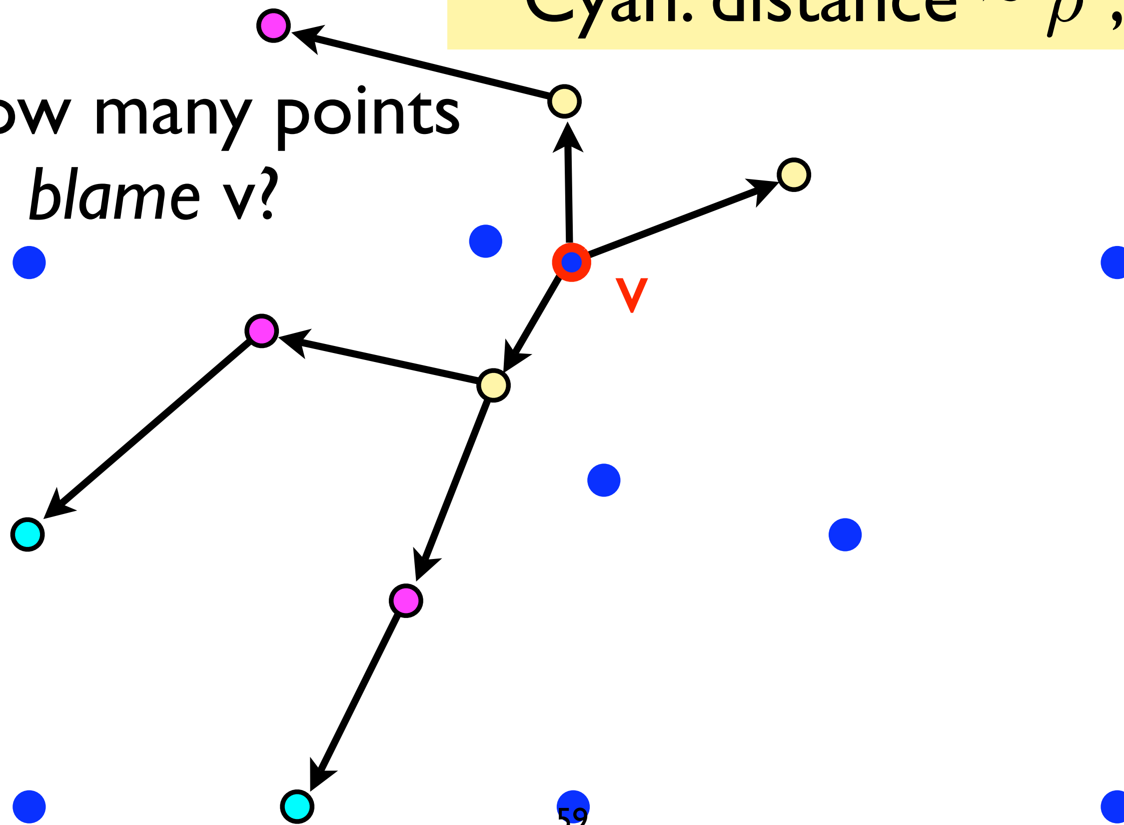
Steiner points grow



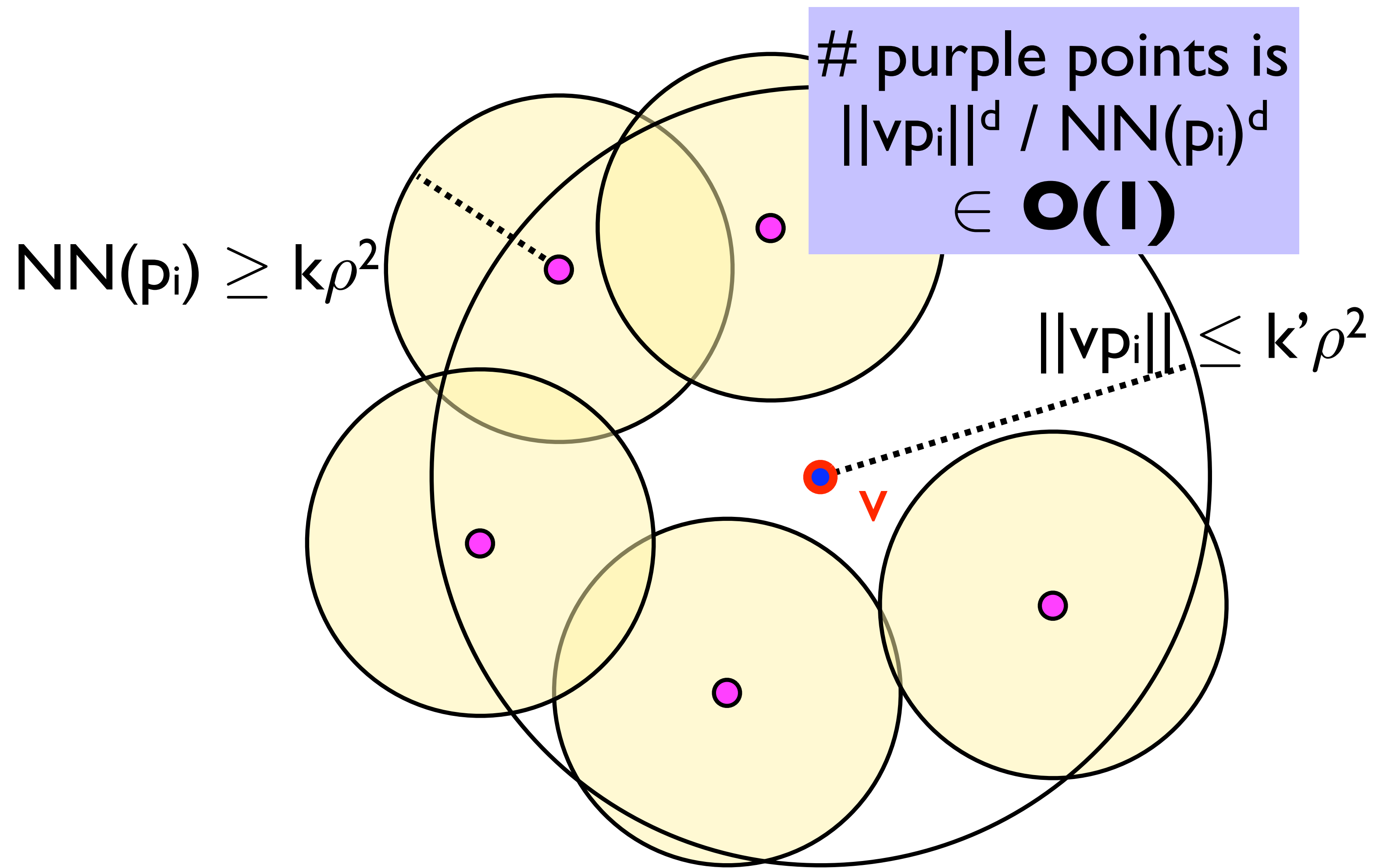
Dynamic stability

Yellow: distance ~ ρ, NN ~ ρ
Purple: distance ~ ρ^2, NN ~ ρ^2
Cyan: distance ~ ρ^3, NN ~ ρ^3

Q: how many points *blame* v?



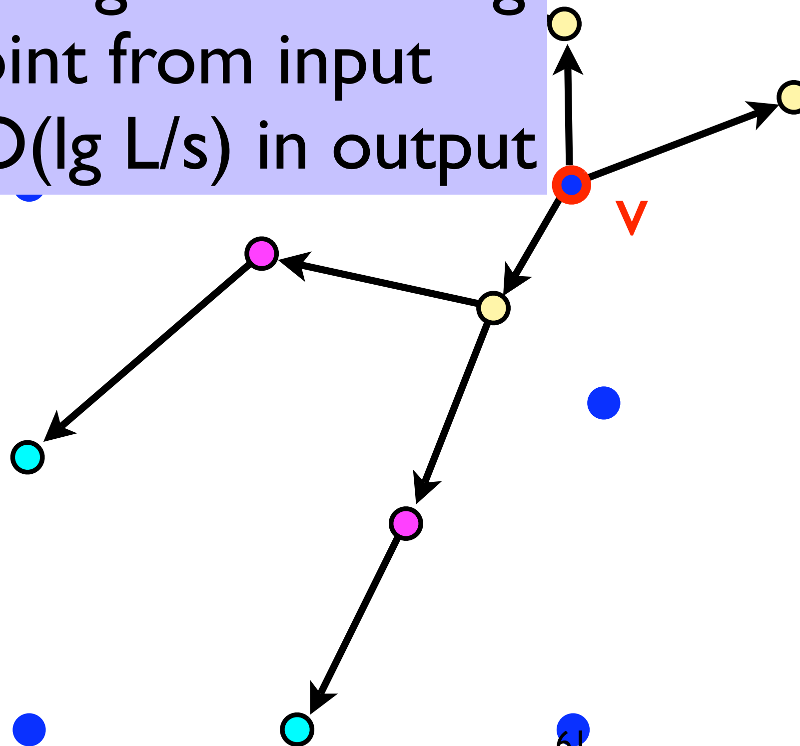
Packing Purple Points



Dynamic stability

Yellow: O(1)
Purple: O(1)
Cyan: O(1)
...
$O(\log_\rho L/s)$ total

Proves: adding or removing one point from input modifies O(lg L/s) in output



Dynamic Algorithm

On update, simulate rerunni scra

Use *Self-adjusting computation*: pay only for the changes.

1. Build the quadtree
2. Add every p to Q
3. npty
    1. v ← Delete-min(Q)
    2. Complete(v)
    3. Add neighbours to Q

Update time: O(log L/s)!

Outline in four parts

- What makes a good mesh?
- Experimental results
- General conceptual algorithm
- Sub-linear dynamic mesh refinement

Handling features

- Details in the document
- Static runtime is $O(n \log L/s + m)$
- Dynamic: if we need *k* points to respect, update time is $O(k \times \log L/s)$
- Future: prove $O(k + \log L/s)$
  - Probably "just" a proof

Moving meshes

- Dynamic update: add or remove
- Kinetic update: move
- Self-adjusting framework can do kinetic
- I have no proofs ... yet

Claims of the Thesis

- The fastest static meshing code
    - And the first without pathologies
- A framework with lots of explicit freedom for point placement
- The first dynamic meshing algorithm
- A few more minor results