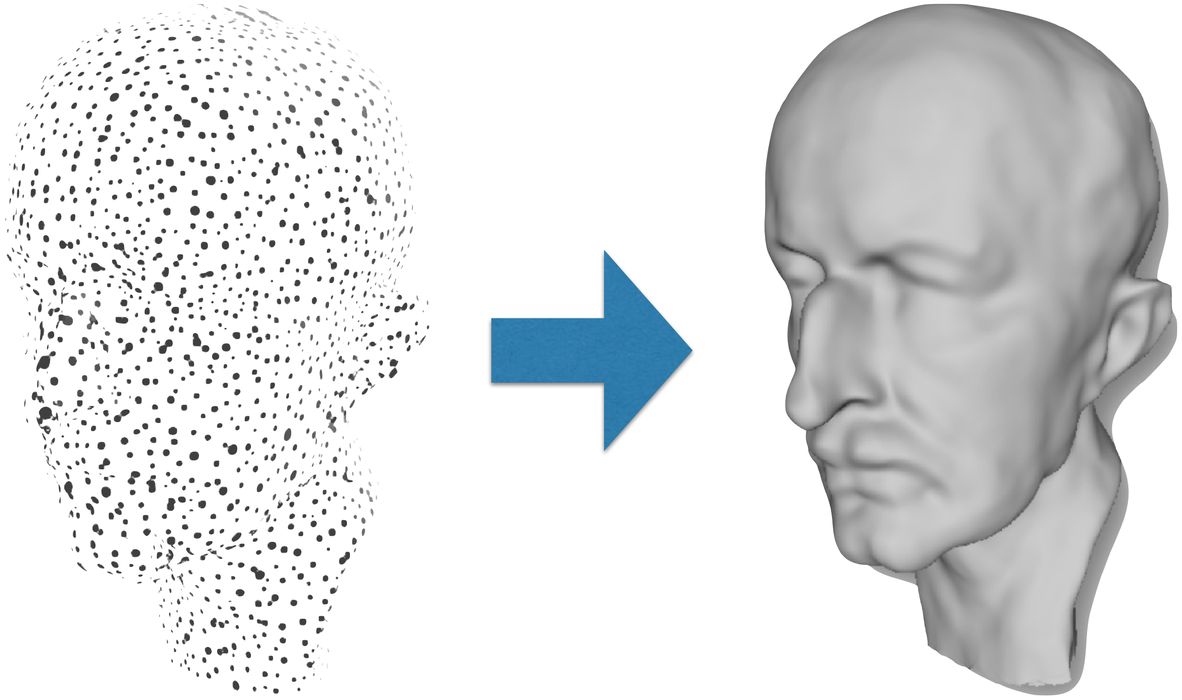
# Geometric Modeling
# Assignment 2: Implicit Surface Reconstruction

Acknowledgements: Olga Diamanti, Julian Panetta



University of Victoria | Computer Science

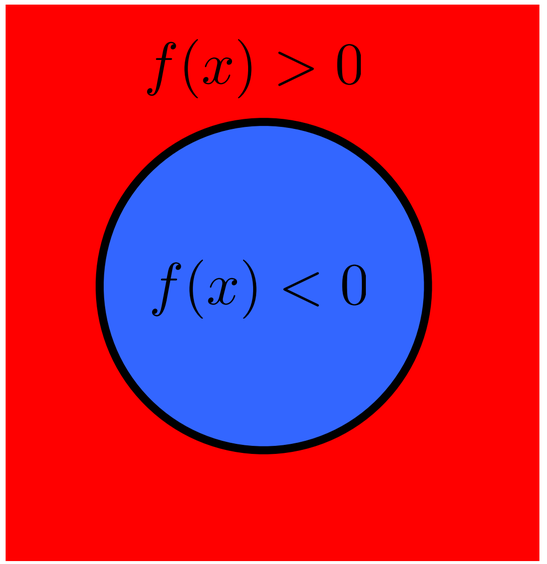
# Surface Reconstruction

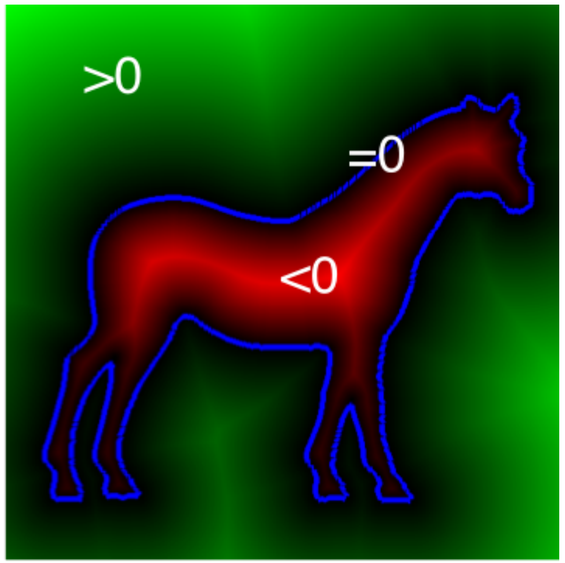


- Input: point cloud with normals
- Output: smooth surface mesh passing near each point

# Implicit Surface Reconstruction

- Remember: surface representation matters!
- Implicit representation bypasses many headaches an explicit approach would encounter.
- Guarantees by construction:
  - 2-Manifold
  - No holes (watertight)
- Robust to noisy point clouds

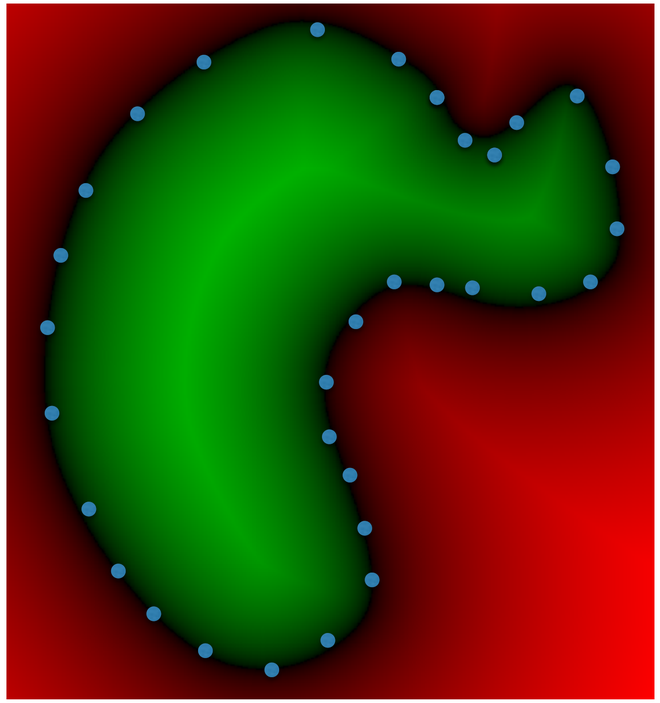


# Simplifies the Problem

Surface interpolation


Scalar field interpolation

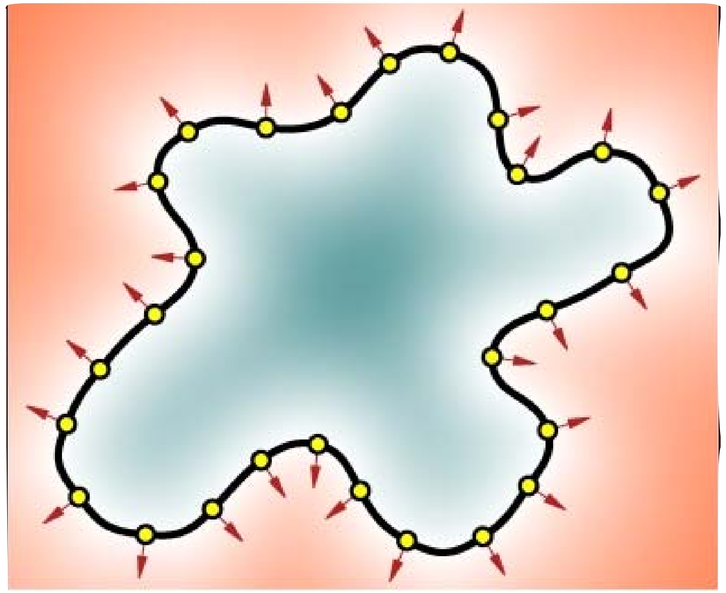


# Constructing the Scalar Field

- Interpolate information from the input cloud:
  - Points tell us where the zero level set should go:

    $f(\mathbf{p}_i) = 0$

  - Normals define (locally) inside/outside

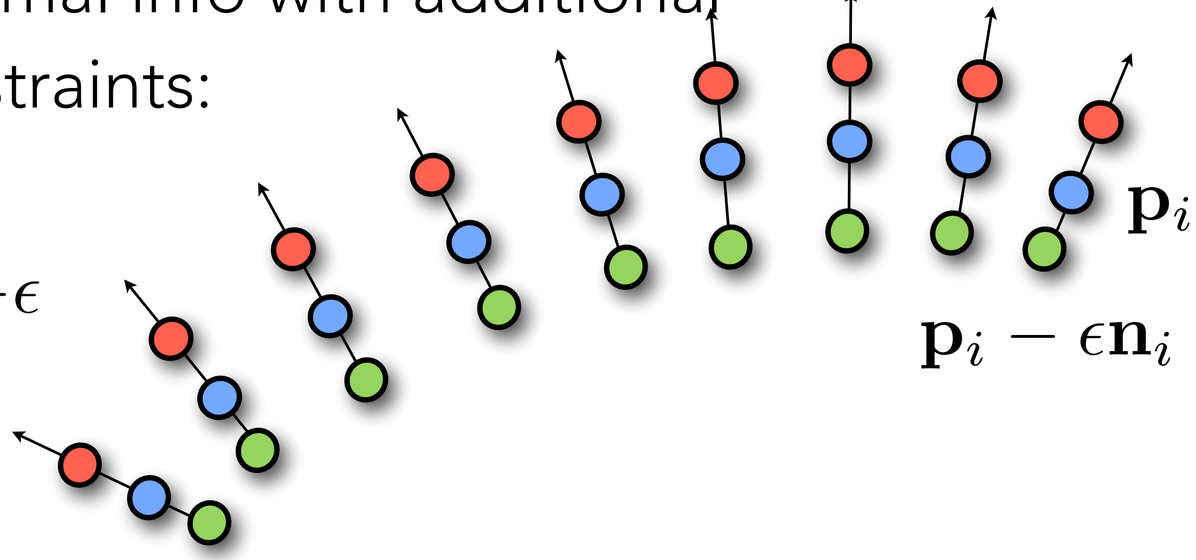


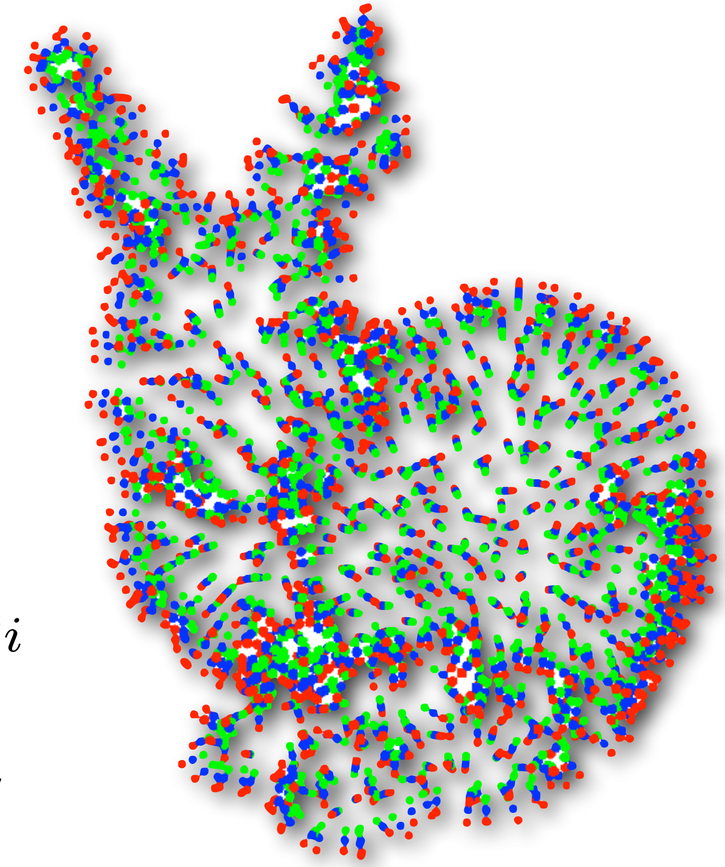
# Step 1: Build the Constraint Set

- Input: .off file with points and normals
- Point constraints $f(\mathbf{p}_i) = 0$
  are insufficient (trivial solution f = 0)
- Incorporate normal info with additional off-surface constraints:

$$f(\mathbf{p}_i + \epsilon \mathbf{n}_i) = \epsilon$$

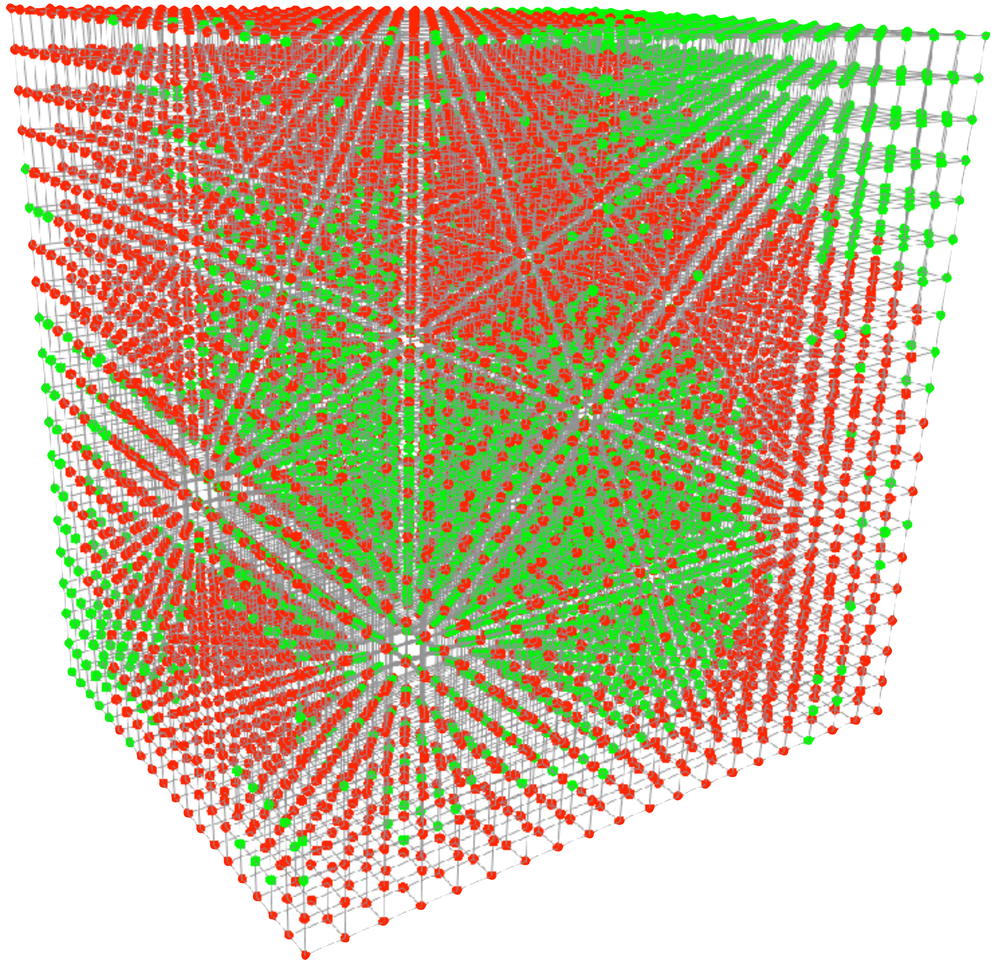
$$f(\mathbf{p}_i - \epsilon \mathbf{n}_i) = -\epsilon$$



# Step 2: Construct Interpolant

- Input: .off file with points and normals
- Construct regular grid
- Compute nodal scalar field satisfying constraints (approximately).
- Method: MLS
  (Moving Least Squares)



# Interpolation Problem

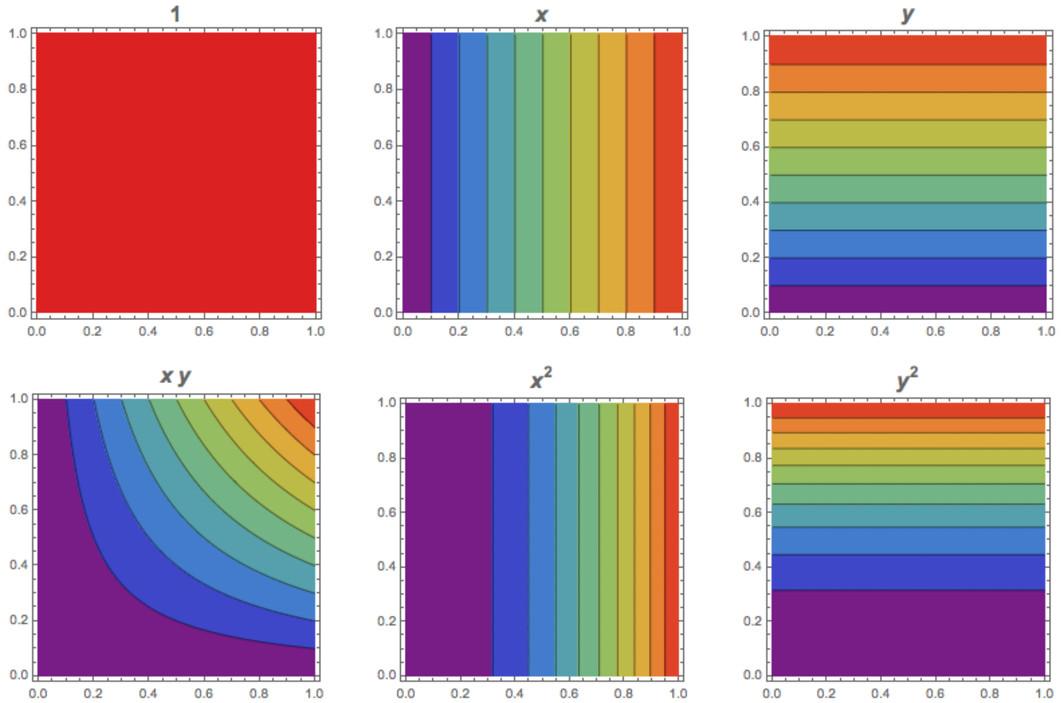
- List of 3N constraint locations, ci (e.g. p0, p0+εn0, …)
- List of 3N values, di
- Together, they describe 3N constraints of the form
  f(ci) = di
- Goal: find the "best" f in the span of chosen basis functions b(x):

$$f(\mathbf{x}) = \sum_j b_j(\mathbf{x}) a_j$$

(By tuning weights aj to best approximate constraints)

# Basis Functions

- For this assignment, we'll use polynomial basis functions:




(but in 3D)

# Constraints in the Basis

- We can express our constraints in this basis:

$$f(\mathbf{c}_i) = \sum_j b_j(\mathbf{c}_i) a_j = d_i$$

In matrix form:

$$B\mathbf{a} = \mathbf{d}$$

- Where matrix $B_{ij} := b_j(c_i)$
  (columns hold basis function's value
  at every constraint location).

$$B = \begin{bmatrix} 1 & x_1 & y_1 & \dots \\ 1 & x_2 & y_2 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

# Overconstrained Linear System

- We'll have many more constraints than basis functions…
- Least-squares solution?

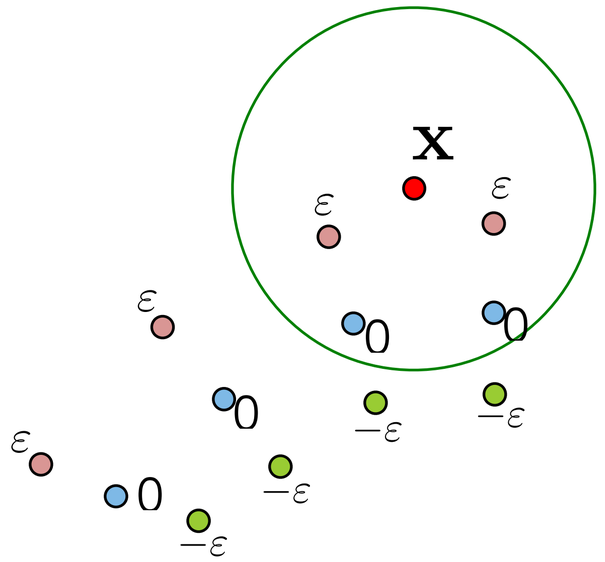
$$\min_f \sum_i (f(\mathbf{c}_i) - d_i)^2$$

$$\min_{\mathbf{a}} \|B\mathbf{a} - \mathbf{d}\|^2$$

- What's bad about this?

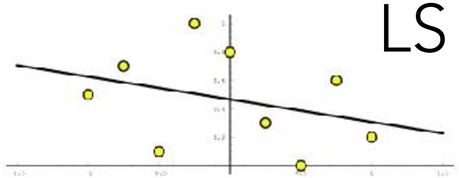
# Problems with Least-squares

- **Global problem**: large matrices (even if basis functions are local)
- **Need many, high-degree basis functions**
  - Evaluating interpolant becomes expensive
- **Better idea:**
  - Construct low degree, local interpolants and stitch them together

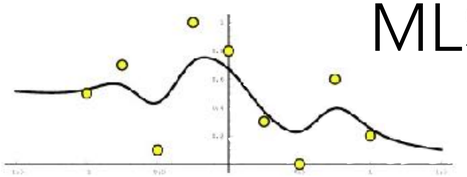


# Moving Least Squares (MLS)

LS



MLS



- MLS builds distinct local interpolant around every eval pt!
- But final stitched function is still guaranteed smooth.
- Idea: weight the constraints based on distance to eval pt x:

$$f_{\mathbf{x}} := \underset{f}{\operatorname{argmin}} \sum_i w\left(\|\mathbf{x} - \mathbf{c_i}\|\right)\left(f(\mathbf{c}_i) - d_i\right)^2$$

- Constraints with **zero weight disappear**!
  (Choose weight function so few kept => **small linear system**)

# MLS in Matrix Form

$$\min_f \sum_i w\left(\|\mathbf{x}-\mathbf{c}_i\|\right)\left(f(\mathbf{c}_i)-d_i\right)^2$$

$$\min_a \|B\mathbf{a}-\mathbf{d}\|^2_{W(\mathbf{x})}$$

Note: some papers call this $W(\mathbf{x})^2$

$$\|B\mathbf{a}-\mathbf{d}\|^2_{W(\mathbf{x})} := \left(B\mathbf{a}-\mathbf{d}\right)^T W(\mathbf{x})\left(B\mathbf{a}-\mathbf{d}\right)$$

$$W(\mathbf{x}) = \begin{bmatrix} w(\|\mathbf{x}-\mathbf{c}_1\|) & & \\ & \ddots & \\ & & w(\|\mathbf{x}-\mathbf{c}_{3N}\|) \end{bmatrix}$$

# MLS Coefficients, Closed Form

- MLS objective function is quadratic in coefficients **a**; find optimum by differentiating and solving a linear system:

$$\begin{aligned} 0 =& \nabla_{\mathbf{a}} \Big( \big(B\mathbf{a} - \mathbf{d}\big)^T W(\mathbf{x}) \big(B\mathbf{a} - \mathbf{d}\big) \Big) \\ =& \quad 2B^T W(\mathbf{x}) B\mathbf{a} - 2B^T W(\mathbf{x})\mathbf{d} \end{aligned}$$

- Thus the coefficients **for point x** are given by solving the system:

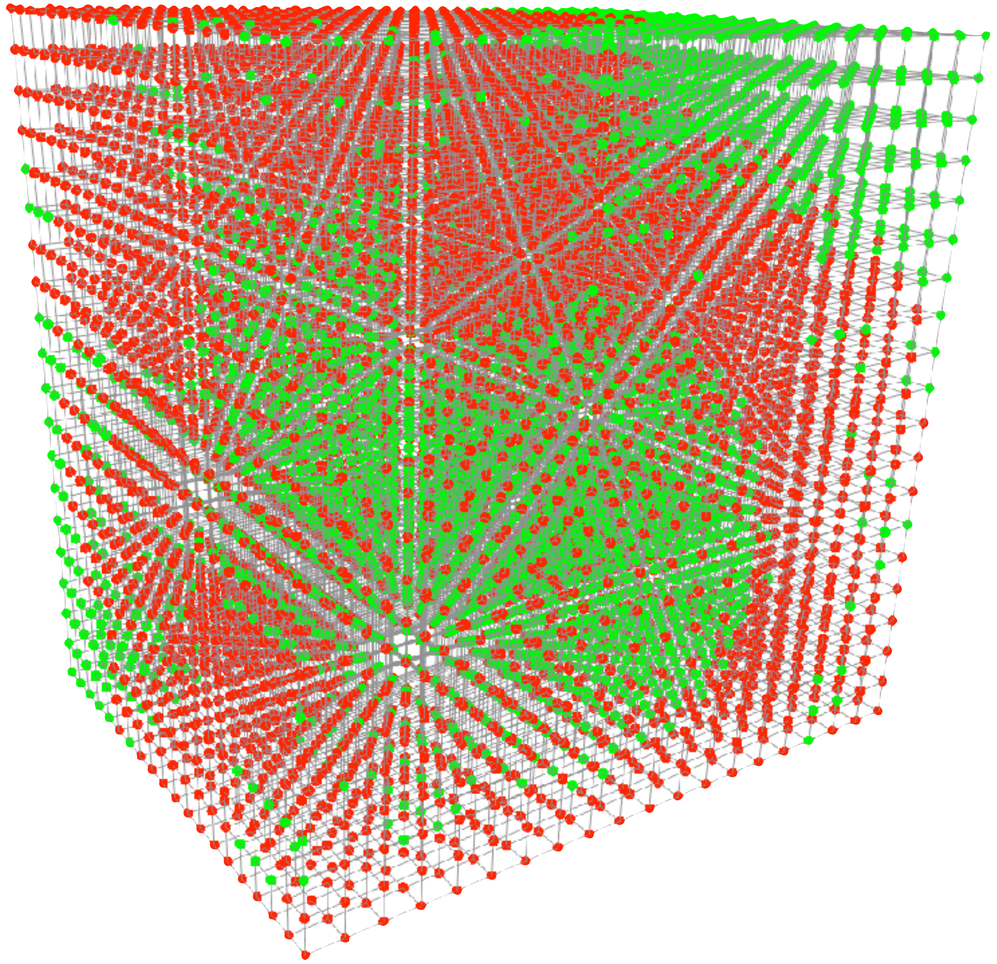
$$\Big( B^T W(\mathbf{x}) B \Big) \mathbf{a}(\mathbf{x}) = B^T W(\mathbf{x}) \mathbf{d}$$

for **a(x)**.

# Step 2: Construct Interpolant

- Input: .off file with points and normals
- Finally, fill in the grid!
- Evaluate local MLS interpolant at each grid point x.

$$f_{\mathbf{x}}(\mathbf{x}) = \sum_j b_j(\mathbf{x}) a_j(\mathbf{x})$$

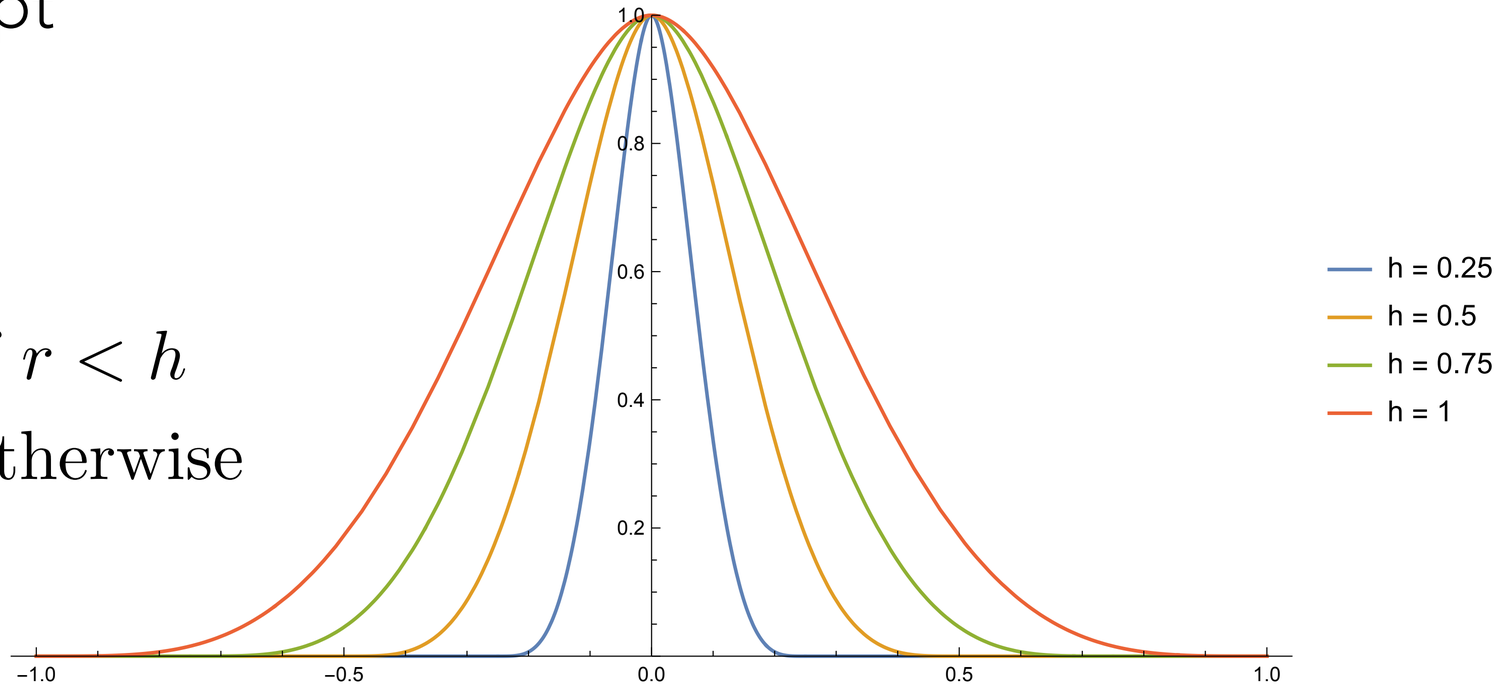


# Wendland Weights

- You'll use **Wendland** weights for w in this assignment

- Vanish at dist "h" from eval pt (most constraints disappear)

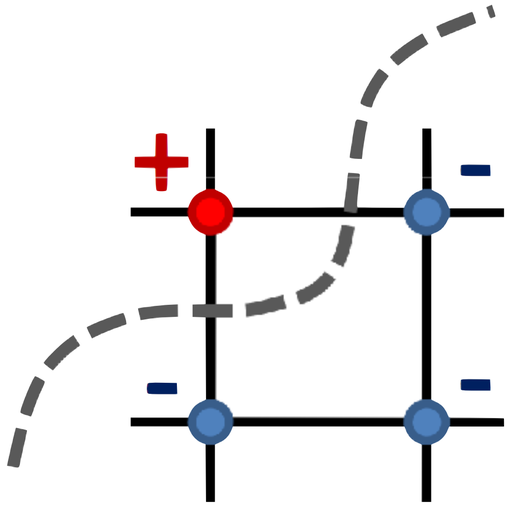
$$w(r) := \begin{cases} \left(1 - \frac{r}{h}\right)^4 \left(4\frac{r}{h} + 1\right) & \text{if } r < h \\ 0 & \text{otherwise} \end{cases}$$

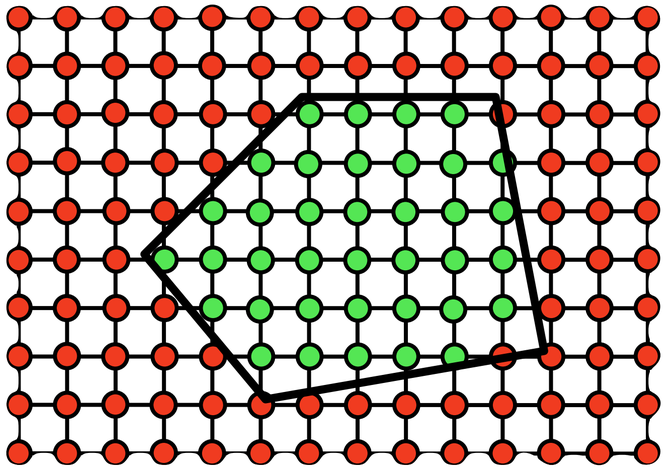


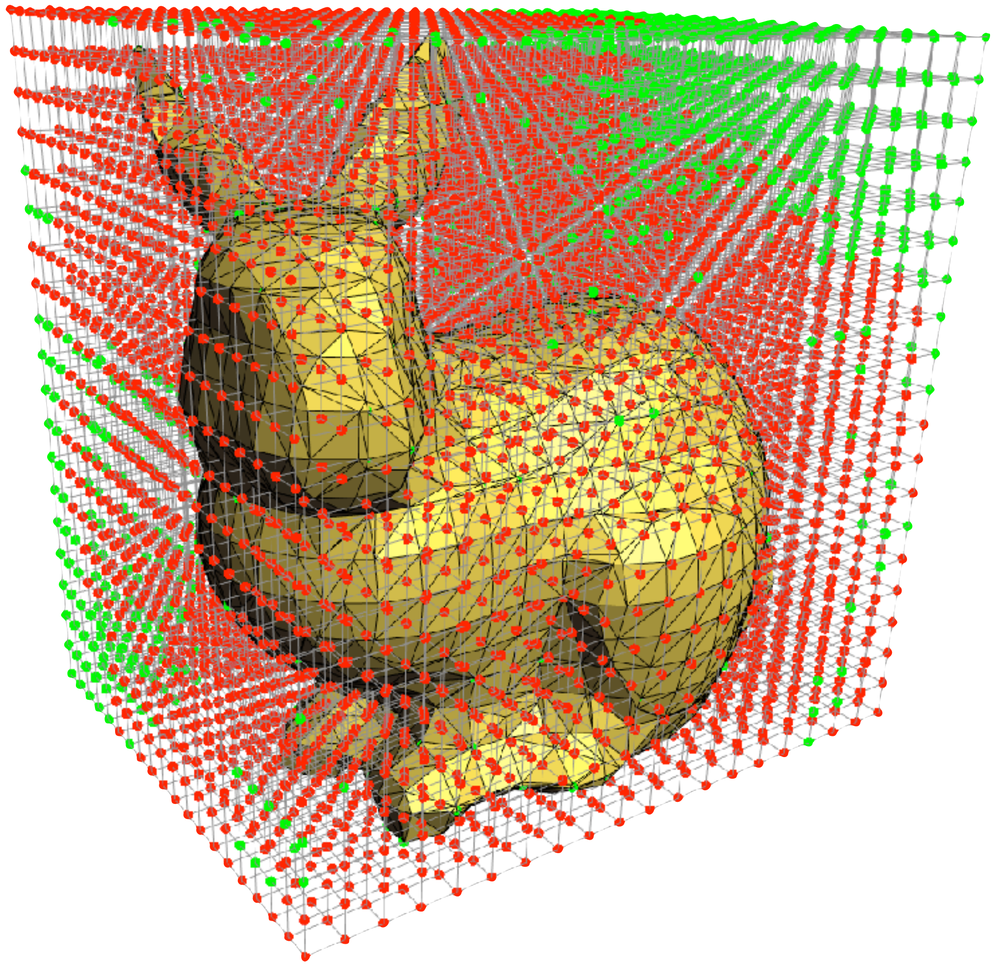
# Step 3: Extract Zero Level Set

- Use the **marching tets** algorithm to extract the grid function's zero isosurface

- Just call igl.marching_tets

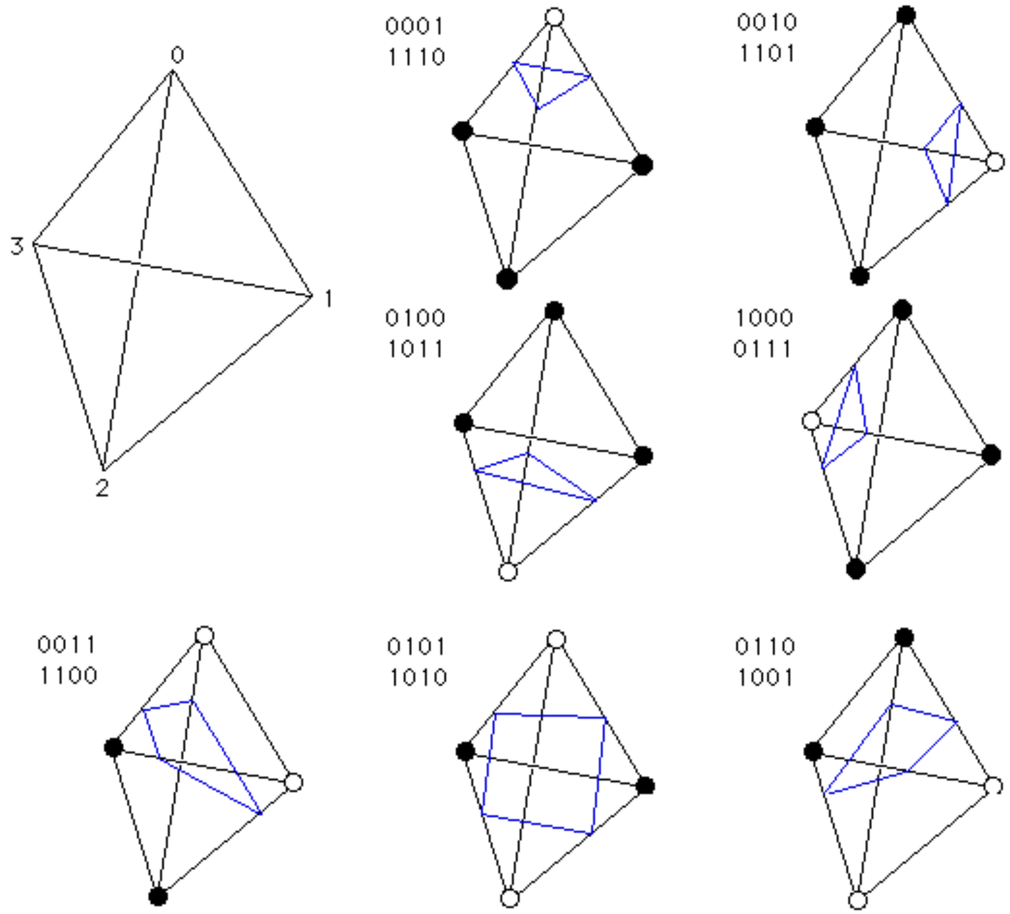




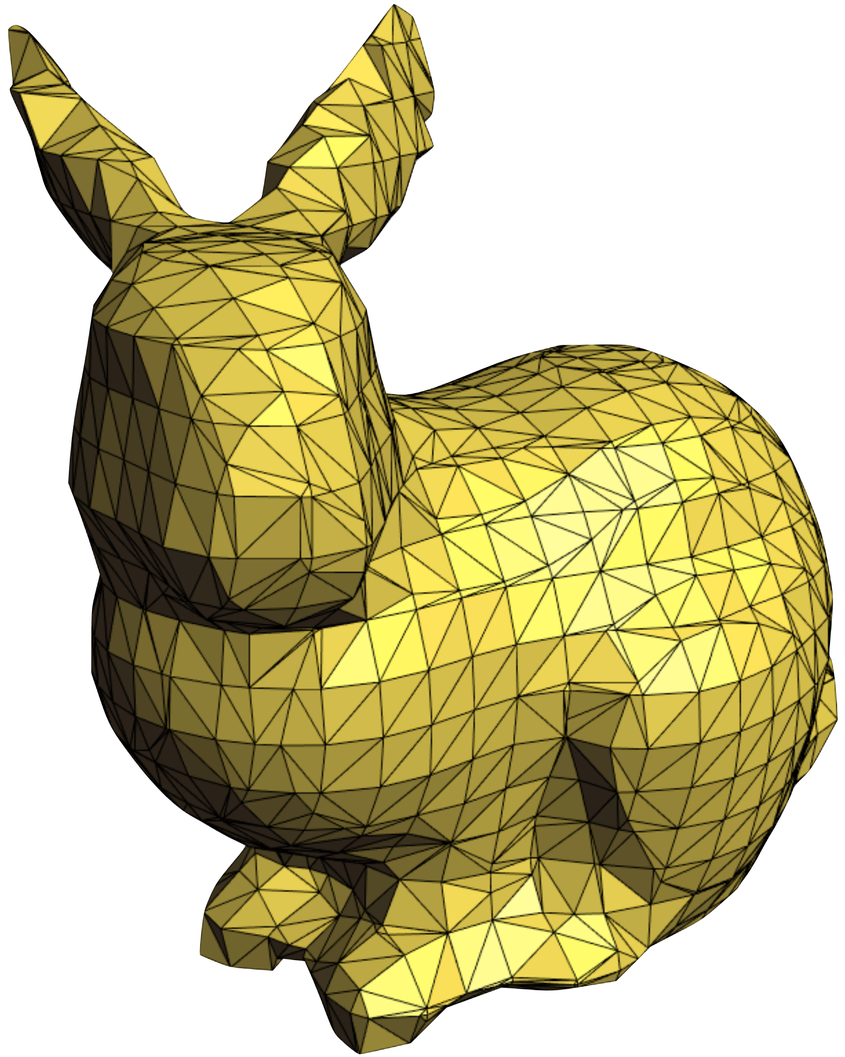
# Marching Tets: General Idea

Look up triangles to create in each grid cell based on corner values:



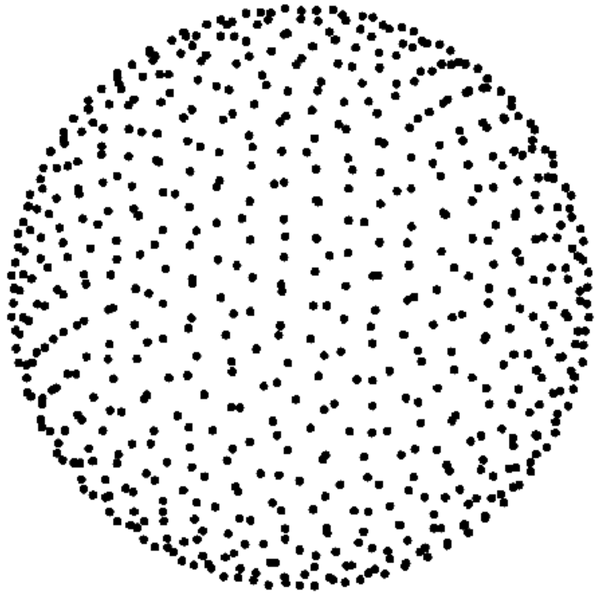
# Final Result from Marching Tets

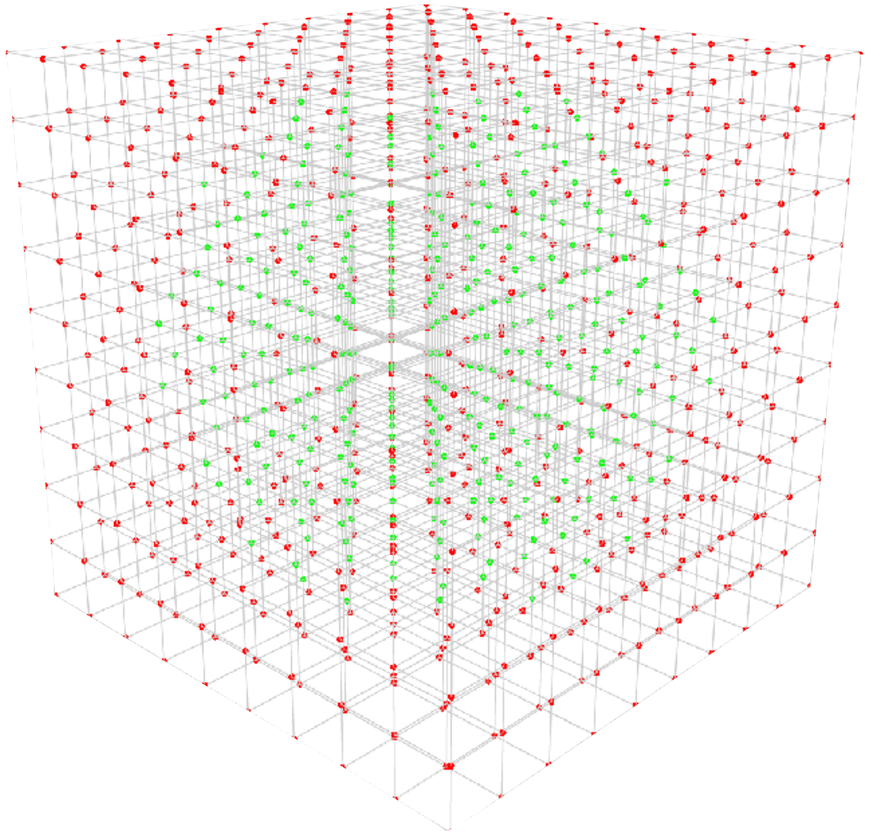
- Final Mesh

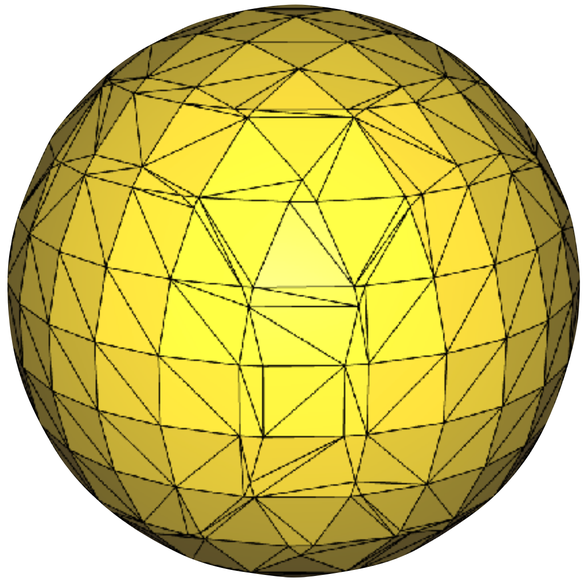


# Provided Code

- Implements pipeline but uses analytic signed distance fn for sphere in place of MLS







# Provided Example: Implicit Sphere

- Step 1: Compute an axis-aligned bounding box

```
bbox_min = np.array([-1., -1., -1.])
bbox_max = np.array([1., 1., 1.])

# Bounding box dimensions
dim = bbox_max - bbox_min
bbox_diag = np.linalg.norm(dim)
```

# Provided Example: Implicit Sphere

- Step 2: construct a grid over the bounding box

```
# Grid spacing
n = 30
x, T = tet_grid((n, n, n),
                bbox_min - 0.05 * bbox_diag,
                bbox_max + 0.05 * bbox_diag)
```

# Provided Example: Implicit Sphere

- Step 3: Fill grid with the values of the implicit function

```
# evaluate implicit function
# Scalar values of the grid points (the implicit function values)
center = np.array([0., 0., 0.])
radius = 1.
fx = np.linalg.norm(x-center, axis=1) - radius
```

$$f(\mathbf{x}) = \|\mathbf{x} - \mathbf{c}\| - r$$

# Provided Example: Implicit Sphere

- Step 4: run marching cubes

```
sv, sf, _, _ = igl.marching_tets(x, T, fx, 0)
```

input: implicit function values at grid points

# Provided Example: Implicit Sphere

- Step 4: run marching cubes

```
sv, sf, _, _ = igl.marching_tets(x, T, fx, 0)
```

input: implicit function value for level set

# Provided Example: Implicit Sphere

- Step 4: run marching cubes

```
sv, sf, _, _ = igl.marching_tets(x, T, fx, 0)
```

input: grid point positions

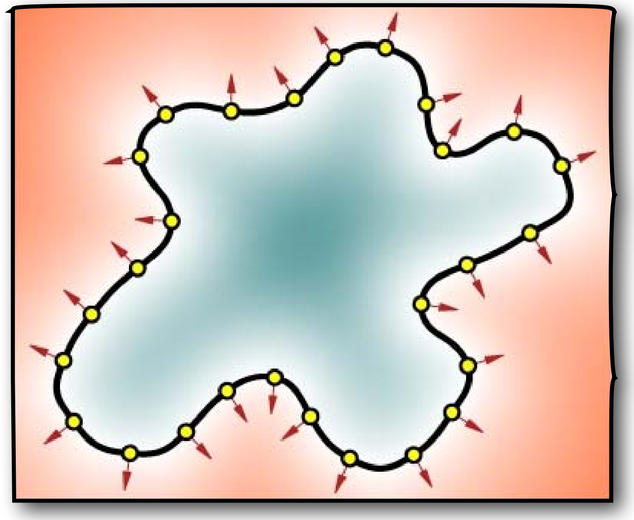
# Provided Example: Implicit Sphere

- Step 4: run marching cubes

```
sv, sf, _, _ = igl.marching_tets(x, T, fx, 0)
```

output: vertices and faces

# Bonus: Better Normal Constraints



- Our method implemented only point constraints
- Normals “constrained” using inward- and outward-offset value constraints
    - Leads to undesirable surface oscillation
- Solution: use the normal to define a linear function at each sample point; interpolate these **functions** with MLS.

- Chen Shen, James F. O'Brien, and Jonathan R. Shewchuk. "**Interpolating and Approximating Implicit Surfaces from Polygon Soup**". In *Proceedings of ACM SIGGRAPH 2004*, pages 896–904. ACM Press, August 2004. **(Section 3.3)**

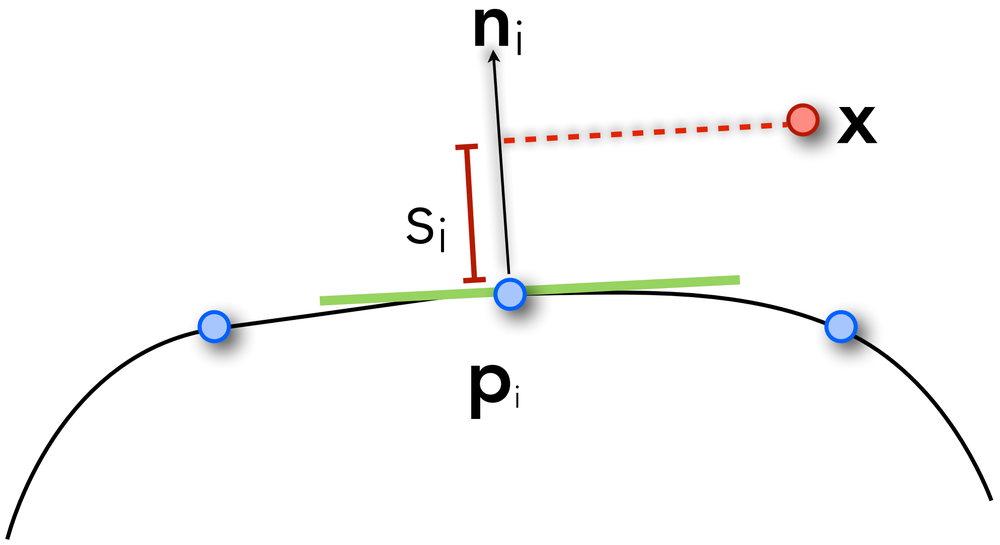
# Bonus: Better Normal Constraints

- Recall, we computed our interpolant by solving:

$$\min_a \|B\mathbf{a} - \mathbf{d}\|^2_{W(\mathbf{x})}$$

  with constraint value di for the 3N constraint locations.

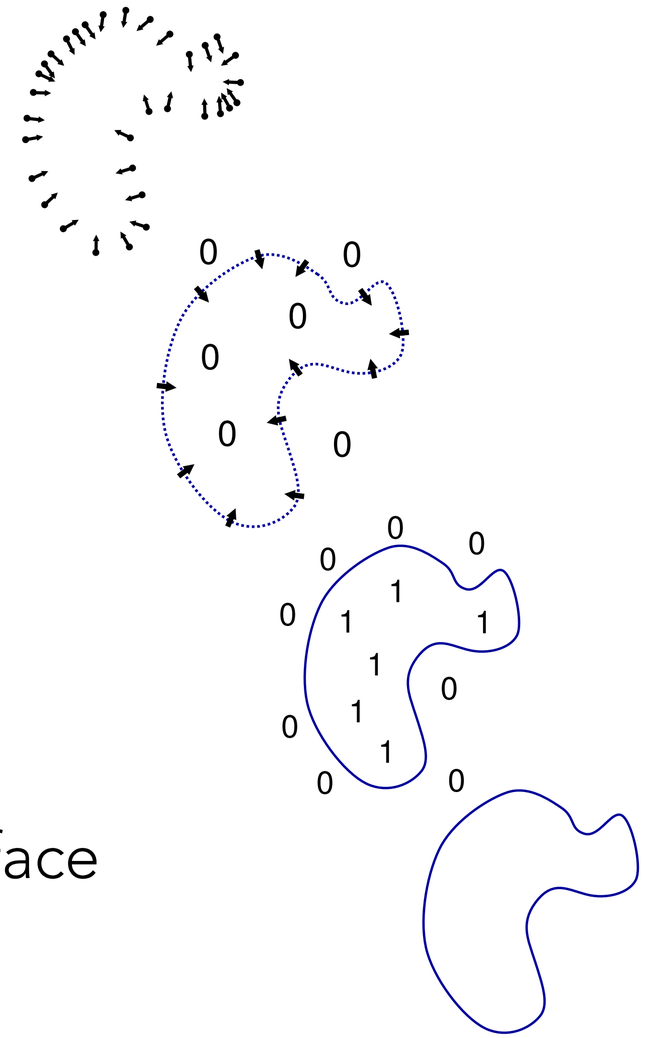
- New scheme: use just one constraint per sample pt
- Replace di with: $s_i(\mathbf{x}) = (\mathbf{x} - \mathbf{p}_i) \cdot \mathbf{n}_i$
- si is the linear function computing signed distance to pi's tangent plane
- Note: $\nabla_{\mathbf{x}} s_i = \mathbf{n}_i$



# Bonus: Poisson Reconstruction

- Explicitly a fit scalar function's gradient to the normals.
  - Smooth out sampled normals to create a global vector field $\vec{V}$
  - Find scalar function $\chi$ whose gradient best approximates this vector field: $min_\chi \|\nabla\chi - \vec{V}\|$

Approximate indicator function




- Advantages:
  - No spurious sheets far from the surface!
  - Robust to noise
- Michael Kazhdan,Matthew Bolitho, Hugues Hoppe. "Poisson Surface Reconstruction."
  In Eurographics Symposium on Geometry Processing, 2006.

# Questions?