Geometry Reconstruction

Baoquan Chen

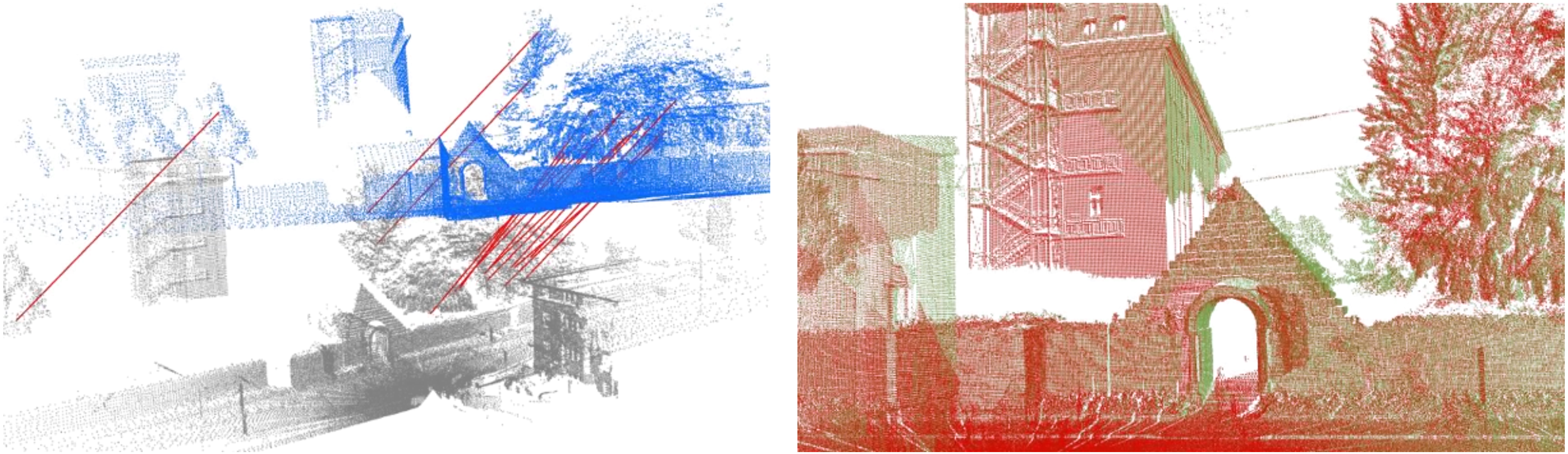
Reconstruction

- Data: 3D point clouds
  - From Lidars and RGB-D cameras
  - From computer vision algorithms such as triangulation, bundle adjustment, and deep learning

Figure: point clouds generate from buildings in Peking University.

Reconstruction

- After getting 3D point clouds
  - Registration

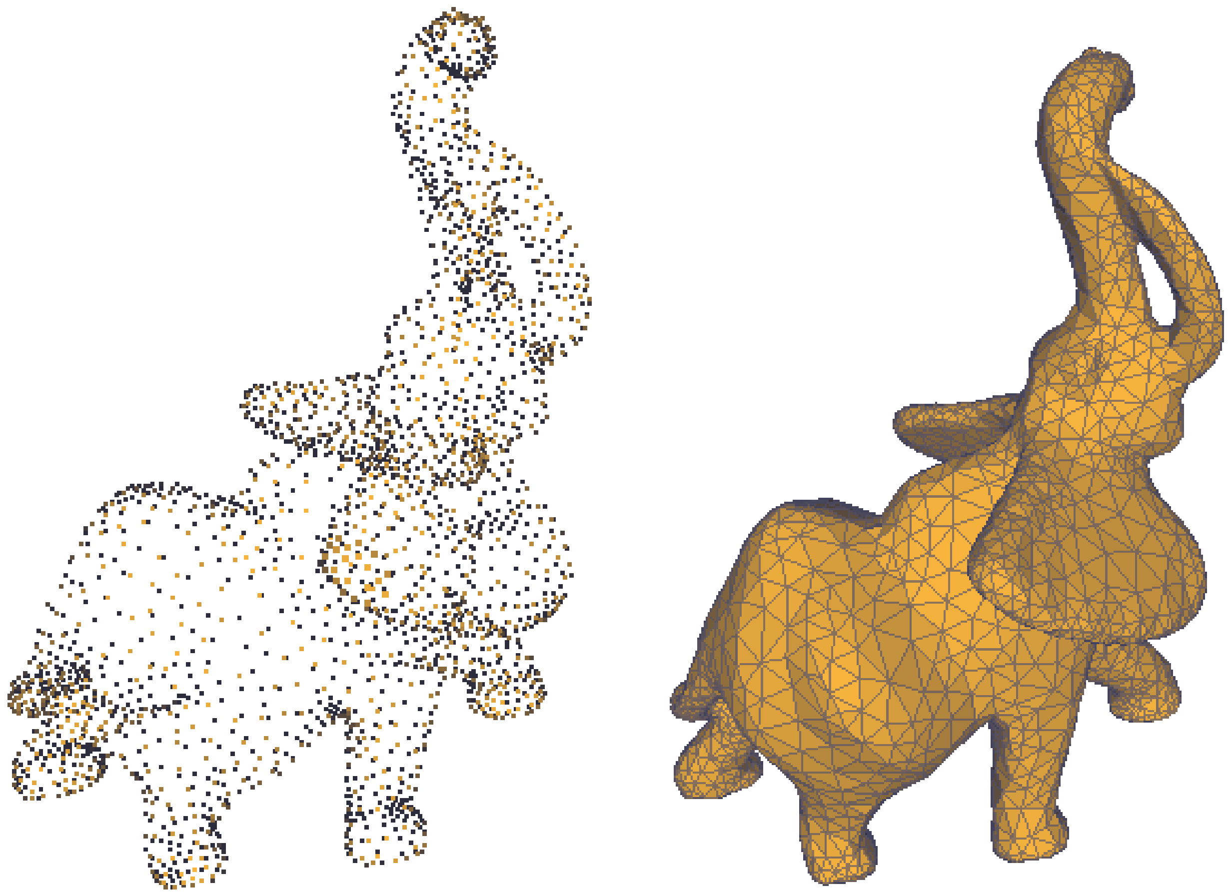


Courtesy: Martin Holzkothen, Michael Korn

Figure
(a) Left: data from two 3D scans of the same environment need to be aligned using point set registration.
(b) Right: data registered successfully using a variant of iterative closest point.

Reconstruction

- After getting 3D point clouds
  - Surface Reconstruction
  - Motivation:
    - 1) Effective rendering of the model
    - 2) Computational analysis
    - 3) other geometry processings: parameterization, morphing, blending etc..

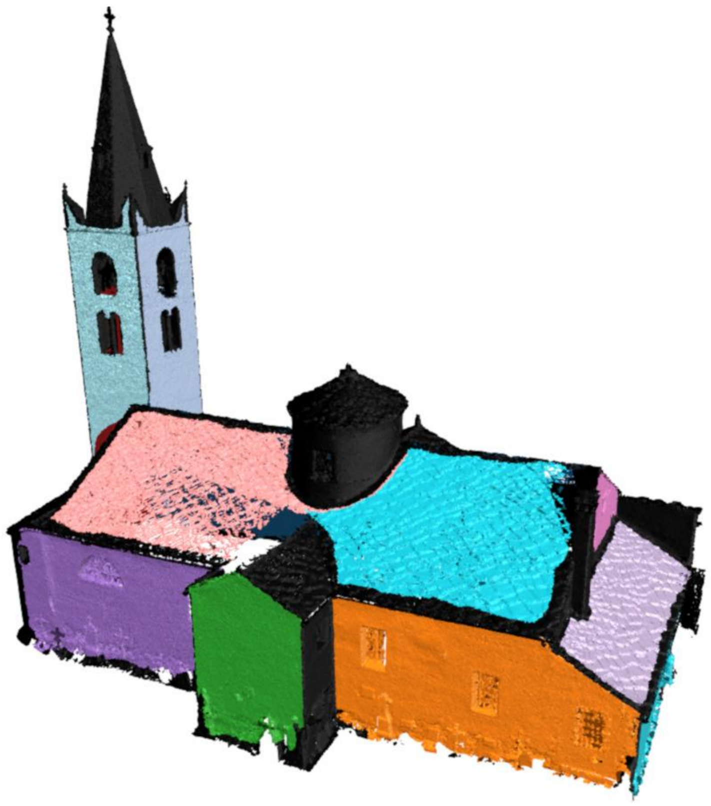


Courtesy: Jiju Peethambaran and Ramanathan Muthuganapathy

Figure: before and after surface reconstruction (triangulation).

Reconstruction

- After getting 3D point clouds
  - Model Fitting
    - Plane
    - Sphere
    - Cube
    - …
  - Usually by RANSAC
    - RANdom SAmple Consensus



Courtesy: https://github.com/STORM-IRIT/Plane-Detection-Point-Cloud

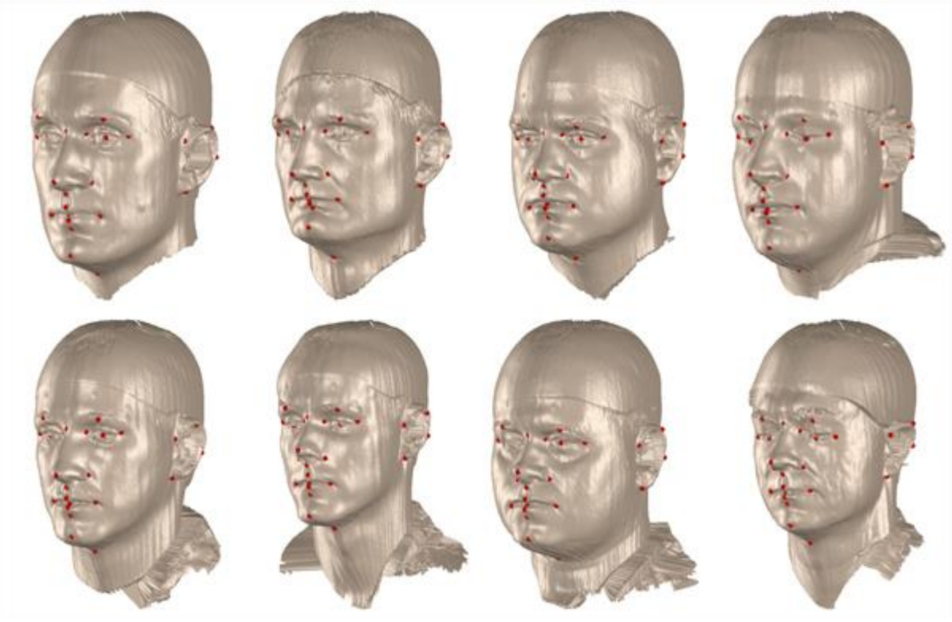
Figure: the plane detection of the model of a building

Outline

- Registration
    - Iterative Closest Point (ICP)
- Surface Reconstruction
    - Delaunay Triangulation
    - Poisson Surface Reconstruction
- Model Fitting
    - RANSAC

Registration

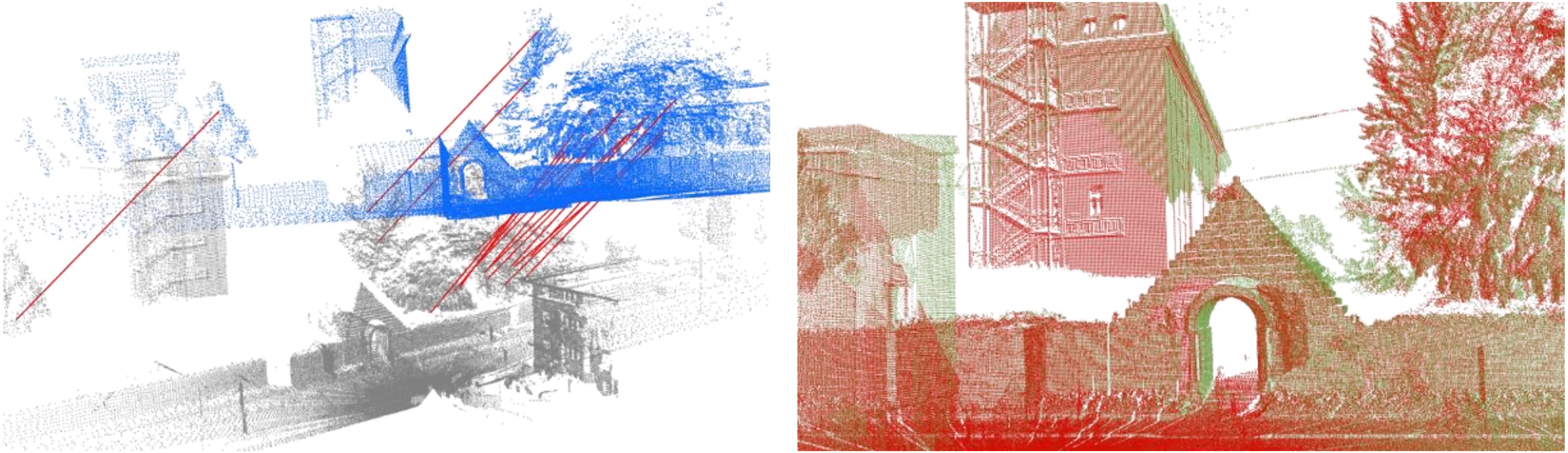
- Definition: transform one model to another based on their partially overlapping features
  - Automated annotation
  - Tracking and motion analysis
  - Shape and data comparison



Courtesy: Schneider and Eisert

Registration

- We need do alignment.
  - Alignment: find a correct transformation

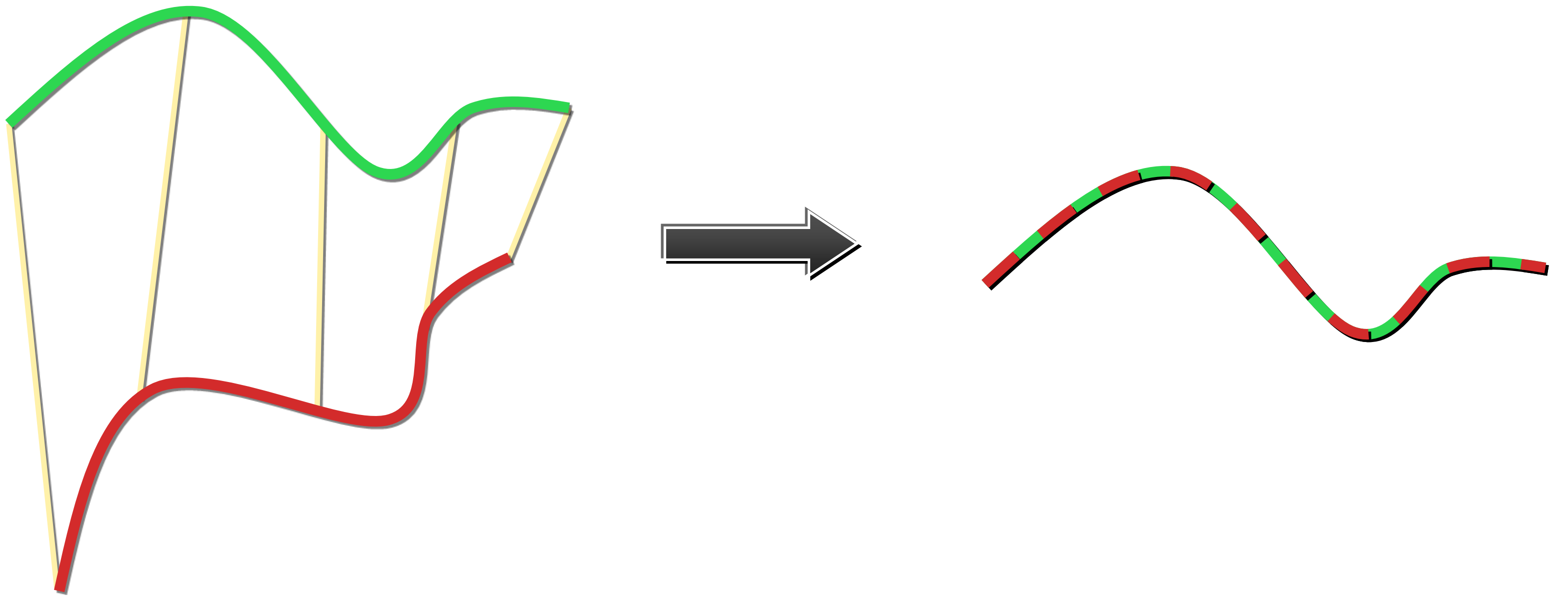


Courtesy: Martin Holzkothen, Michael Korn

Figure
(a) Left: data from two 3D scans of the same environment need to be aligned using point set registration.
(b) Right: data registered successfully using a variant of iterative closest point.

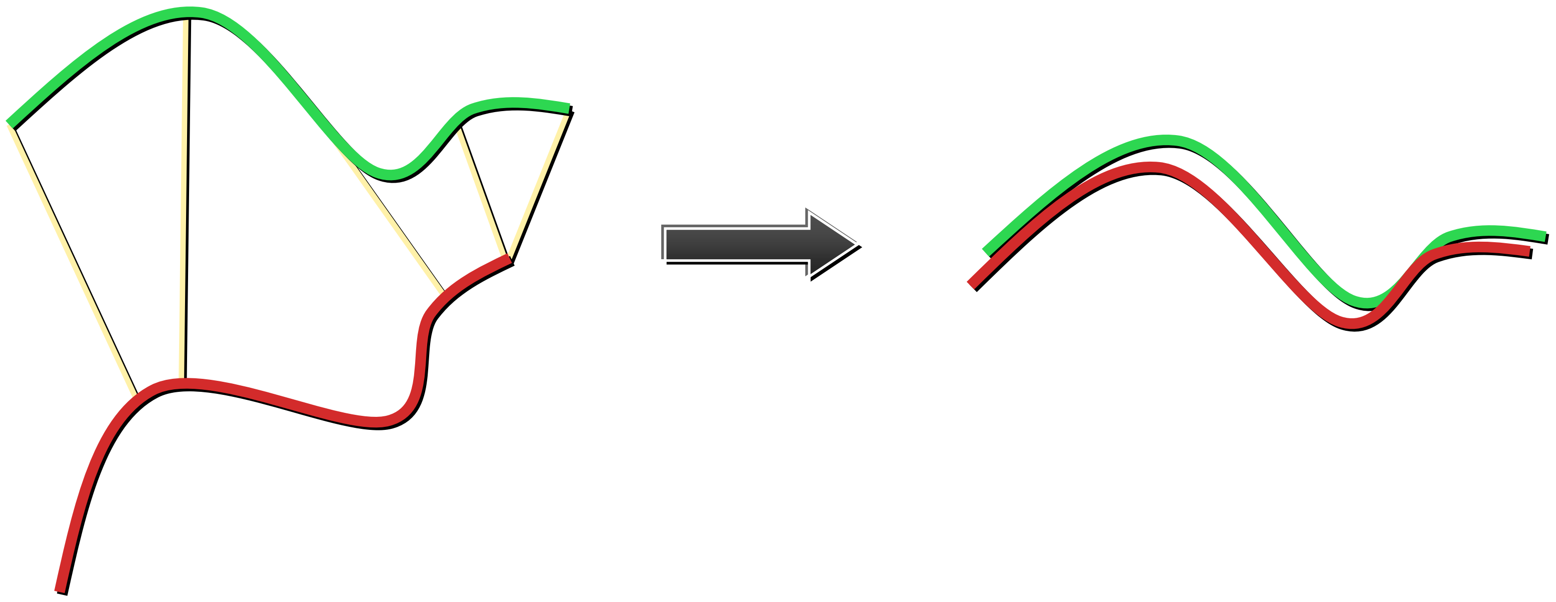
Alignment

- If correct correspondences are known, we can easily find correct relative rotation/translation



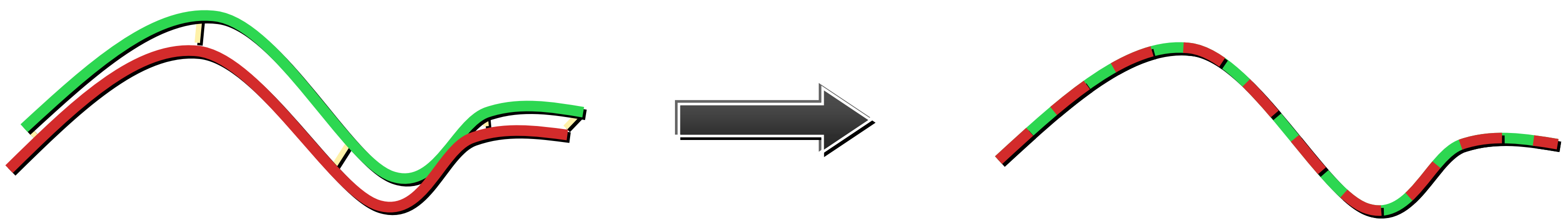
Alignment

- How to find correspondences:
  - User input? Feature detection? Signatures?
- Alternative: assume closest points correspond



Alignment

- … and iterate to find alignment
  - Iterative Closest Points (ICP) [Besl & McKay 92]
- Converges if starting position "close enough"



ICP

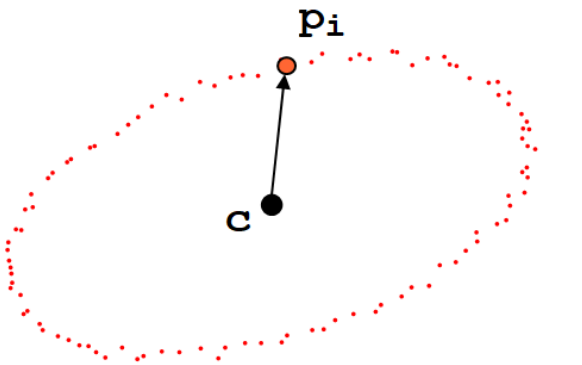
- 1. **Initialize** transformation R, t by PCA (Principle Component Analysis)
- 2. **Match** each to closest point $p_i' = Rp_i + t$ on other scan
- 3. **Reject** pairs with distance > k times median
- 4. Construct **error function**:

$$E = \sum |Rp_i + t - q_i|^2$$

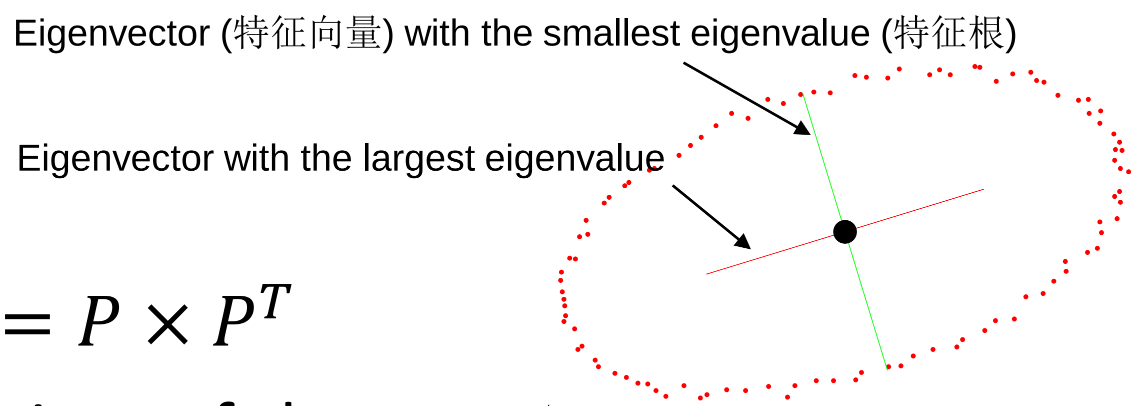
- 5. **Minimize** the error by SVD
- 6. **Loop** step 2-5 until the error is small enough

ICP: Initialize by PCA

- PCA: Principal Component Analysis
  - It can be used to compute axes
- Consider a set of points $p_1, \ldots, p_n$ with centroid location c
- Let P be a matrix whose i-th column is vector $p_i - c$



$$\mathbf{P} = \begin{pmatrix} \mathbf{p1_x} - \mathbf{c_x} & \mathbf{p2_x} - \mathbf{c_x} & \cdots & \mathbf{pn_x} - \mathbf{c_x} \\ \mathbf{p1_y} - \mathbf{c_y} & \mathbf{p2_y} - \mathbf{c_y} & \cdots & \mathbf{pn_y} - \mathbf{c_y} \\ \mathbf{p1_z} - \mathbf{c_z} & \mathbf{p2_z} - \mathbf{c_z} & \cdots & \mathbf{pn_z} - \mathbf{c_z} \end{pmatrix}$$



- Build the covariance matrix (协方差矩阵) : $M = P \times P^T$
- Eigenvectors of M represent principal directions of shape variation
  - The eigenvectors form orthogonal axes (2 vectors in 2D; 3 vectors in 3D)
    - Note they are “un-signed”: lacking an orientation.

ICP: Initialize by PCA

- After finding axes, how to do alignment?
- PCA-based alignment
  - Let c_S, c_T be centroids of source and target.
- First, translate source to align c_S with c_T:

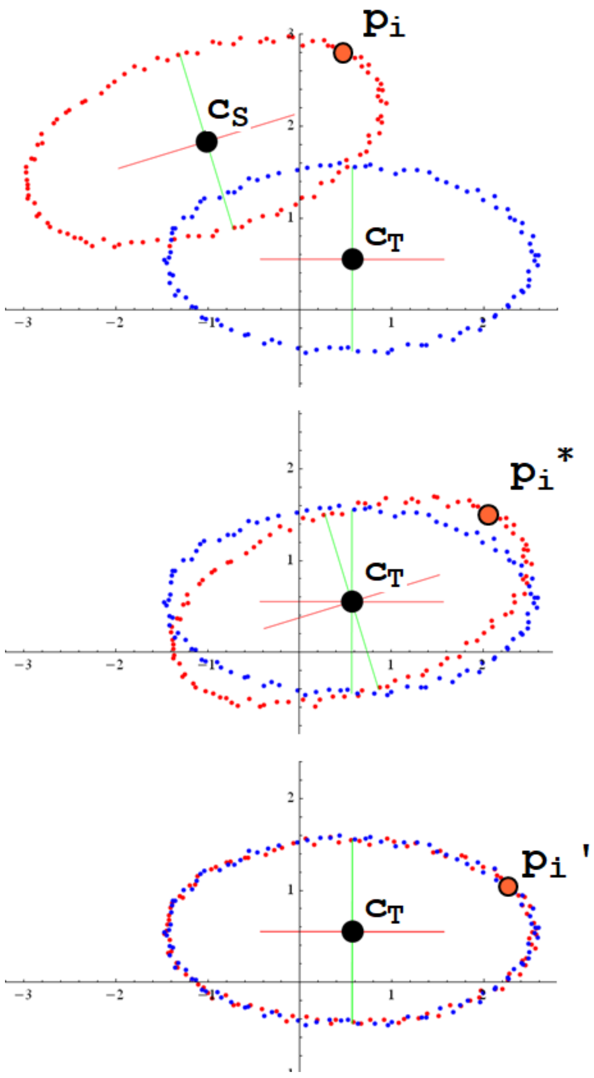
$$p_i^* = p_i + (c_T - c_S)$$

- Next, find rotation R that aligns two sets of PCA axes

$$p_i' = c_T + R \times (p_i^* - c_T)$$

- Combined:

$$p_i' = c_T + R \times (p_i - c_S)$$



ICP: Initialize by PCA

- After finding axes, how to do alignment?
- PCA-based alignment
  - Let c_S, c_T be centroids of source and target.
- First, translate source to align c_S with c_T:

$$p_i^* = p_i + (c_T - c_S)$$

- Next, find rotation R that aligns two sets of PCA axes

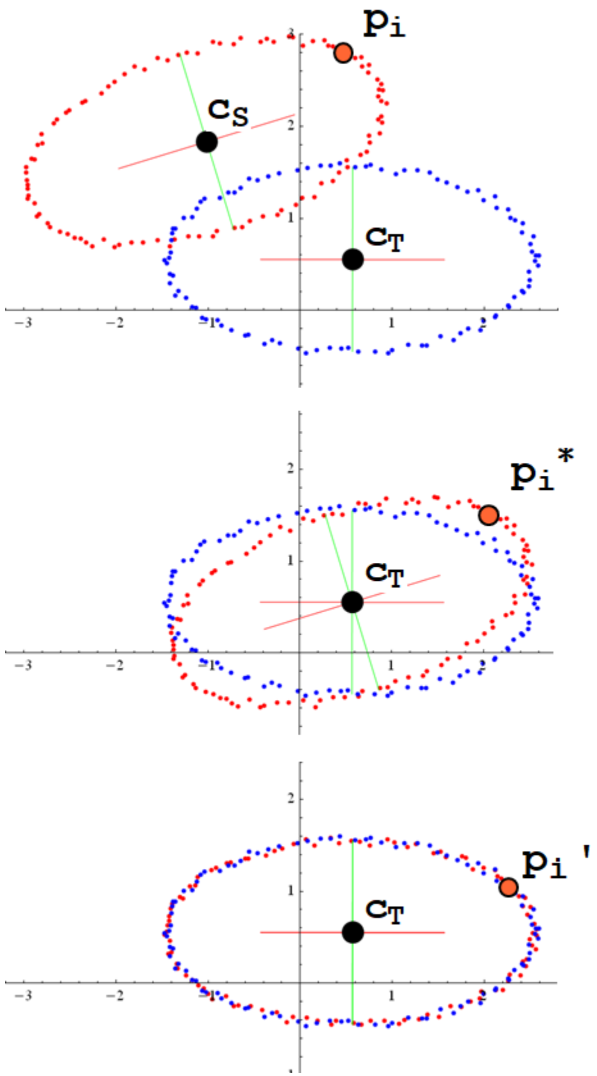
$$p_i' = c_T + R \times (p_i^* - c_T)$$

- Combined:

$$p_i' = c_T + R \times (p_i - c_S)$$

- Then we get the initial transformation:

$$R = R, t = c_T - R \times c_S$$



ICP

- 1. **Initialize** transformation R, t by PCA
- 2. **Match** each to closest point $p_i' = Rp_i + t$ on other scan
- 3. Reject pairs with distance > k times median
- 4. Construct error function:

$$E = \sum |Rp_i + t - q_i|^2$$

- 5. Minimize the error by SVD
- 6. Loop step 2-5 until the error is small enough

ICP

- 1. **Initialize** transformation R, t by PCA
- 2. **Match** each to closest point $p_i' = Rp_i + t$ on other scan
- 3. **Reject** pairs with distance > k times median
- 4. Construct error function:

$$E = \sum |Rp_i + t - q_i|^2$$

- 5. Minimize the error by SVD
- 6. Loop step 2-5 until the error is small enough

ICP

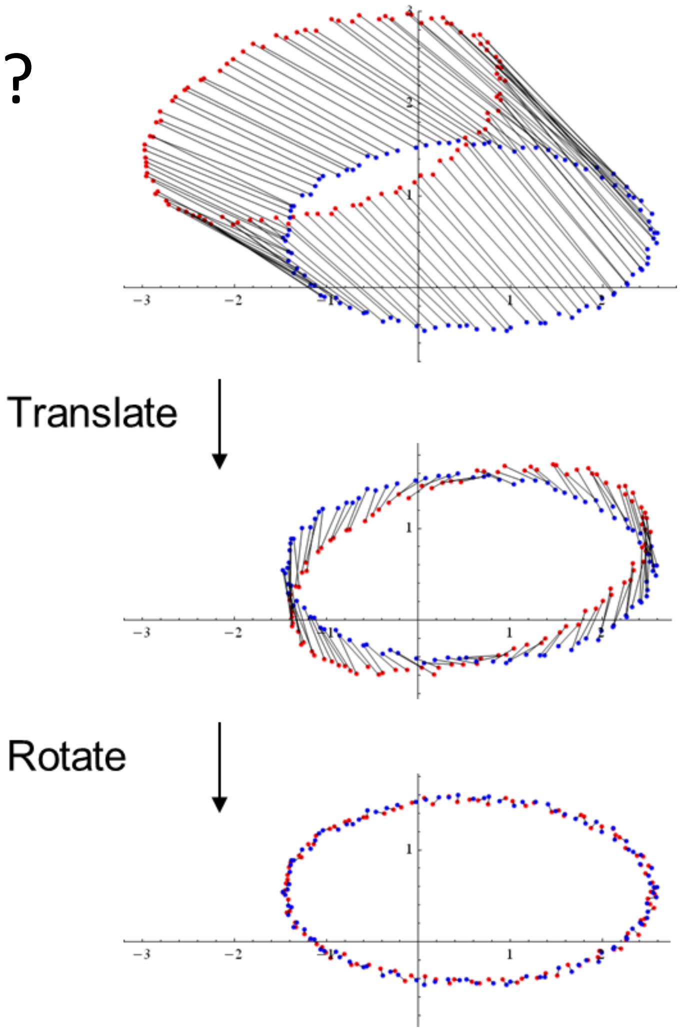
- 1. **Initialize** transformation R, t by PCA
- 2. **Match** each to closest point $p_i' = Rp_i + t$ on other scan
- 3. **Reject** pairs with distance > k times median
- 4. Construct **error function**:

$$E = \sum |Rp_i + t - q_i|^2$$

- 5. Minimize the error by SVD
- 6. Loop step 2-5 until the error is small enough

ICP: Minimize by SVD

- Now we have pairs (p_i, q_i) from two scans
- How to find a better R, t to minimize error function?
- Use Singular Value Decomposition (SVD/奇异值分解):
  - Let P be a matrix whose i-th column is vector $p_i - c_S$
  - Let Q be a matrix whose i-th column is vector $q_i - c_T$
  - Forming the cross-covariance matrix (互协方差矩阵)
$$M = P \times Q^T$$
  - Computing SVD
$$M = U \times \Sigma \times V^T$$
  - The rotation matrix is
$$R = V \times U^T$$
  - Translate and rotate the source:
$$R = V \times U^T, t = c_T - R \times c_S$$



ICP

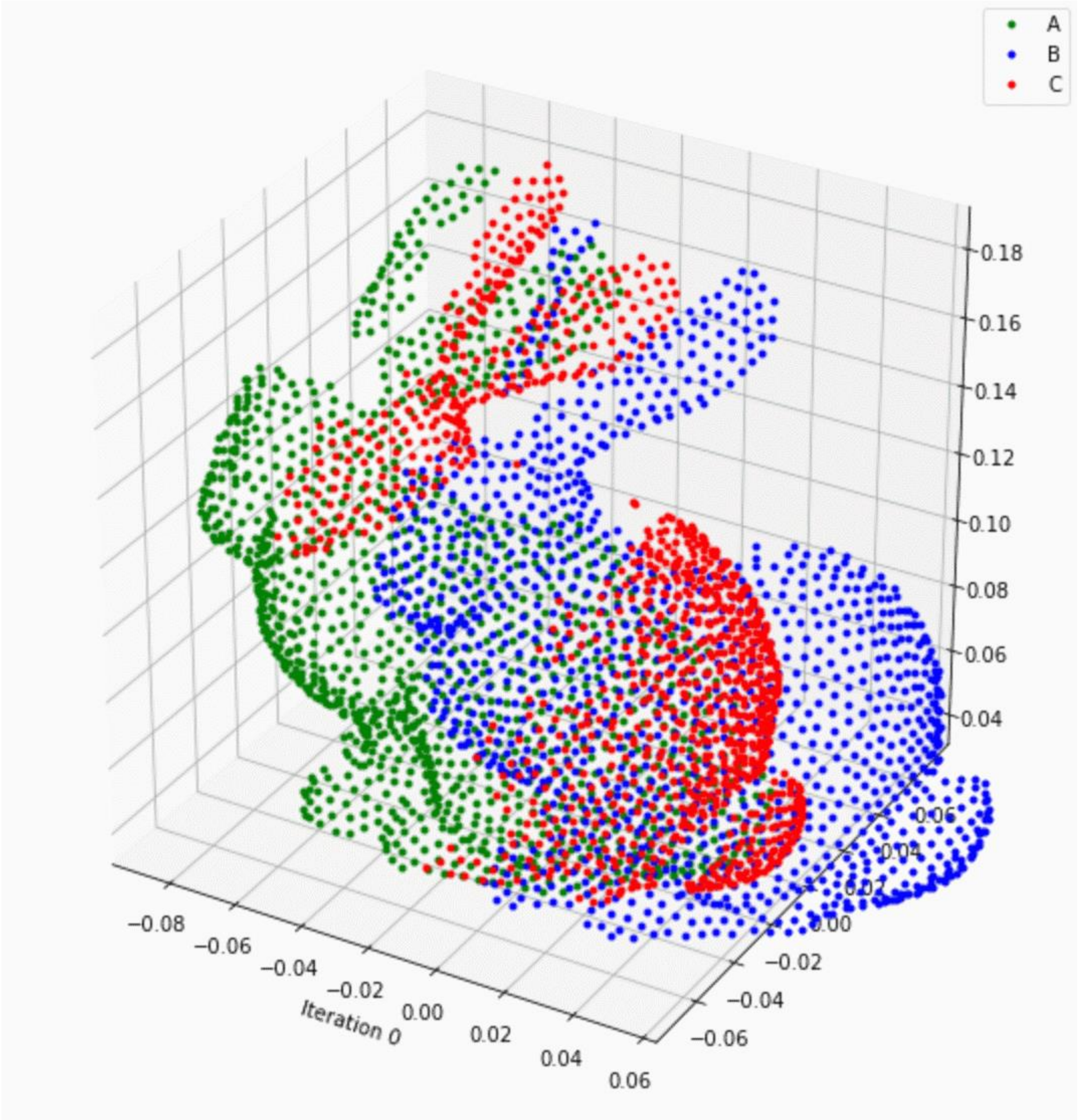
- 1. **Initialize** transformation R, t by PCA
- 2. **Match** each to closest point $p_i' = Rp_i + t$ on other scan
- 3. **Reject** pairs with distance > k times median
- 4. Construct **error function**:

$$E = \sum |Rp_i + t - q_i|^2$$

- 5. **Minimize** the error by SVD
- 6. **Loop** step 2-5 until the error is small enough

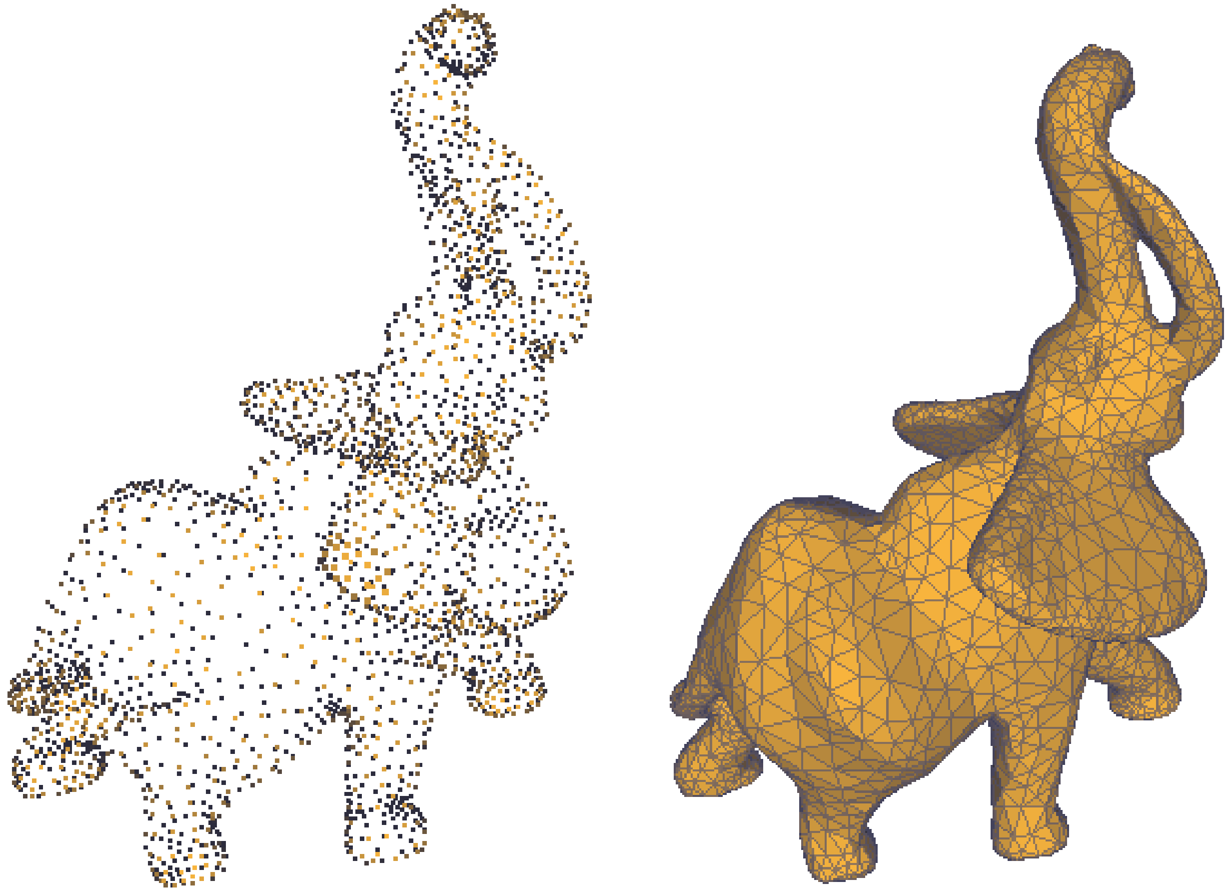
ICP

- Visualization of an ICP variant
  - https://laempy.github.io/pyoints/tutorials/icp.html#References
- Also, there are many variants of ICP. They are more robust and more efficient. Refer to:
  - https://gfx.cs.princeton.edu/proj/iccv05_course/iccv05_icp_gr.ppt



Surface Reconstruction

- Motivation:
  - 1) Effective rendering of the model
  - 2) Computational analysis
  - 3) Other geometry processings: parameterization, morphing, blending etc..
- Methods:
  - CAD: manually construct
  - Algorithm: automatically generate
    - mainly from point cloud

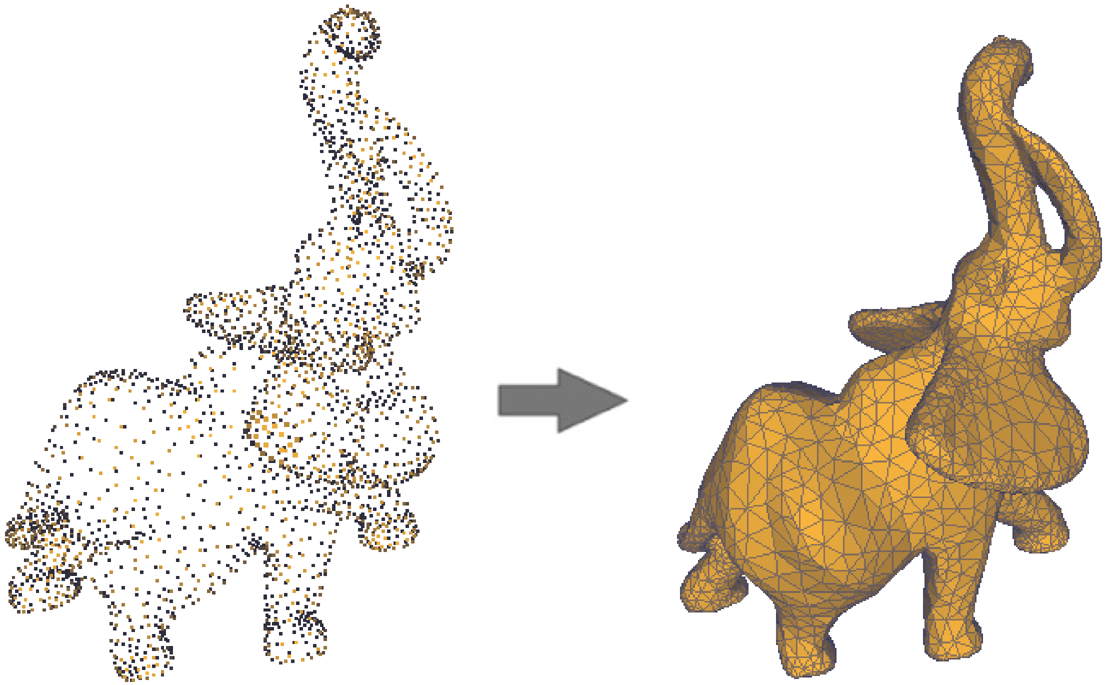


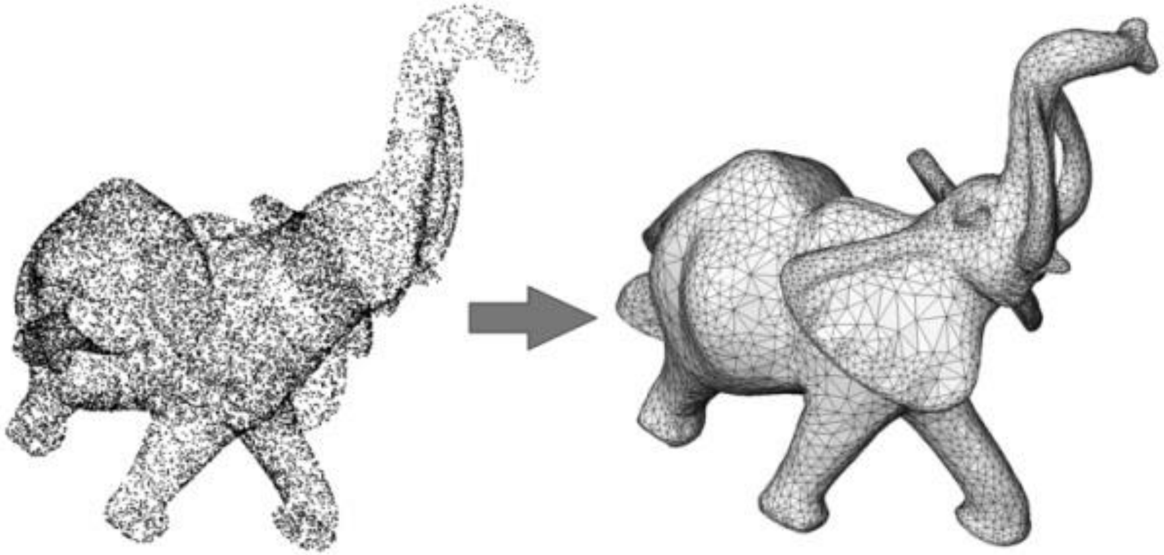
Courtesy: Jiju Peethambaran and Ramanathan Muthuganapathy

Figure: before and after surface reconstruction (triangulation).

Surface Reconstruction Algorithm

- Classic surface reconstruction from point cloud:
  - Directly: Triangulation
    - Delaunay Triangulation
  - Indirectly: Implicit Surfaces + Marching Cubes
    - Poisson Surface Reconstruction





Left: Delaunay Triangulation

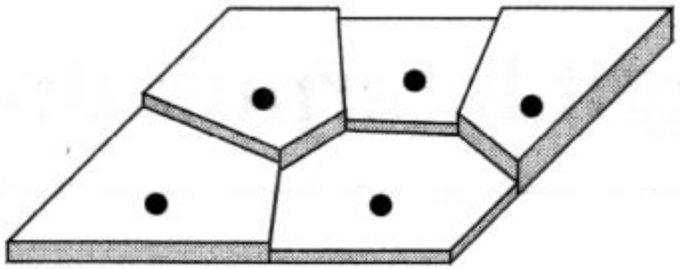
Right: Poisson Surface Reconstruction

Delaunay Triangulation in 2D

- Motivation (in 2D): Terrains
- Set of data points $A \subset R^2$
- Height $f(p)$ defined at each point p in A
- How can we most naturally approximate height of points not in A?

Delaunay Triangulation in 2D

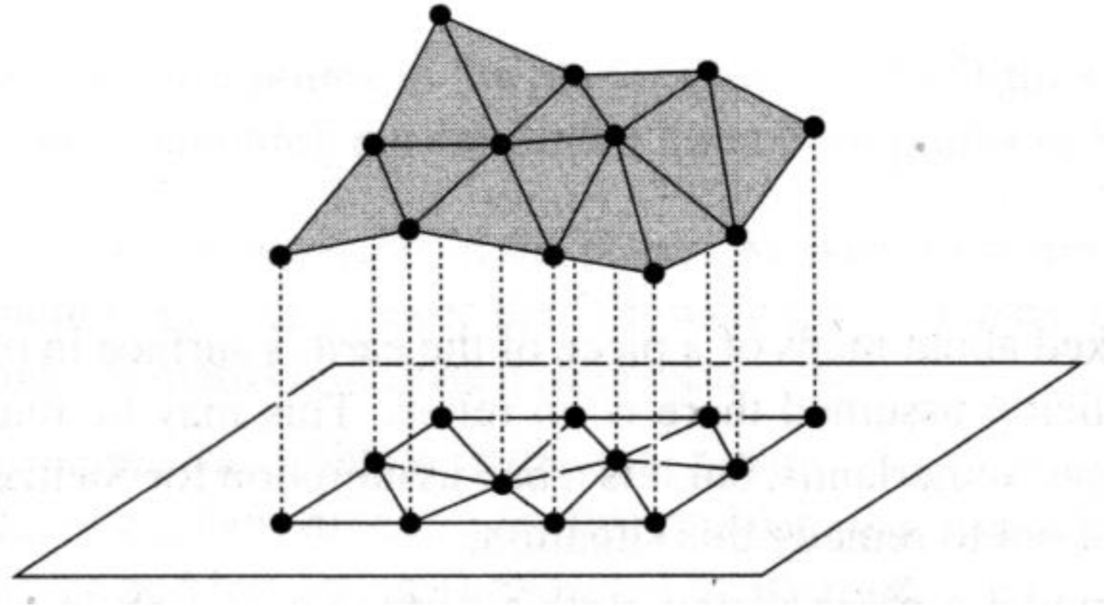
- Motivation (in 2D): Terrains
- Set of data points $A \subset R^2$
- Height $f(p)$ defined at each point p in A
- How can we most naturally approximate height of points not in A?
- Option: Discretize
  - Let $f(p')$ = height of nearest point $p \in A$ for points $p' \notin A$
  - Does not look natural



Borrowed from Glenn Eguchi, MIT

Delaunay Triangulation in 2D

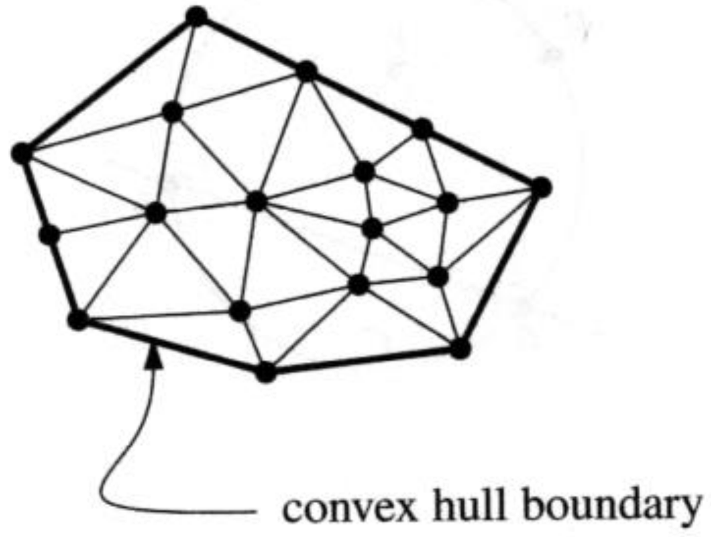
- Motivation (in 2D): Terrains
- Set of data points $A \subset R^2$
- Height $f(p)$ defined at each point p in A
- How can we most naturally approximate height of points not in A?
- Option: Discretize
  - Does not look natural
- Option: Linear Interpolation?
  - Determine a **triangulation** of $A \subset R^2$
  - Then raise points to desired height
  - Now, we can get heights of points not in A



Borrowed from Glenn Eguchi, MIT

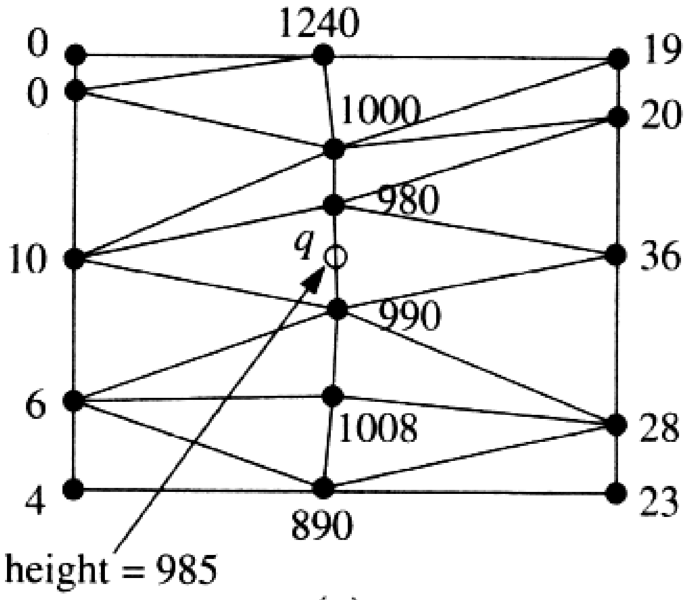
Delaunay Triangulation in 2D

- Formal Definition
  - **maximal planar subdivision**: a subdivision S such that no edge connecting two vertices can be added to S without destroying its **planarity** (its edges intersect only at their endpoints).
  - **triangulation** of set of points P: a maximal planar subdivision whose vertices are elements of P.
- It can be proved:
  - Outer polygon must be convex hull
  - Internal faces must be triangles

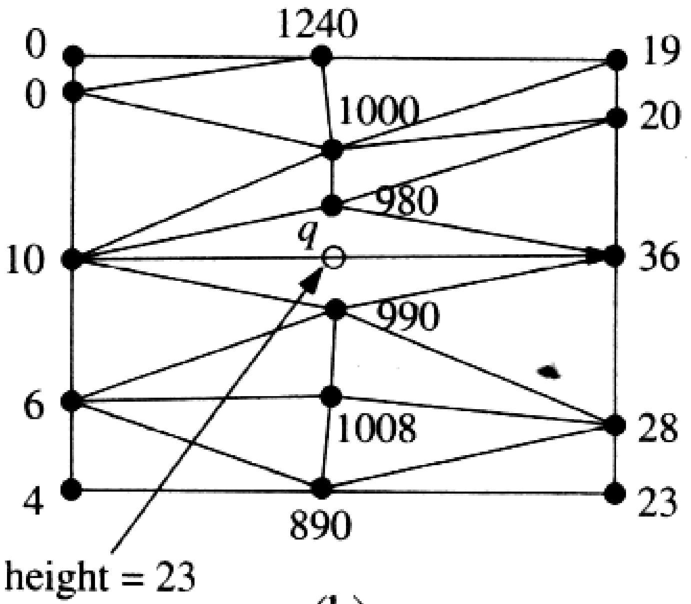


Delaunay Triangulation in 2D

- Let’s revisit Terrain Problem:
  - We believe that some triangulations are “better” than others.
  - Why?
  - See an example.



(a)

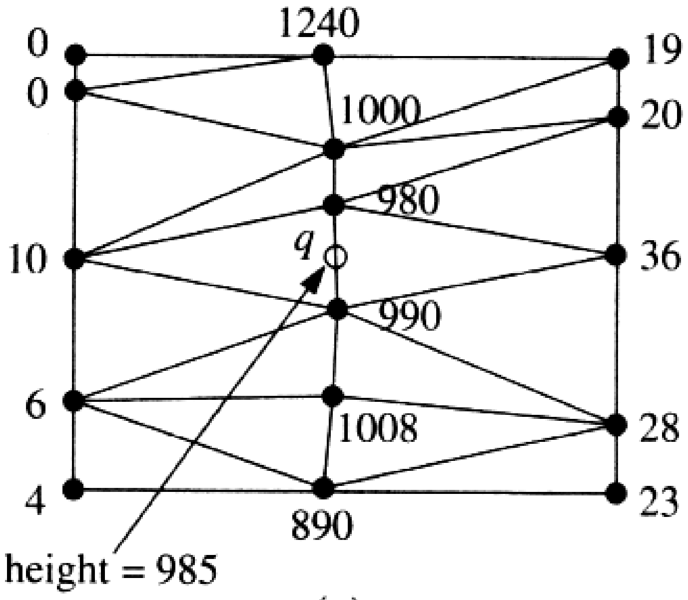


(b)

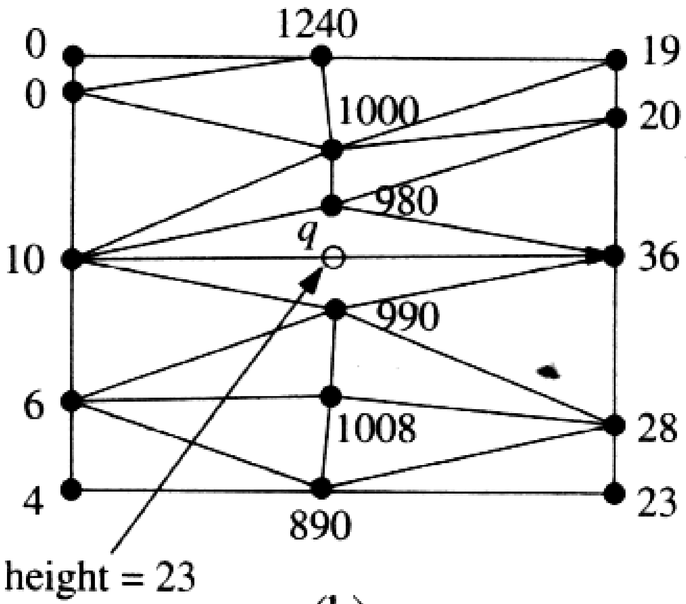
- Which value is $f(q)$ more likely to be?

Delaunay Triangulation in 2D

- Let's revisit Terrain Problem:
  - We believe that some triangulations are "better" than others.
- From the example, we know that:
  - We should avoid skinny triangles, i.e. maximize minimum angle of triangulation



(a)



(b)

Delaunay Triangulation in 2D

- Let's give a definition: **Angle Optimal Triangulations**
- Create **angle vector** $A(T)$ of the sorted angles of triangulation T:

$$A(T) = (\alpha_1, \alpha_2, \dots, \alpha_{3m}), \qquad \alpha_1 \le \alpha_2 \le \cdots \le \alpha_{3m}$$

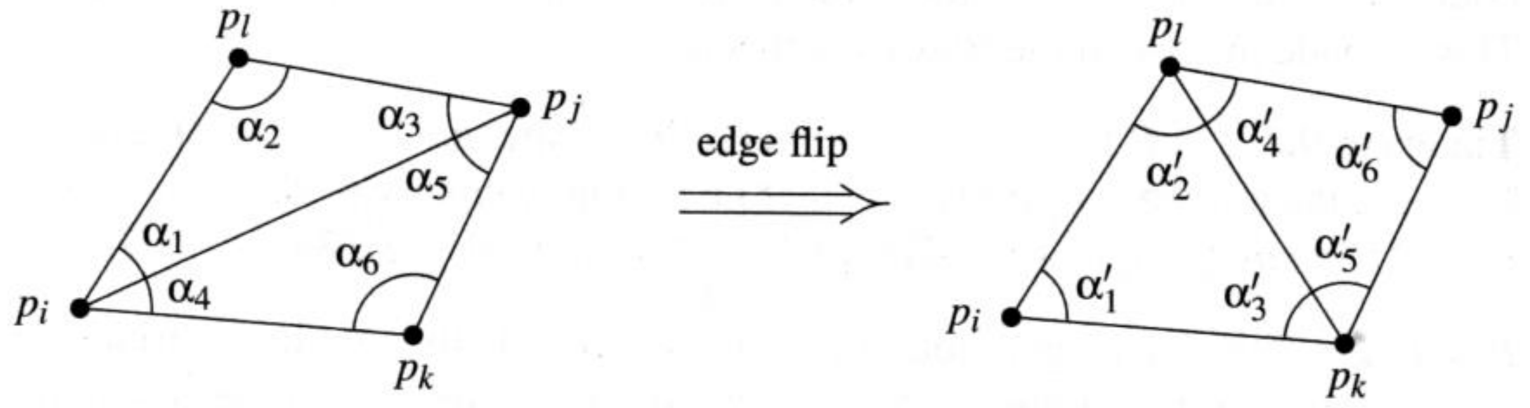
- $A(T)$ is larger than $A(T')$ if and only if there exists an i such that:

$$\alpha_j = \alpha'_j, \forall j < i \text{ and } \alpha_i > \alpha'_i$$

- Best triangulation is triangulation that is **angle optimal**, i.e. has the largest angle vector (maximizes minimum angle).

Delaunay Triangulation in 2D

- For example, consider two adjacent triangles of T:



- If the two triangles form a convex quadrilateral, we could have an alternative triangulation by performing an **edge flip** on their shared edge.
- And, if:

$$\min_{1 \le i \le 6} \alpha_i < \min_{1 \le i \le 6} \alpha'_i$$

- We can get a **better** triangulation by such an edge flip.
- And we call such an edge $p_i p_j$ by **illegal edge**.

Delaunay Triangulation in 2D

- By the idea of “edge flip”, we can give a naïve algorithm to get a “best” triangulation:
- 1. Compute a triangulation of input points P.
- 2. Flip illegal edges of this triangulation until all edges are legal.

- Algorithm will terminate because there is a finite number of triangulations.
- The final “best” triangulation can be called **Delaunay Triangulation**.
- Delaunay Triangulation has many interesting properties including maximizing minimum angle.
- The **Delaunay triangulation** is the straight-line dual of the **Voronoi Diagram**. Note: The Delaunay edges don't have to cross their Voronoi duals.

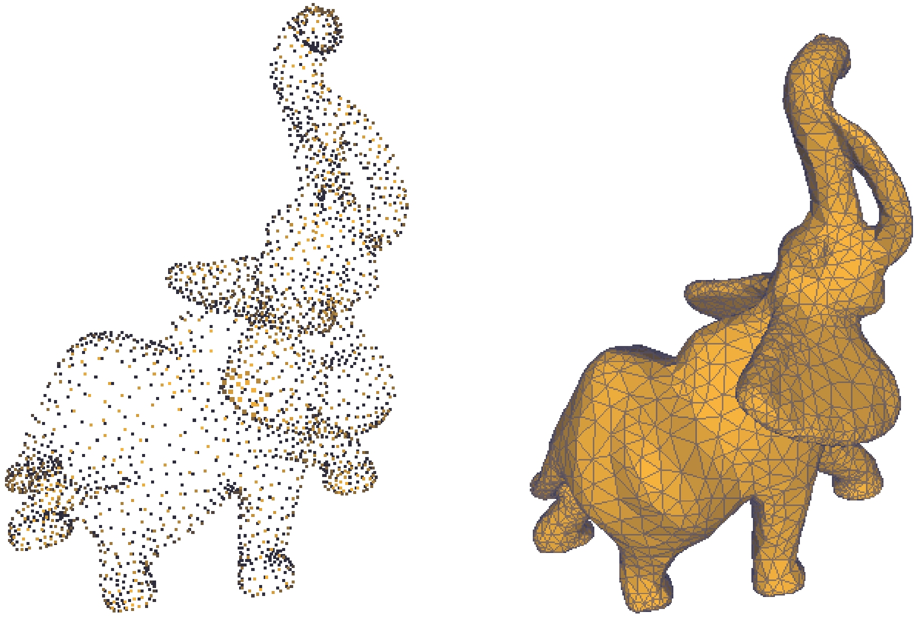
Delaunay Triangulation in 2D

- However, such a naïve algorithm is too slow.
- There is a series of developed methods to perform efficient Delaunay Triangulation using divide and conquer algorithm.
- Here is some 2D visualization demo:
  - https://cartography-playground.gitlab.io/playgrounds/triangulation-delaunay-voronoi-diagram/
  - https://travellermap.com/tmp/delaunay.htm

- Most importantly, it can be extended to arbitrary dimensions (3D or dD).

Delaunay Triangulation in 3D

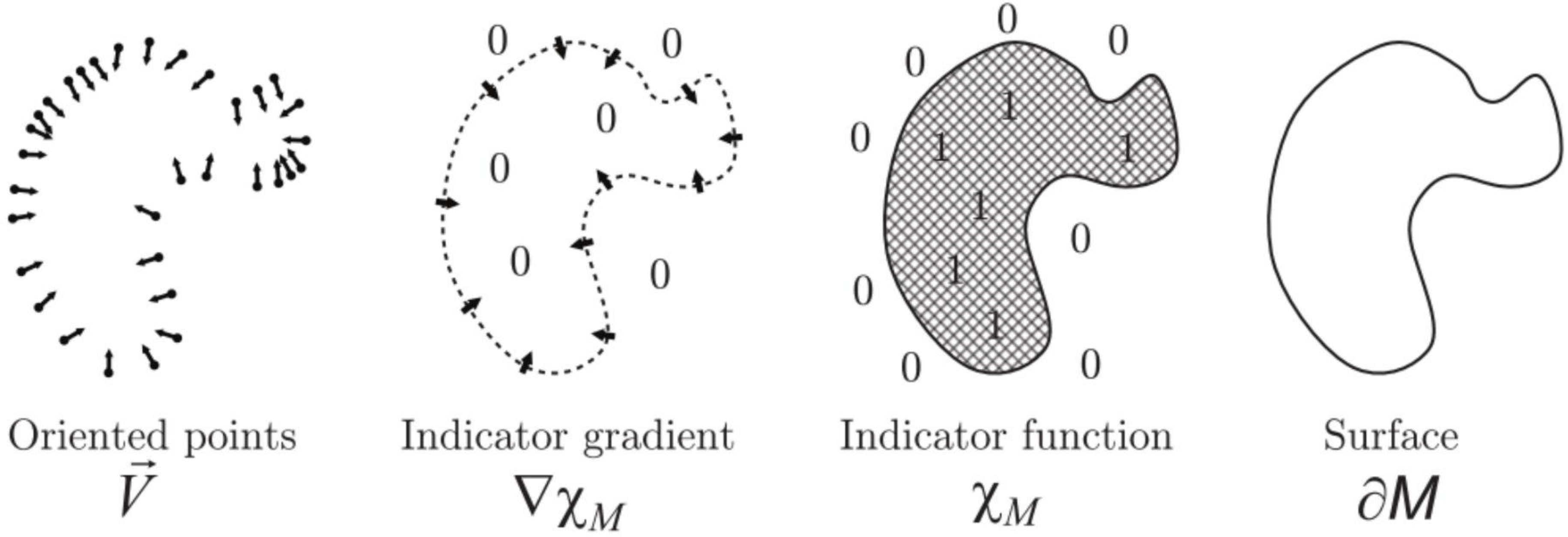
- Extend to a 3D point cloud: set of points $A \subset R^3$
- Attribute $f(p)$ defined at each point p in A
  - The attribute can be color, UV, depth (from a certain view), …
- How to know the attribute of points not in A?
  - The point cloud is sparse.
  - But we want a continuous $f(p)$.
- Option: Bilinear Interpolation!
  - Via triangulation in 3D
  - That's why the **Mesh** is such a powerful 3D representation



Courtesy: Jiju Peethambaran and Ramanathan Muthuganapathy

Poisson Surface Reconstruction

- The input is oriented points $\vec{V}$ (points with normals) sampled from a shape M.
- Poisson Surface Reconstruction construct an implicit surface first.
- χ_M is indicator function of M.
- We want to get $\nabla\chi_M$ to represent the surface ∂M



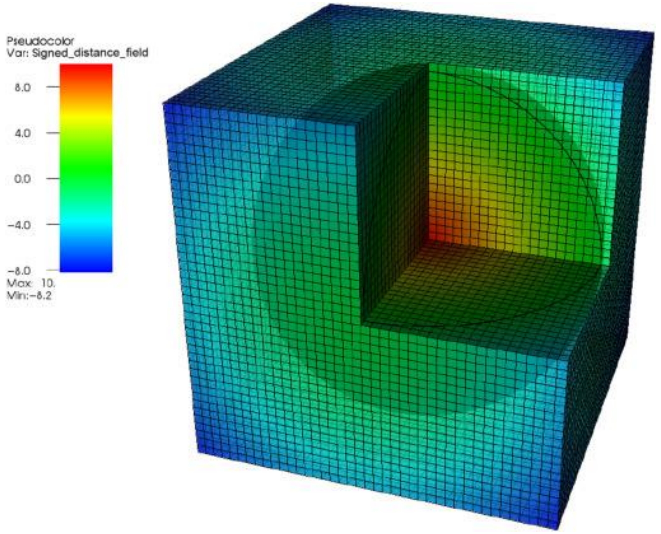
Courtesy: Kazhdan et al. Poisson surface reconstruction. SGP 2006.

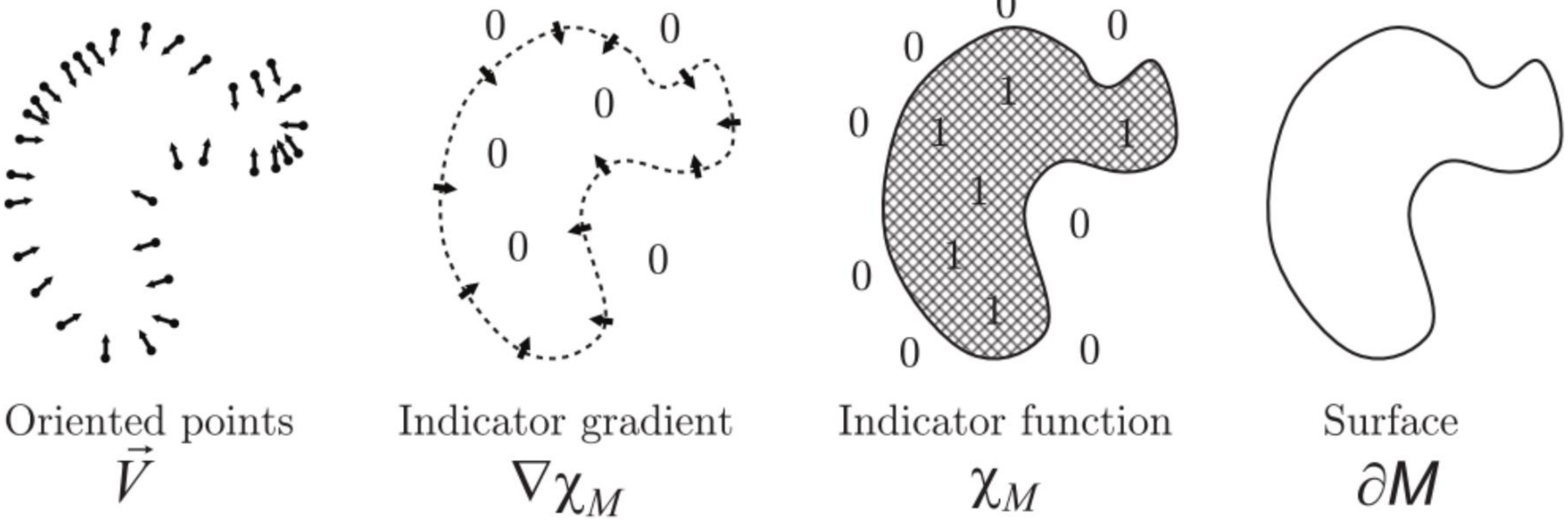
Poisson Surface Reconstruction

- According to Possion Equation:

$$\nabla \cdot (\nabla \chi) = \nabla \cdot \vec{V} \Leftrightarrow \Delta \chi = \nabla \cdot \vec{V}$$

- For 3D, we can construct an octree and solve it by PDE
- Then we get the implicit representation of surface.

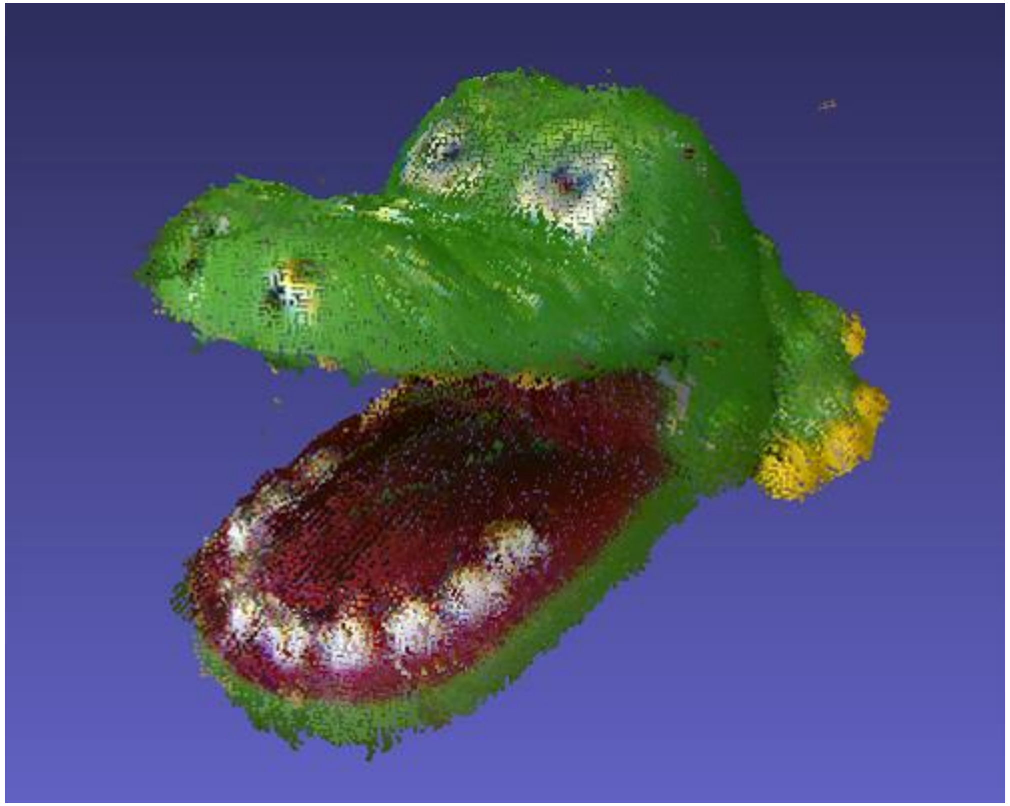




Courtesy: Kazhdan et al. Poisson surface reconstruction. SGP 2006.

Poisson Surface Reconstruction

- Finally we get the triangle mesh by Marching Cubes.
- An example:



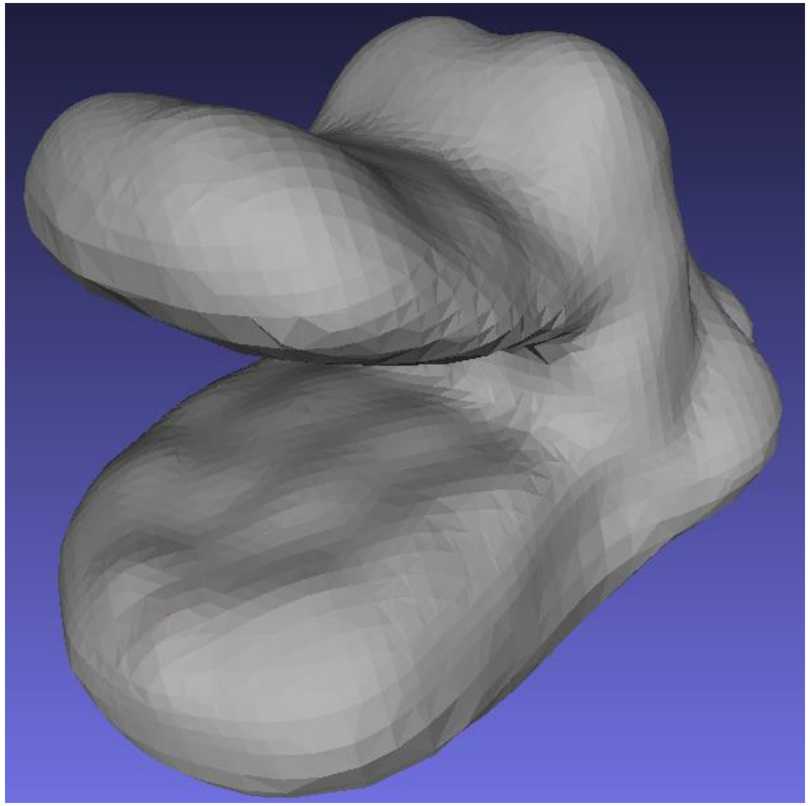
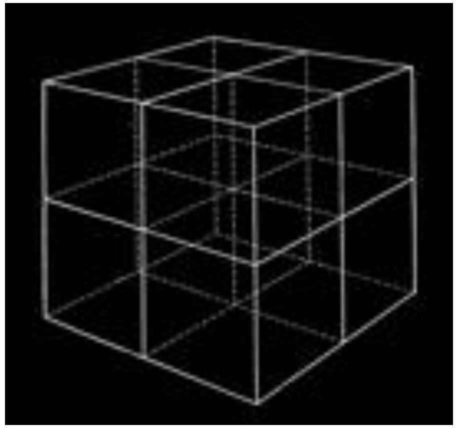


Figure: (a) On the left is the point cloud. (b) On the right is the reconstructed mesh.



Marching Cubes:

A High Resolution 3D Surface Construction Algorithm

William E. Lorenson

Harvey E. Cline

General Electric Company

Corporate Research and Development, SIGGRAPH 1987

- W. Lorensen and H. Cline. “Marching cubes: A High Resolution 3D Surface Construction Algorithm”, Proceedings of SIGGRAPH 1987, pages 163-169, 1987.

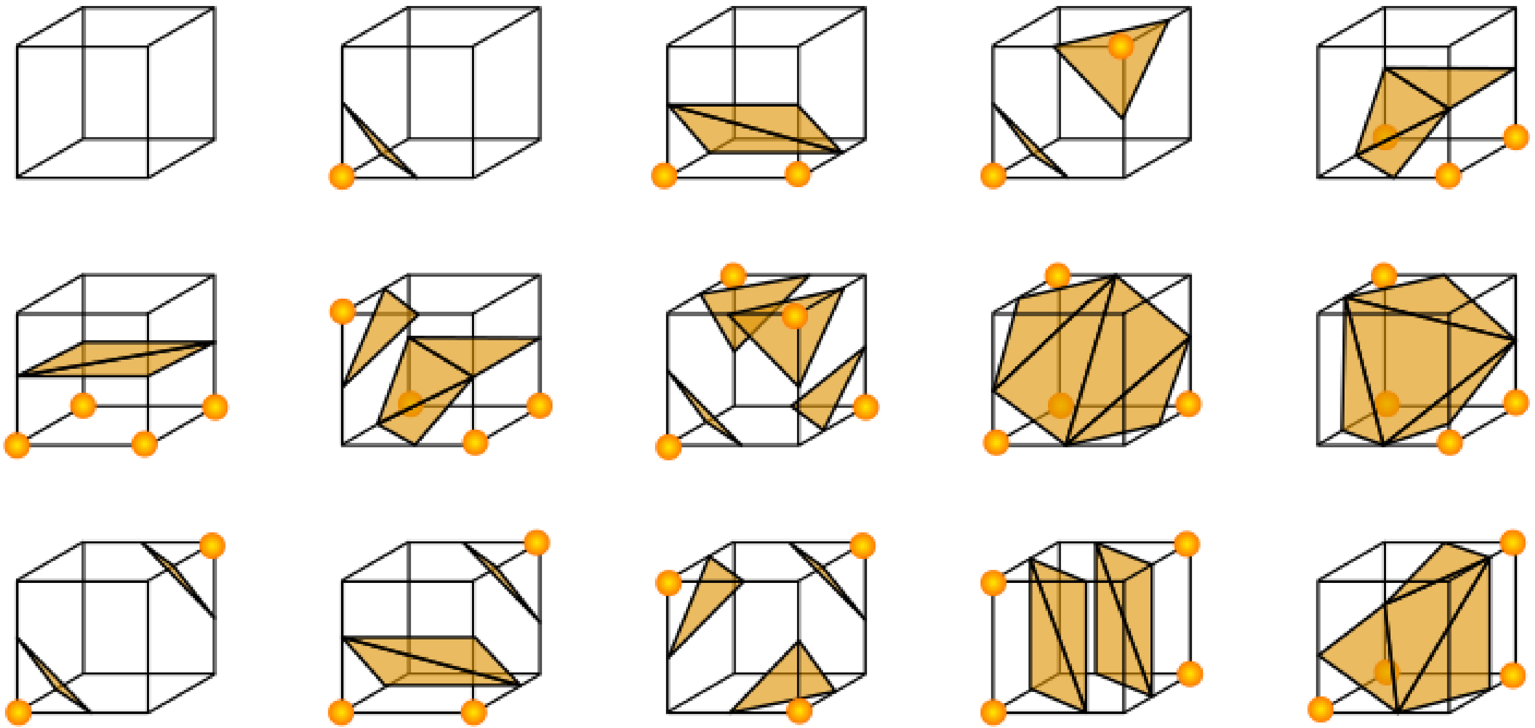
- The Visible Human: http://www.nlm.nih.gov/research/visible/visible_human.html
- Marching Cubes Demo/Tutorial: http://users.polytech.unice.fr/~lingrand/MarchingCubes/accueil.html

Step 1: Surface Intersection

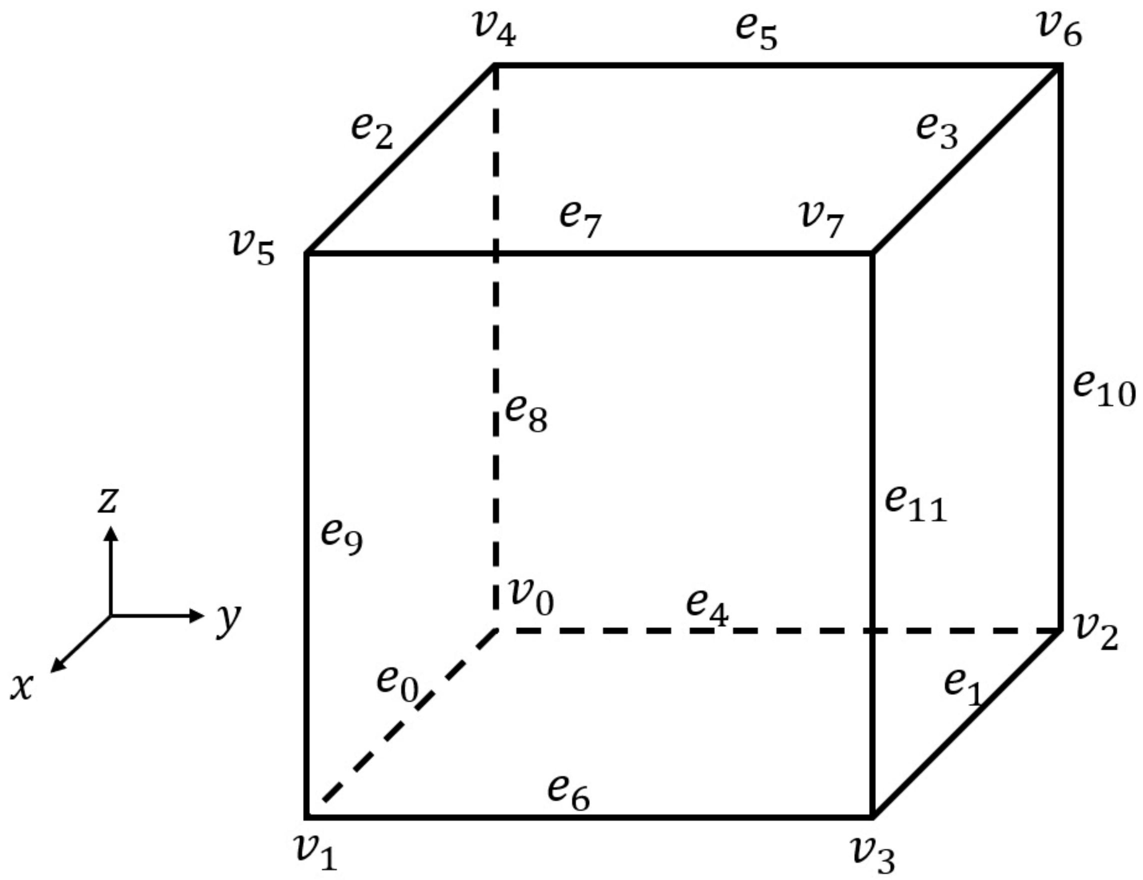
- Given user specified value T_U, binary vertex assignment: $(p(i,j,k) \geq T_U ? 1 : 0)$
  - Set cube vertex to value of 1 if the data value at that vertex exceeds (or equals) the value of the surface we are constructing
  - Otherwise, set cube vertex to 0
- If a vertex = 1 then it is "inside" the surface
- If a vertex = 0 then it is "outside"
- Any cube with vertices of both types is "intersected" by the surface.

Step 2 : Triangulation

- For each cube, we have 8 vertices with 2 possible states each (inside or outside).
- This gives us 2^8 possible patterns = 256 cases.
- Enumerate cases to create a LUT (LookUp Table)
- Use symmetries to reduce problem from 256 to 15 cases.



Step 2 : Triangulation

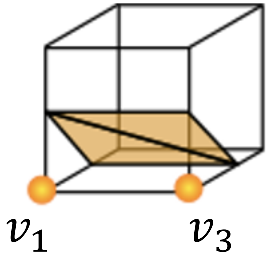


Vertex bit mask:

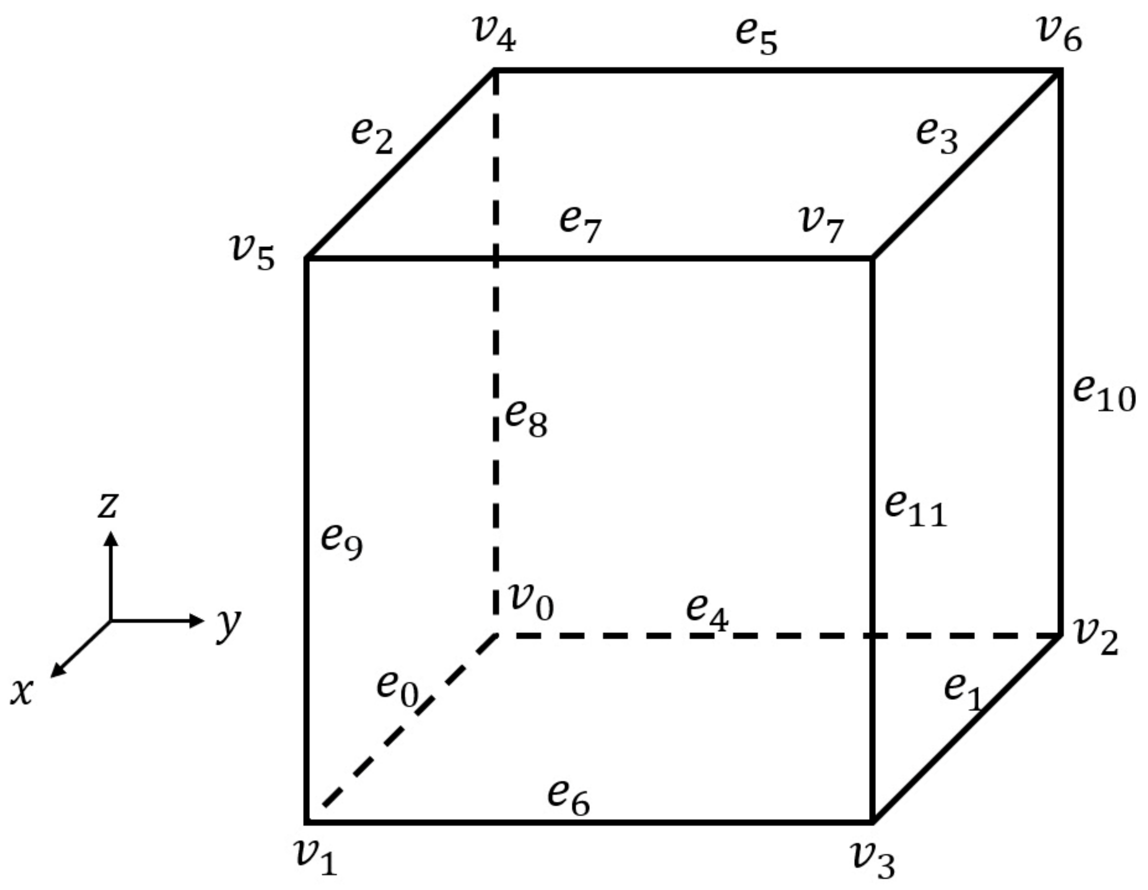
v_7	**v_6**	**v_5**	**v_4**	**v_3**	**v_2**	**v_1**	**v_0**

- 2.1 Use vertex bit mask to create an index for each case based on the state of the vertexes.

- Example:
- If v1 and v3 are “outside” the surface and thus our index would be 245 (1111-0101 for binary).



Step 2 : Triangulation

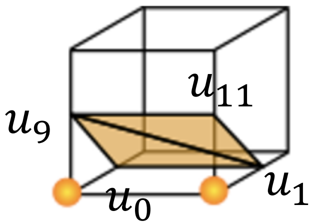


Vertex bit mask:	v_7	v_6	v_5	v_4	v_3	v_2	v_1	v_0

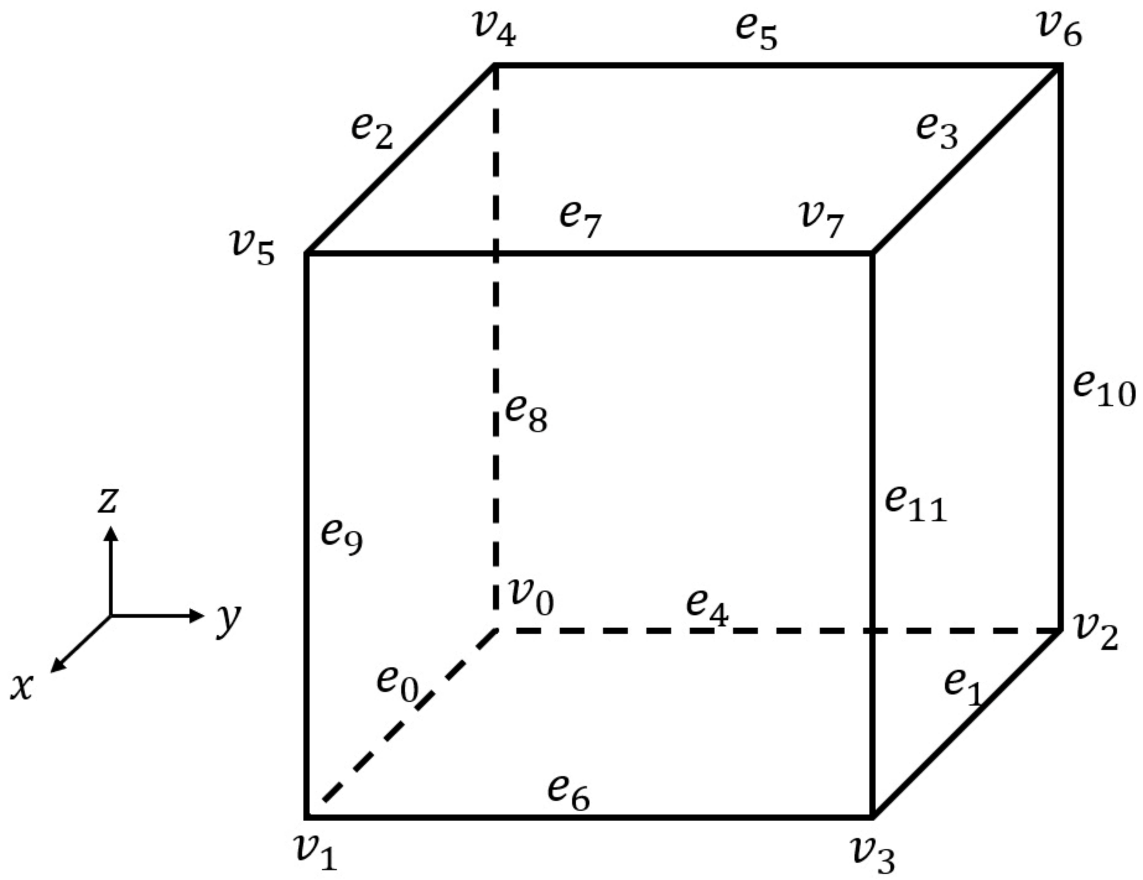
Edge State:	e_{11}	e_{10}	e_9	e_8	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0

- 2.2 Using the index to tell which edge the surface intersects, we can then linearly interpolate the surface intersection along the edge.

- Example:
- We search edge state by index 245, getting code 2563 (1010-0000-0011 for binary) which means e11, e9, e1 and e0 are intersected by the surface.
- And we insert intersection vertices on e11, e9, e1 and e0, generating u11, u9, u1 and u0.



Step 2 : Triangulation

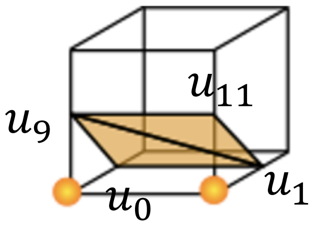


Vertex bit mask:	v_7	v_6	v_5	v_4	v_3	v_2	v_1	v_0				
Edge State:	e_{11}	e_{10}	e_9	e_8	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0

Edge Order:	o_0, o_1, o_2	o_3, o_4, o_5	…

- 2.3 Using the index to tell how the intersection vertices are connected to form a triangle facet.

- Example:
- And we insert intersection vertices on e11, e9, e1 and e0, generating u11, u9, u1 and u0.
- Finally we search edge order by index 245, getting array [[0, 1, 11], [9, 0, 11]], which means there are two triangle facets to generate: u0-u1-u11 and u9-u0-u11.



Step 3 : Surface normals

- To calculate surface normal, we need to determine gradient vector $\vec{G}$ (derivative of the density function).
- To estimate the gradient vector at the surface of interest, we first estimate the gradient vectors at the vertices and interpolate the gradient at the intersection.
- The gradient at cube vertex (i, j, k), is estimated using central differences along the three coordinate axes by:

$$G_X(i,j,k) = \frac{D(i+1,j,k) - D(i-1,j,k)}{\Delta x}$$

$$G_Y(i,j,k) = \frac{D(i,j+1,k) - D(i,j-1,k)}{\Delta y}$$

$$G_Z(i,j,k) = \frac{D(i,j,k+1) - D(i,j,k-1)}{\Delta z}$$

D(i,j,k) is the density at pixel (i, j) in slice k.

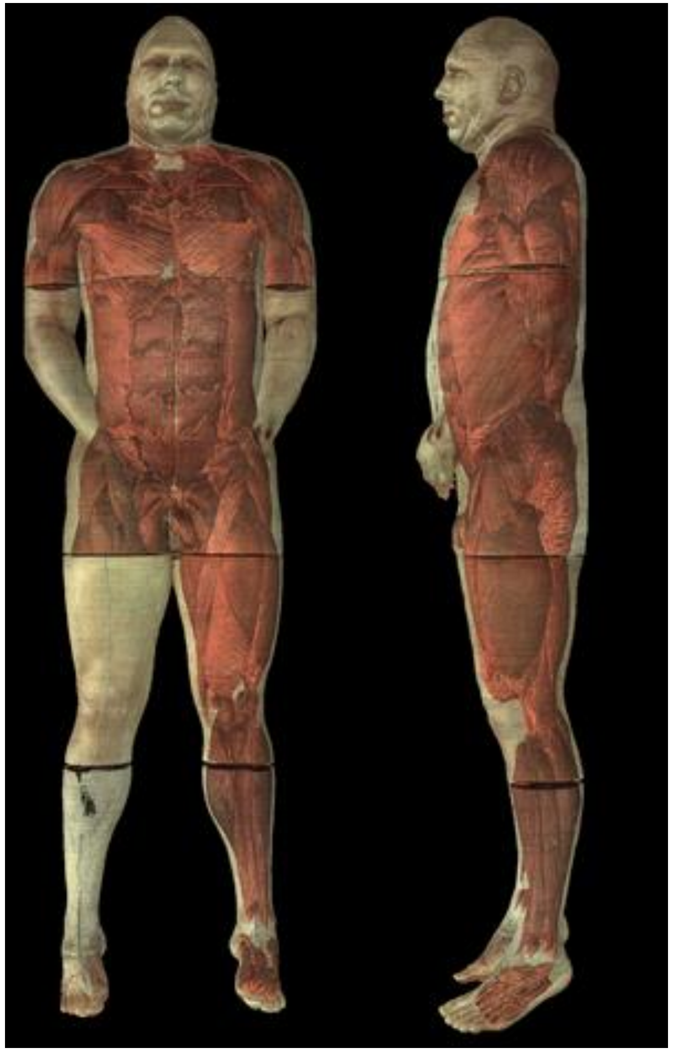
Δx, Δy, Δz are lengths of the cube edges

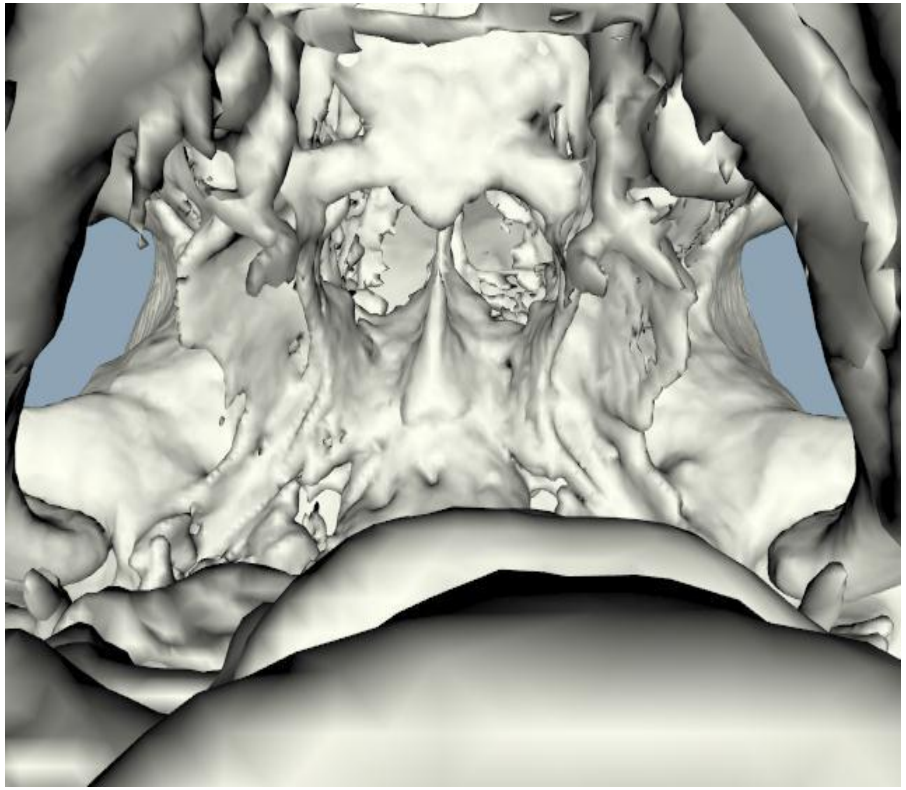
Step 3 : Surface normals

- Dividing the gradient by its length produces the unit normal at the vertex required for rendering.
- Then the algorithm linearly interpolates this normal to the point of intersection.

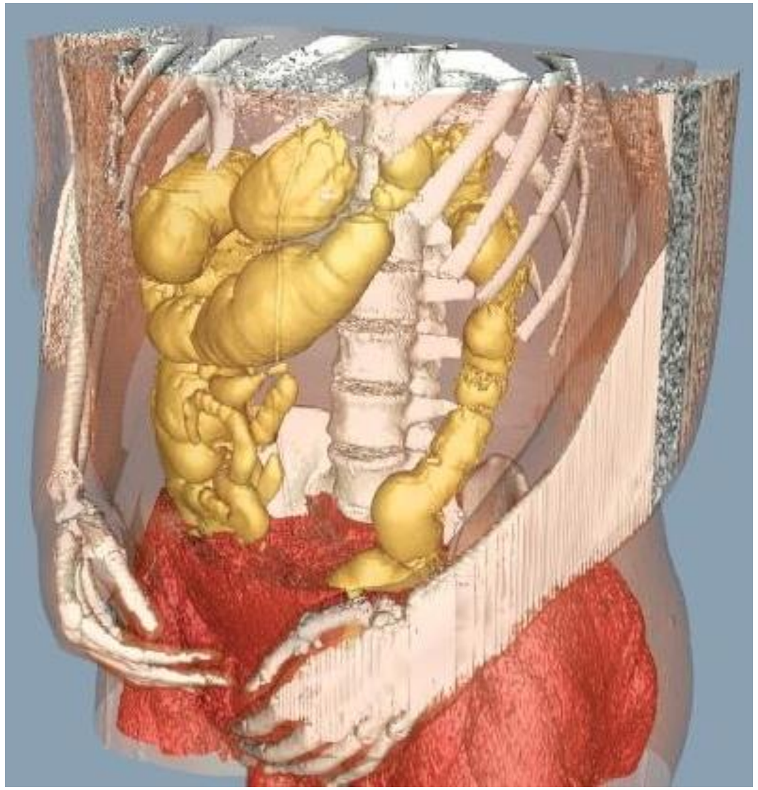
$$\vec{N}(i,j,k) = \frac{\vec{G}(i,j,k)}{\left\|\vec{G}(i,j,k)\right\|_2}$$

Results –The Visible Man



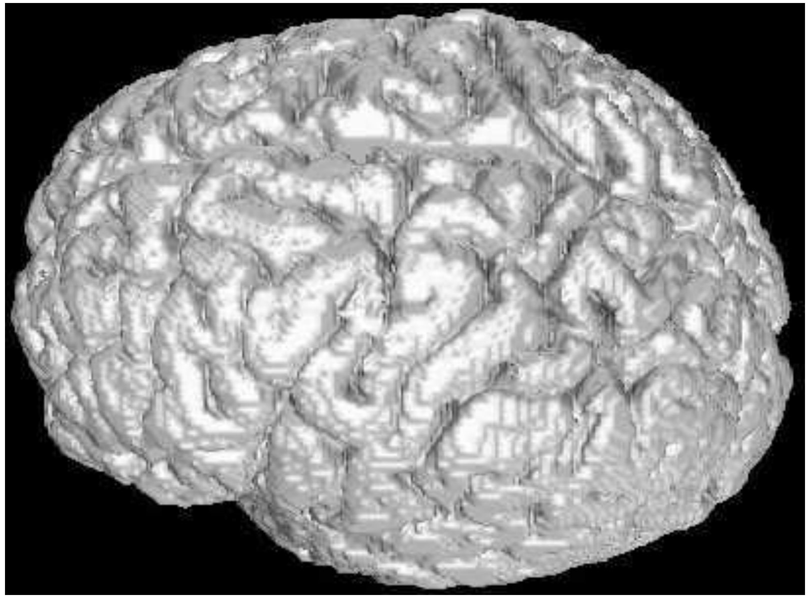


Inside skeleton view

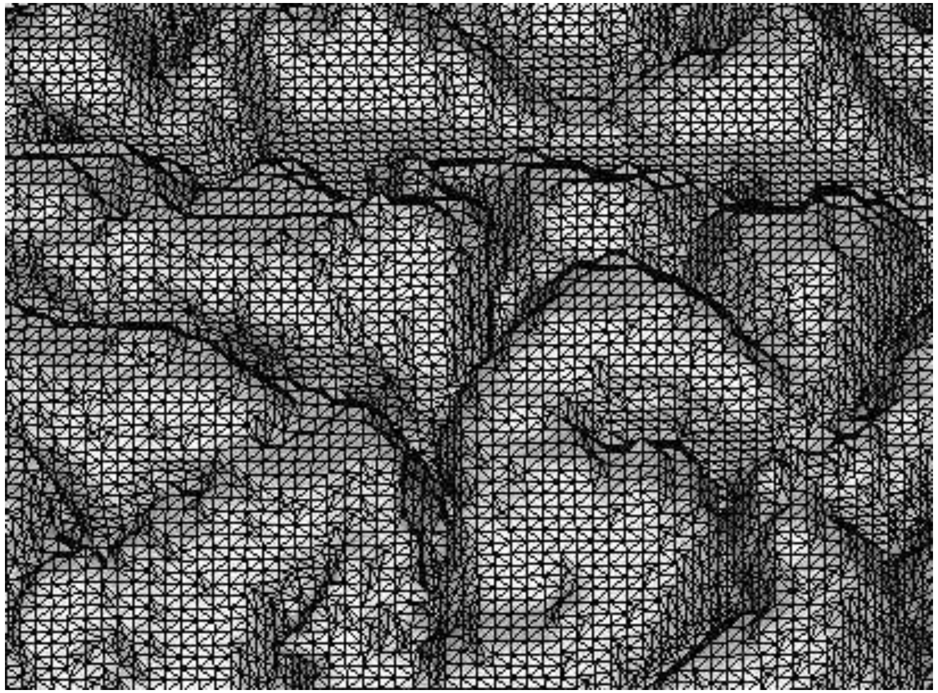


Torso / bowels

Results



Human brain surface reconstructed by using marching cubes (128,984 vertices and 258,004 triangles



Magnified display of brain surface

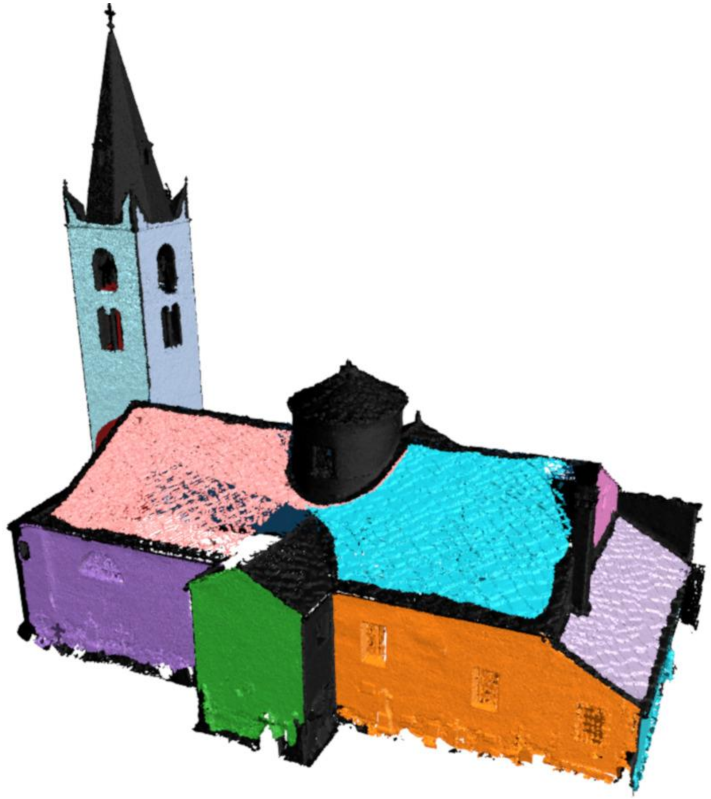
Algorithm Summary

1. Scan 2 slices and create cube
2. Calculate index for cube based on vertices
3. Use index to lookup list of edges intersected
4. Use densities to interpolate edge intersections
5. Calculate unit normal at each edge vertex using central differences. Interpolate normal to each triangle vertex
6. Output the triangle vertices and vertex normals
7. March to next position and repeat.

- Enhancements:
  - Take advantage of pixel-to-pixel, line-to-line, and slice-to-slice coherence by keeping previous calculations.
  - Thousands more ….

Model Fitting

- Motivation
  - How do we fit models (i.e., a parametric representation of data that's smaller than the data) to data?
- For example, given such a point cloud:
  - How to find a plane?
  - A 3D plane can be described as:
    - $Ax + By + Cz = D$
  - But how to know the parameters?

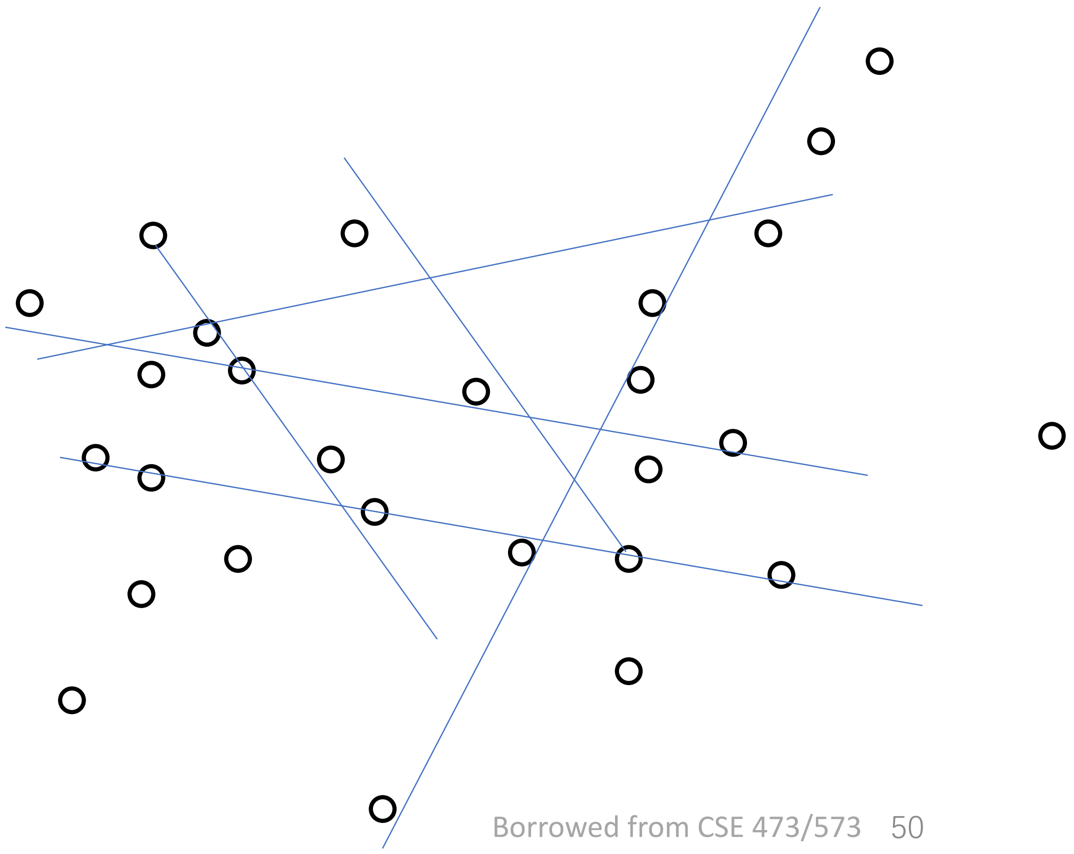


Courtesy: https://github.com/STORM-IRIT/Plane-Detection-Point-Cloud

Figure: the plane detection of the model of a building

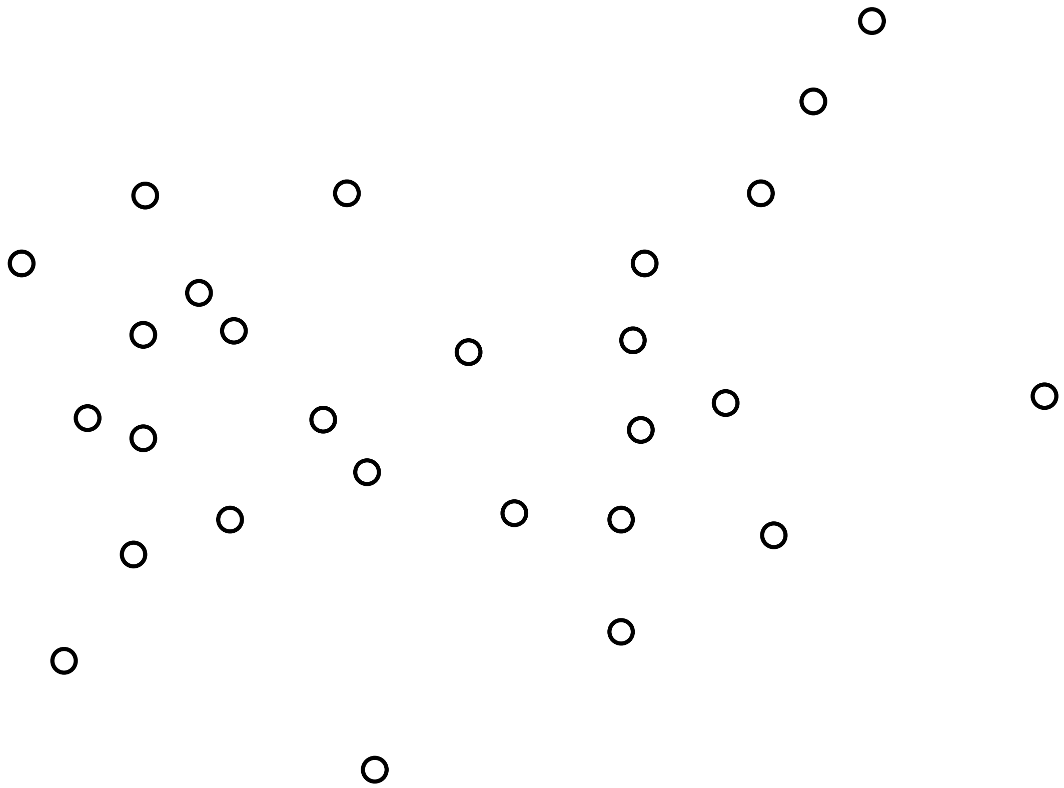
Model Fitting

- Let's consider a simple case: line fitting.
- Given a set of 2D points, how to fit the best possible line to these points?
- Brute Force Search - 2^N possibilities!
- Not feasible.
- Better Strategy?



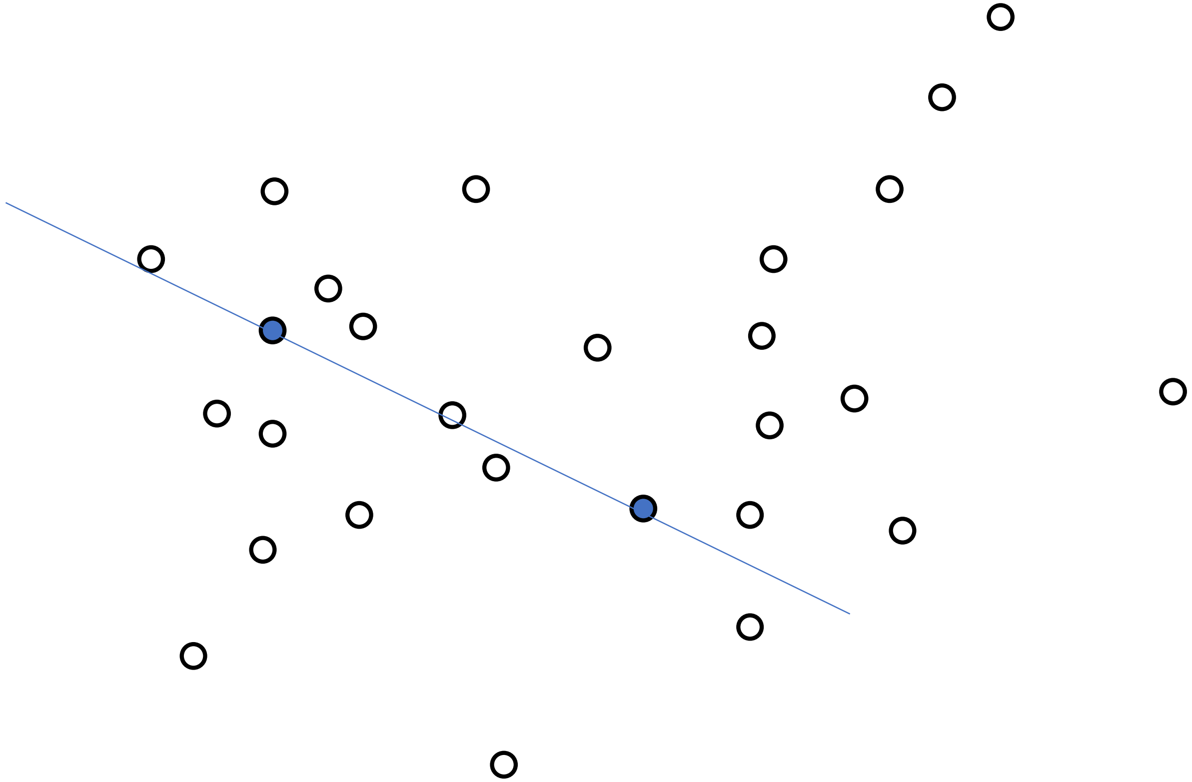
RANSAC

- Random Search – Much Faster!



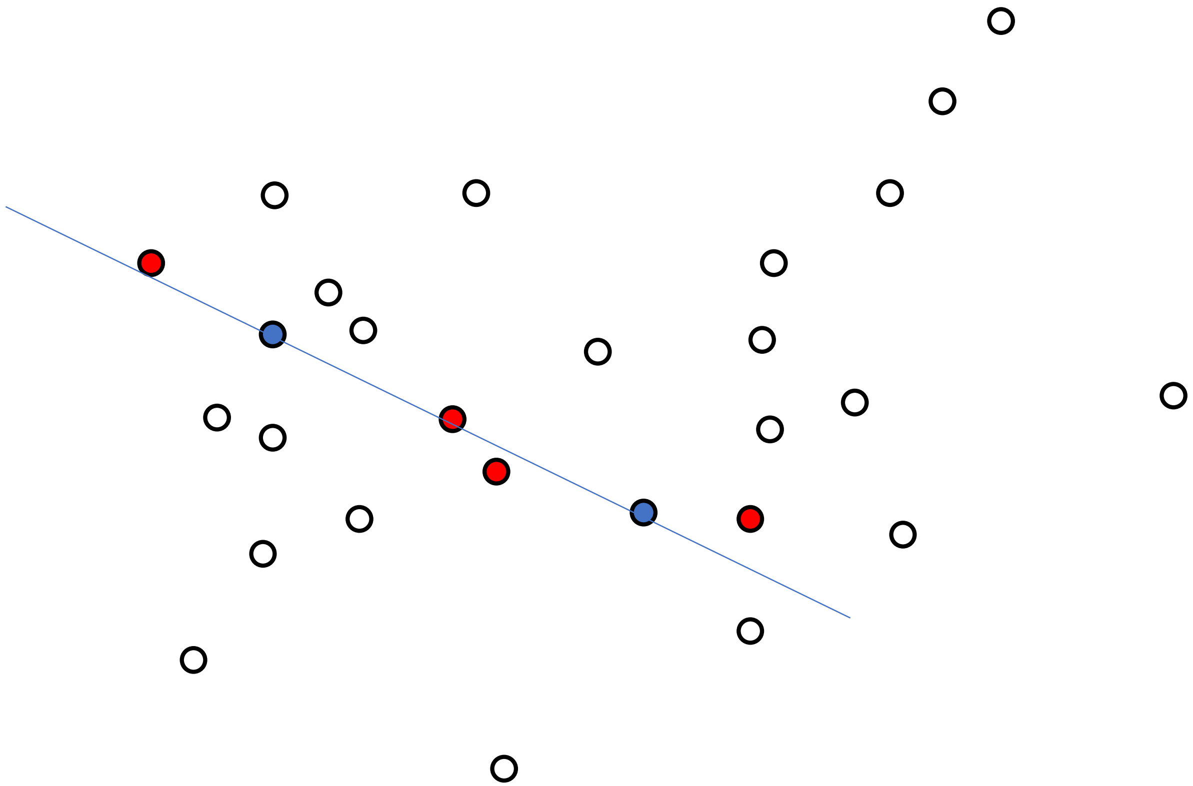
RANSAC

- See how RANSAC works.
- Iteration 1:



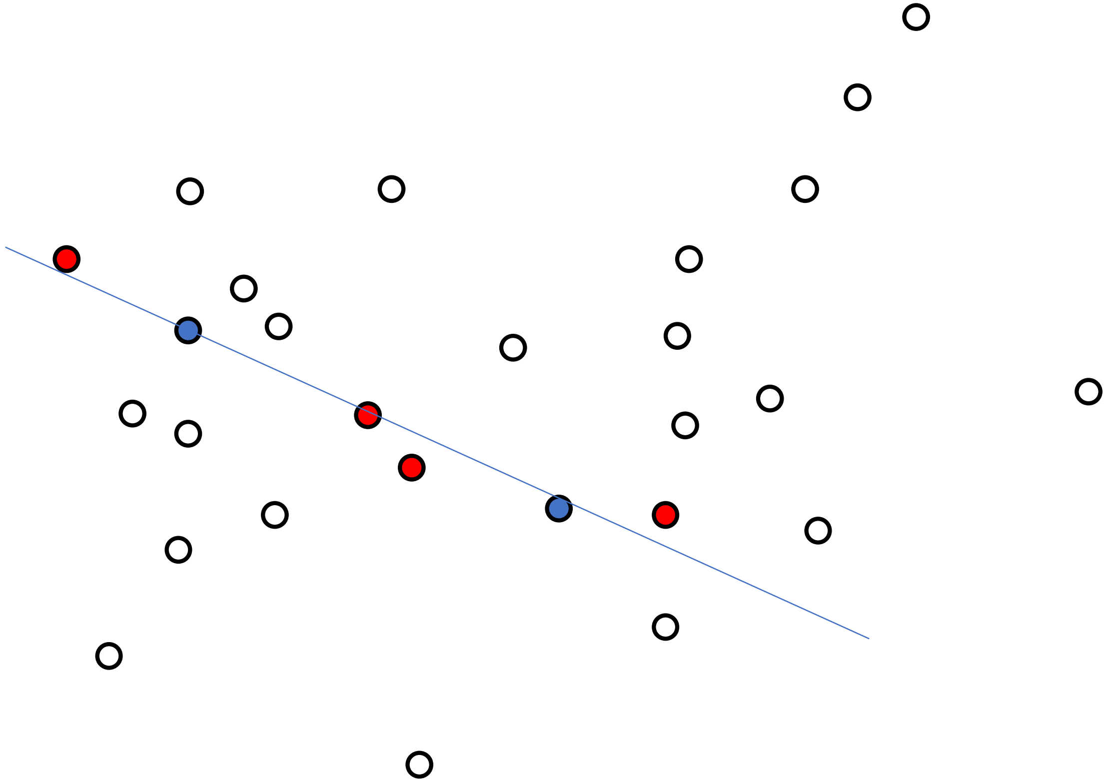
RANSAC

- See how RANSAC works.
- Iteration 1:



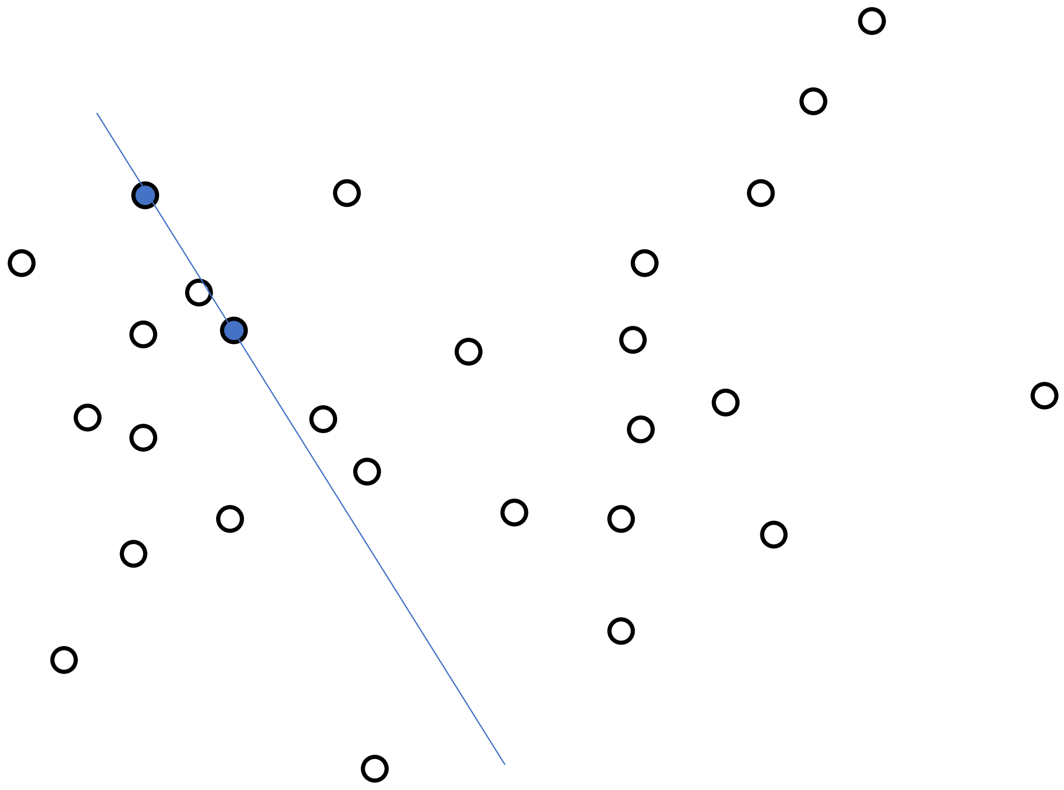
RANSAC

- See how RANSAC works.
- Iteration 1:



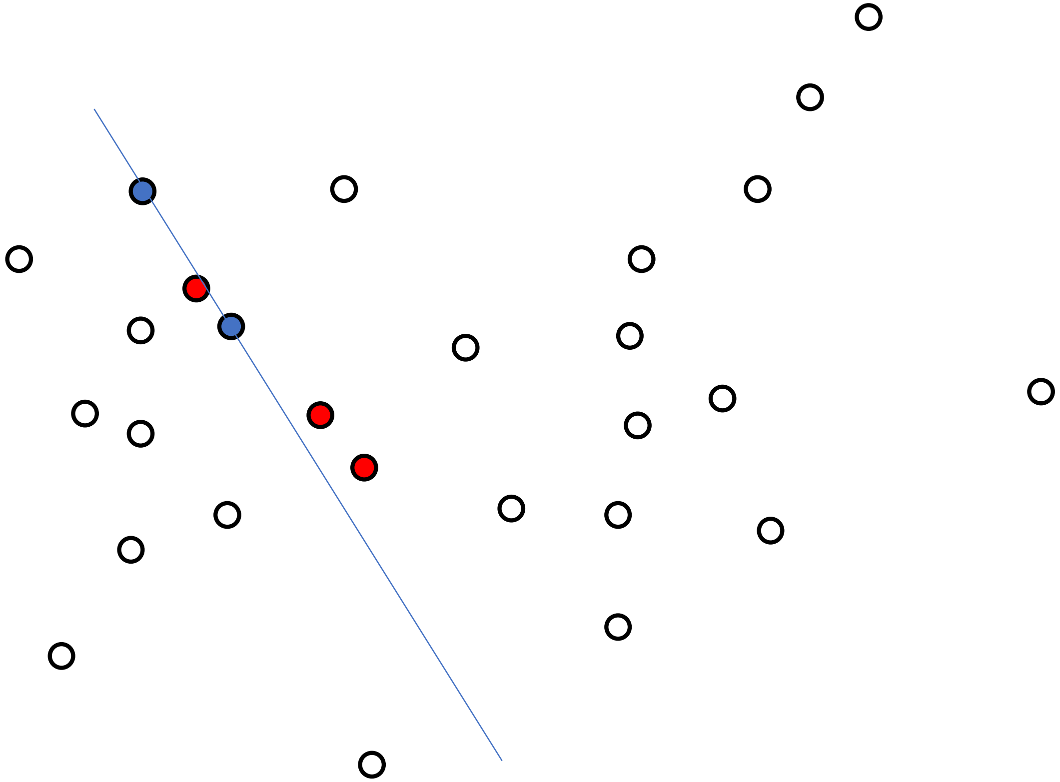
RANSAC

- See how RANSAC works.
- Iteration 2:



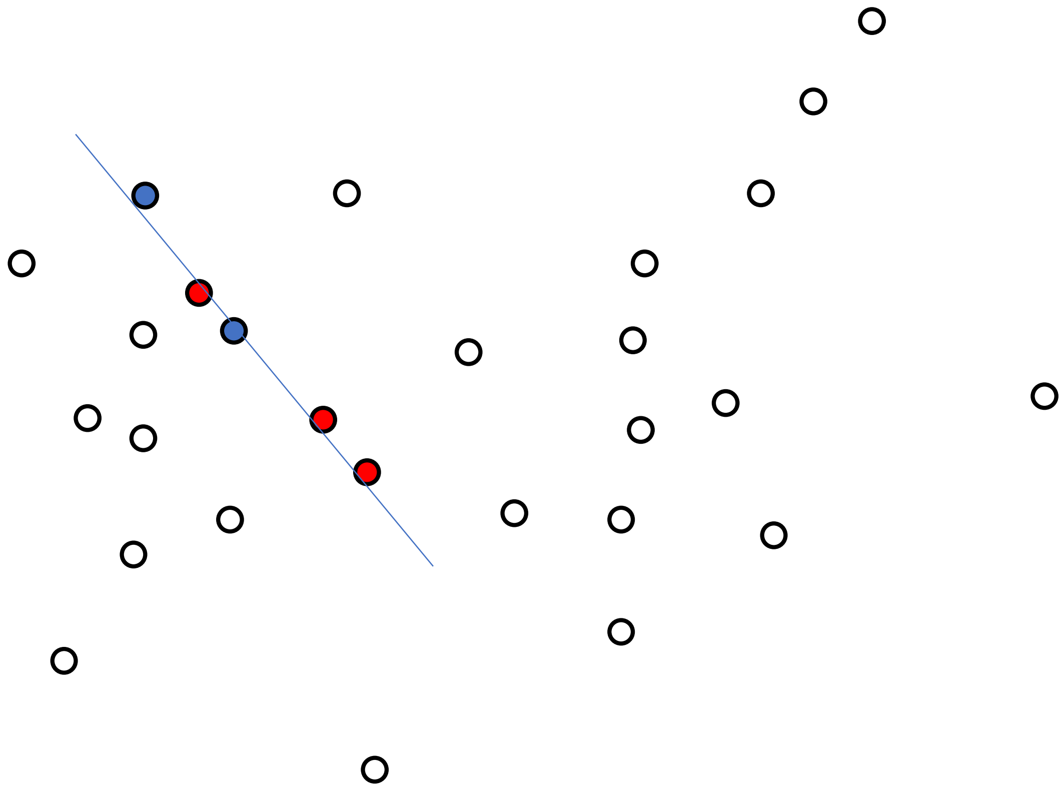
RANSAC

- See how RANSAC works.
- Iteration 2:



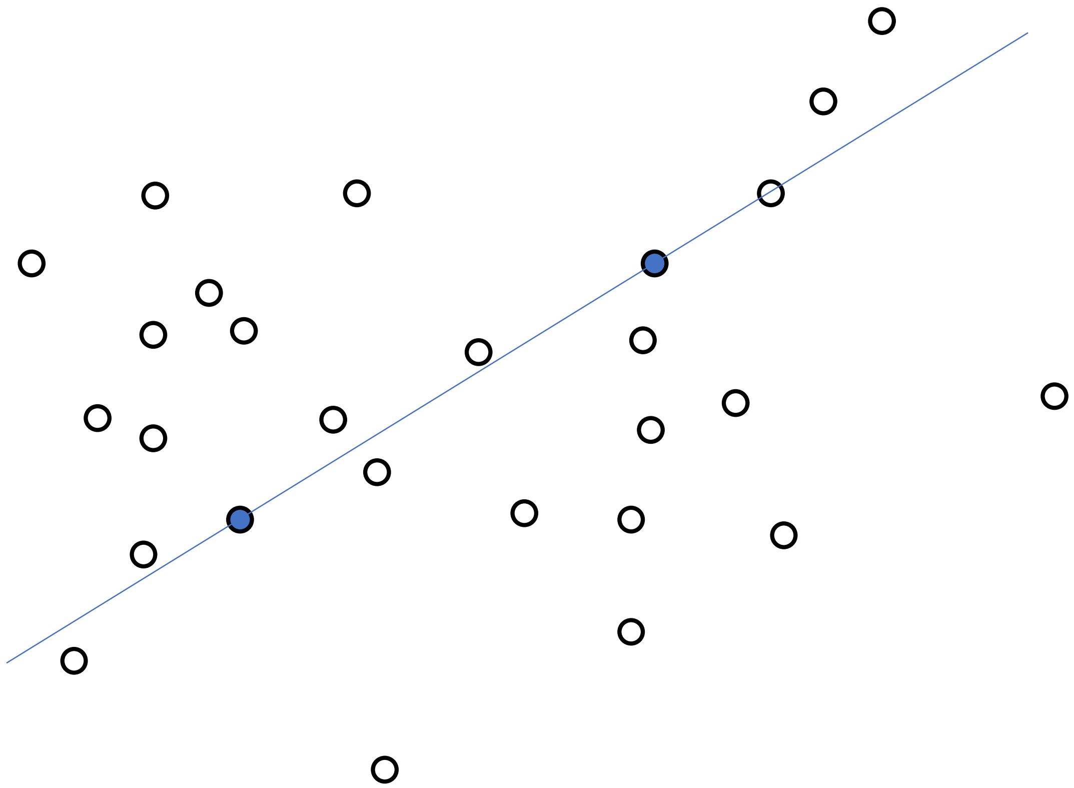
RANSAC

- See how RANSAC works.
- Iteration 2:



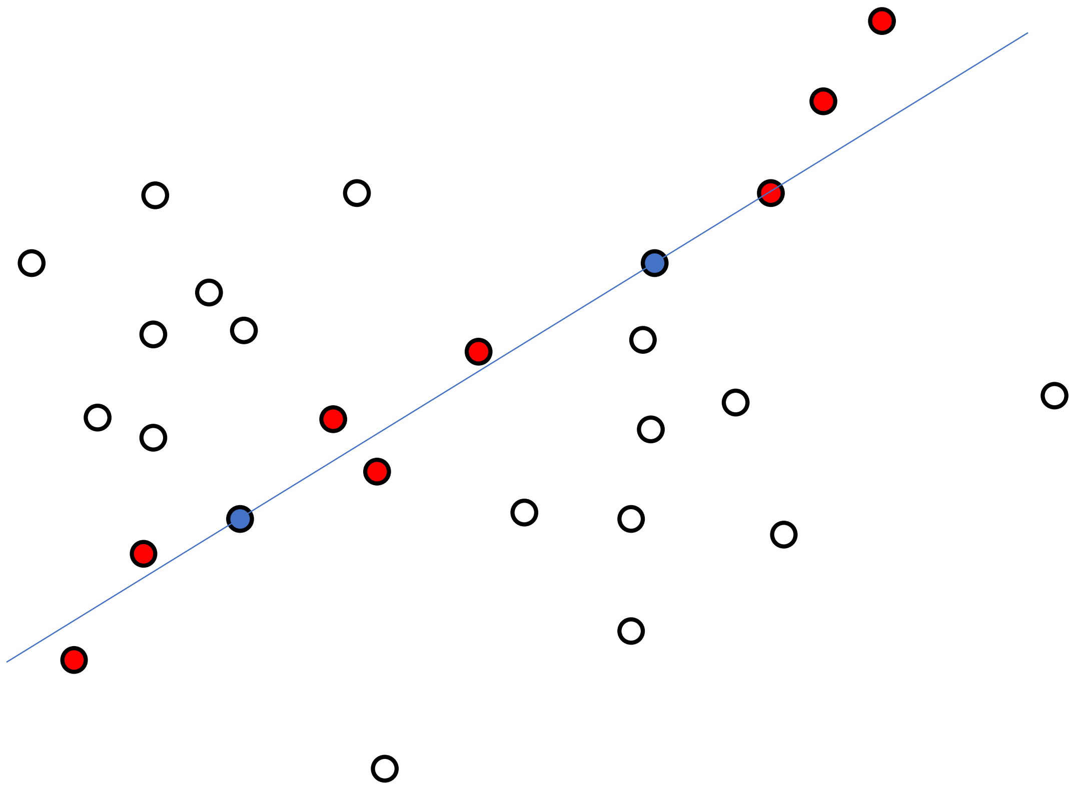
RANSAC

- See how RANSAC works.
- ……
- Iteration 5:



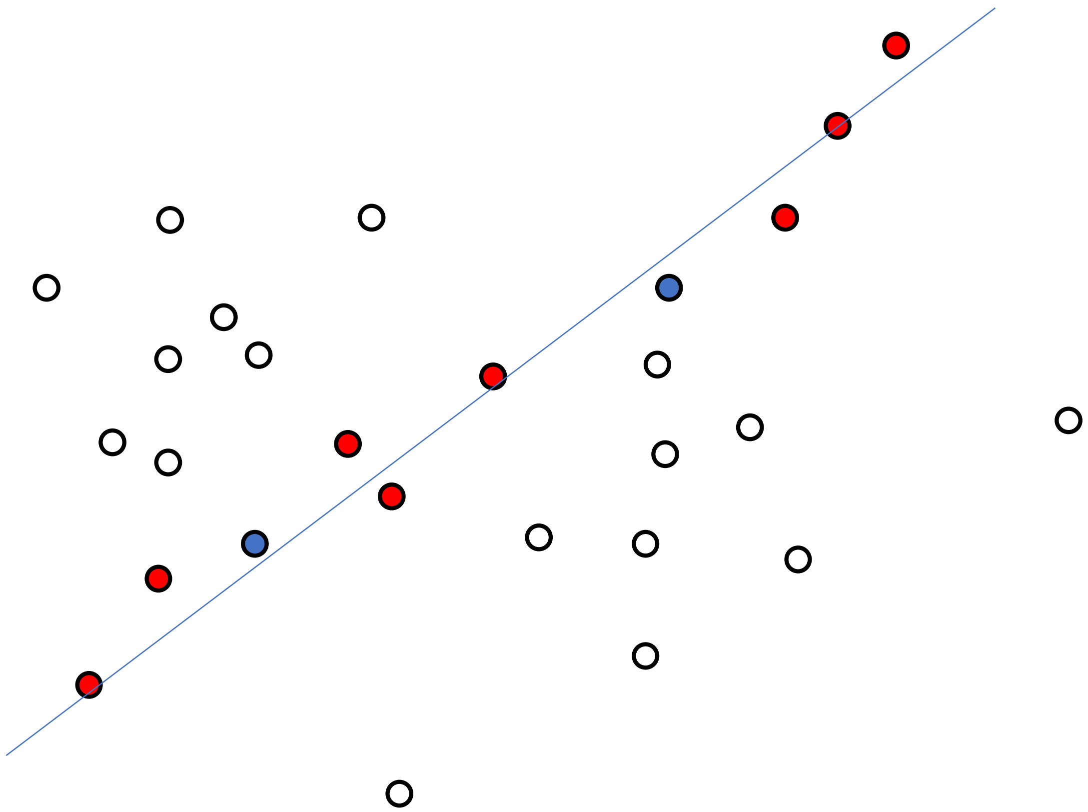
RANSAC

- See how RANSAC works.
- Iteration 5:



RANSAC

- See how RANSAC works.
- Iteration 5:

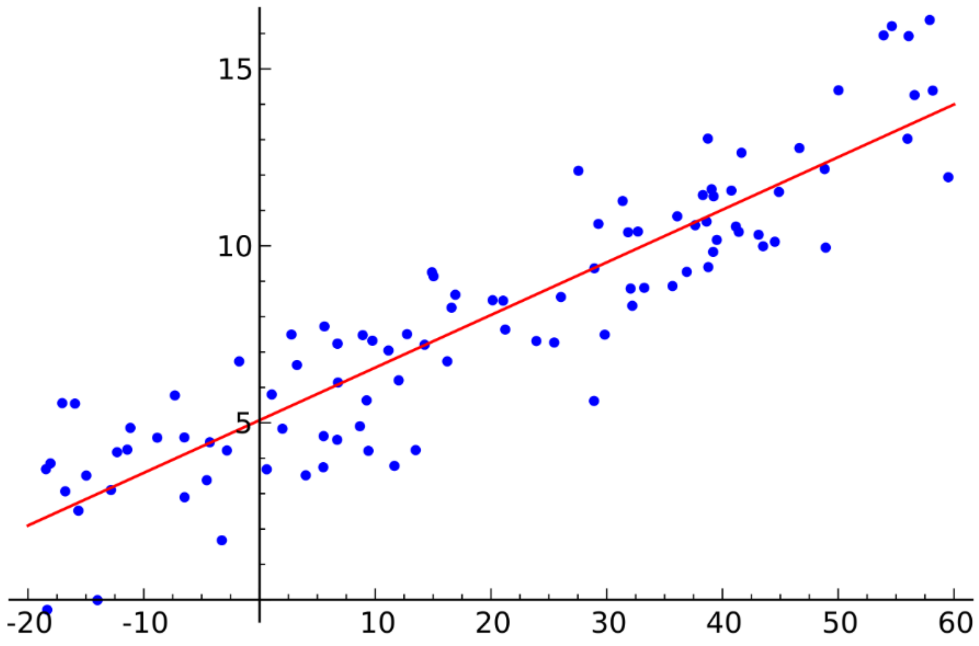


RANSAC

- **Algorithm RANSAC**
- Determine:
  - n – the smallest number of points required (e.g., for lines, $n = 2$, for circles, $n = 3$)
  - k – the number of iterations required
  - t – the threshold used to identify a point that fits well
  - d – the number of nearby points required to assert a model fits well
- Until k iterations have occurred
  - Draw a sample of n points from the data uniformly and at random
  - Fit to that set of n points
  - For each data point outside the sample
    - Test the distance (fitting error) from the point to the structure
    - If the distance is less than t, the point is close (called inlier)
  - If there are d or more points close to the structure
    - Then there is a good fit. Refit the structure using all inliers. Add the result to a collection of good fits.
- Use the best fit from this collection (using the fitting error as a criterion)

RANSAC

- RANSAC will distinguish inliers with outliers.
- A detail: how to fit multiple inliers after RANSAC?
  - Least Squares



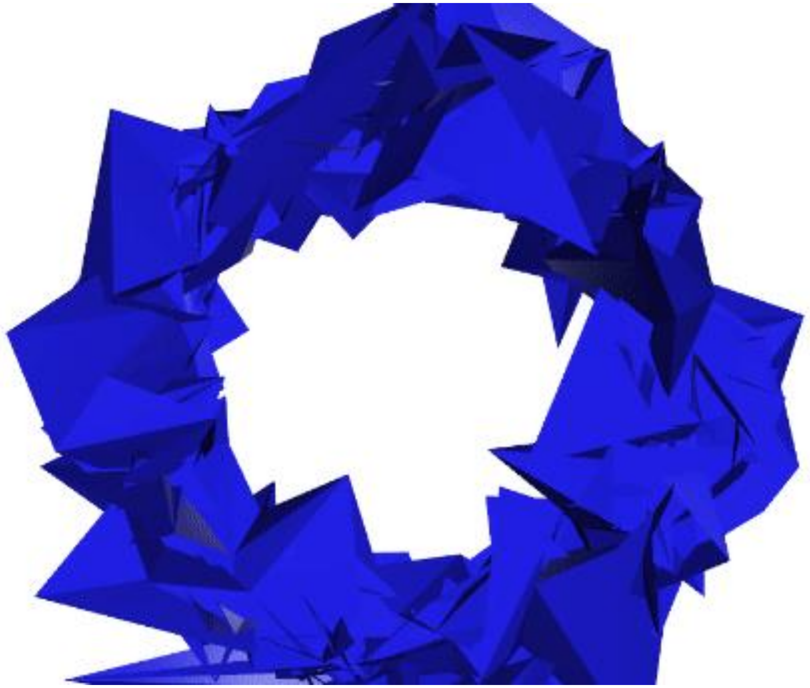
Use Least-Squares to fit a line with much more points than minimal need (2 points)

Model Fitting

- Now we can describe the general process of model fitting:
- 1. Give a parametric model (e.g. plane, line, sphere)
- 2. Use RANSAC to find inliers from a point cloud
- 3. Use Least-Squares to fit the model to the inliers
- 4. Take the parameters and the inliers as output

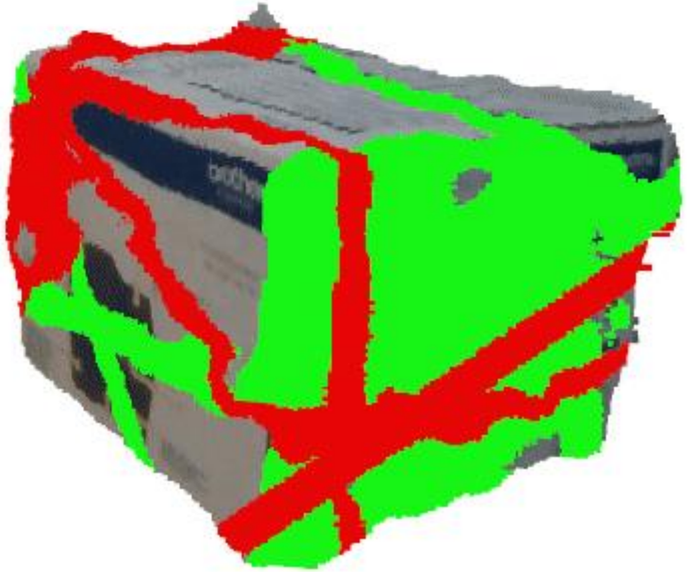
Model Fitting

- Visualization: https://github.com/leomariga/pyRANSAC-3D/tree/Animations
  - Red: current trials
  - Green: currently best trial
  - Circle:



Model Fitting

- Visualization: https://github.com/leomariga/pyRANSAC-3D/tree/Animations
  - Red: current trials
  - Green: currently best trial
  - Cuboid:



Summary

- Registration
  - PCA
  - SVD
  - ICP
- Surface Reconstruction
  - Delaunay Triangulation
  - Poisson Surface Reconstruction
- Model Fitting
  - RANSAC
  - Least Squares