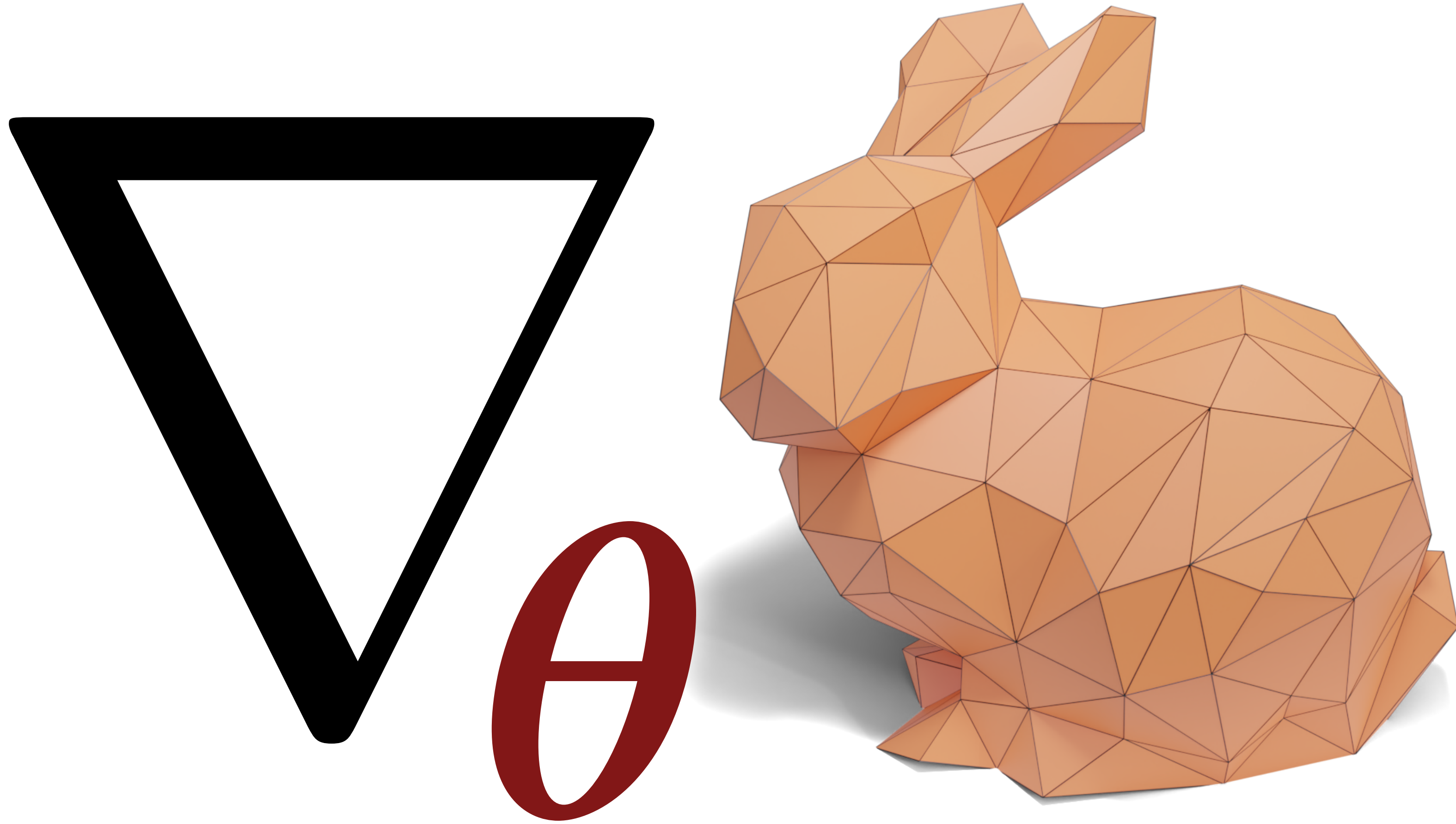


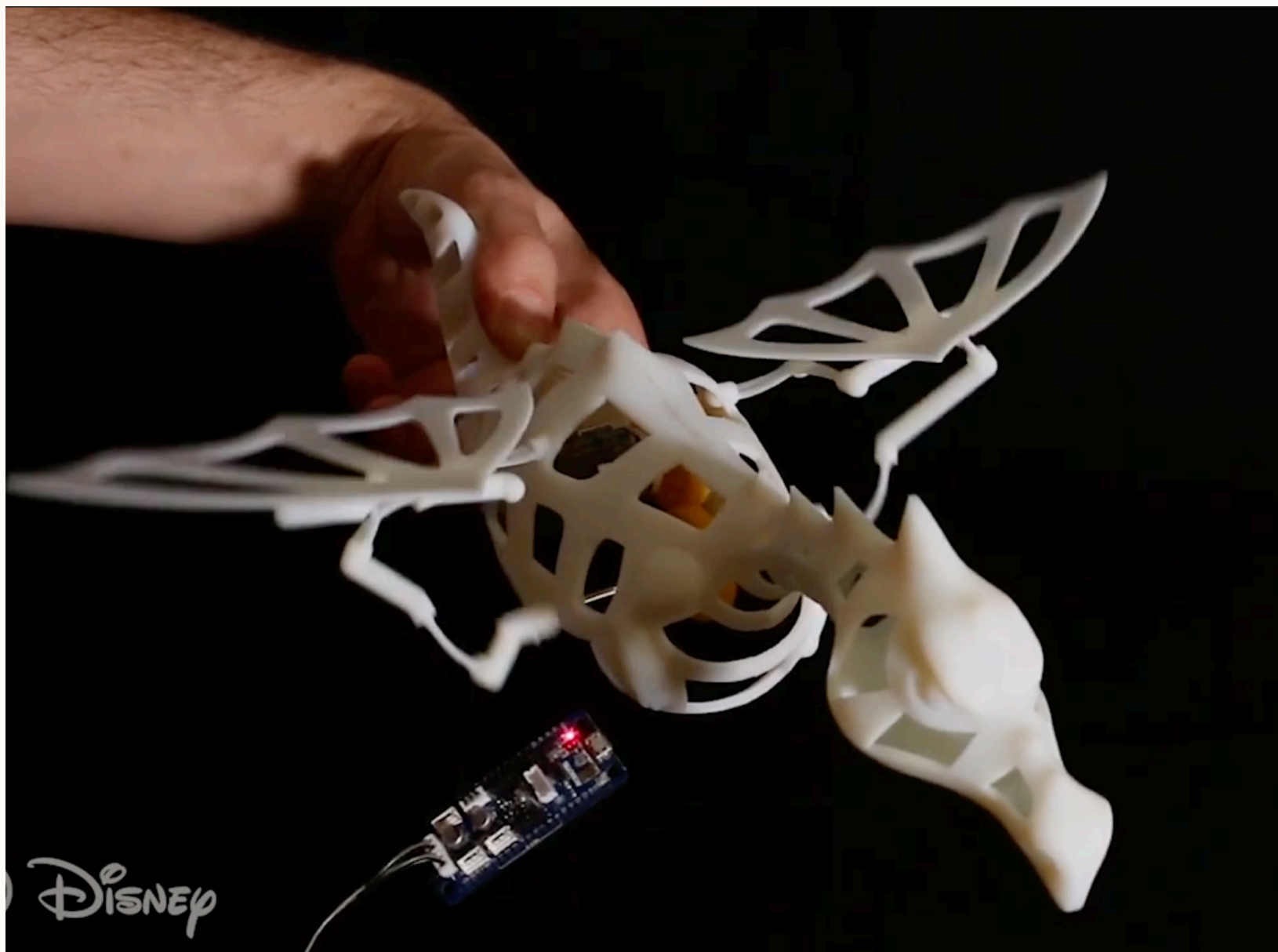


DIFFERENTIABLE GEOMETRY PROCESSING

IN PYTHON

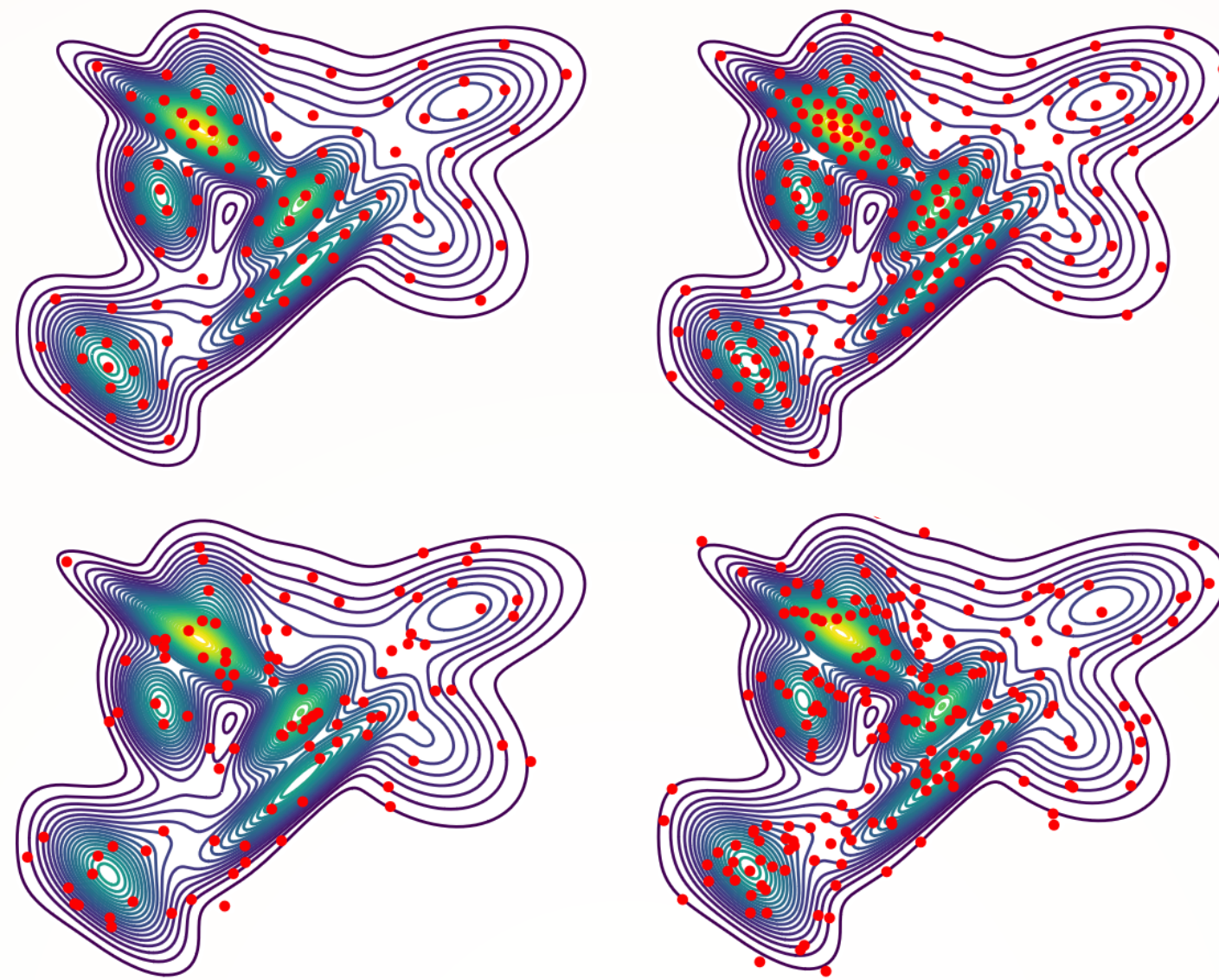
ANA DODIK, AHMED H. MAHMOUD





Computational Design

[Megaro et al. 2017]



Machine Learning

[Claici et al. 2018]



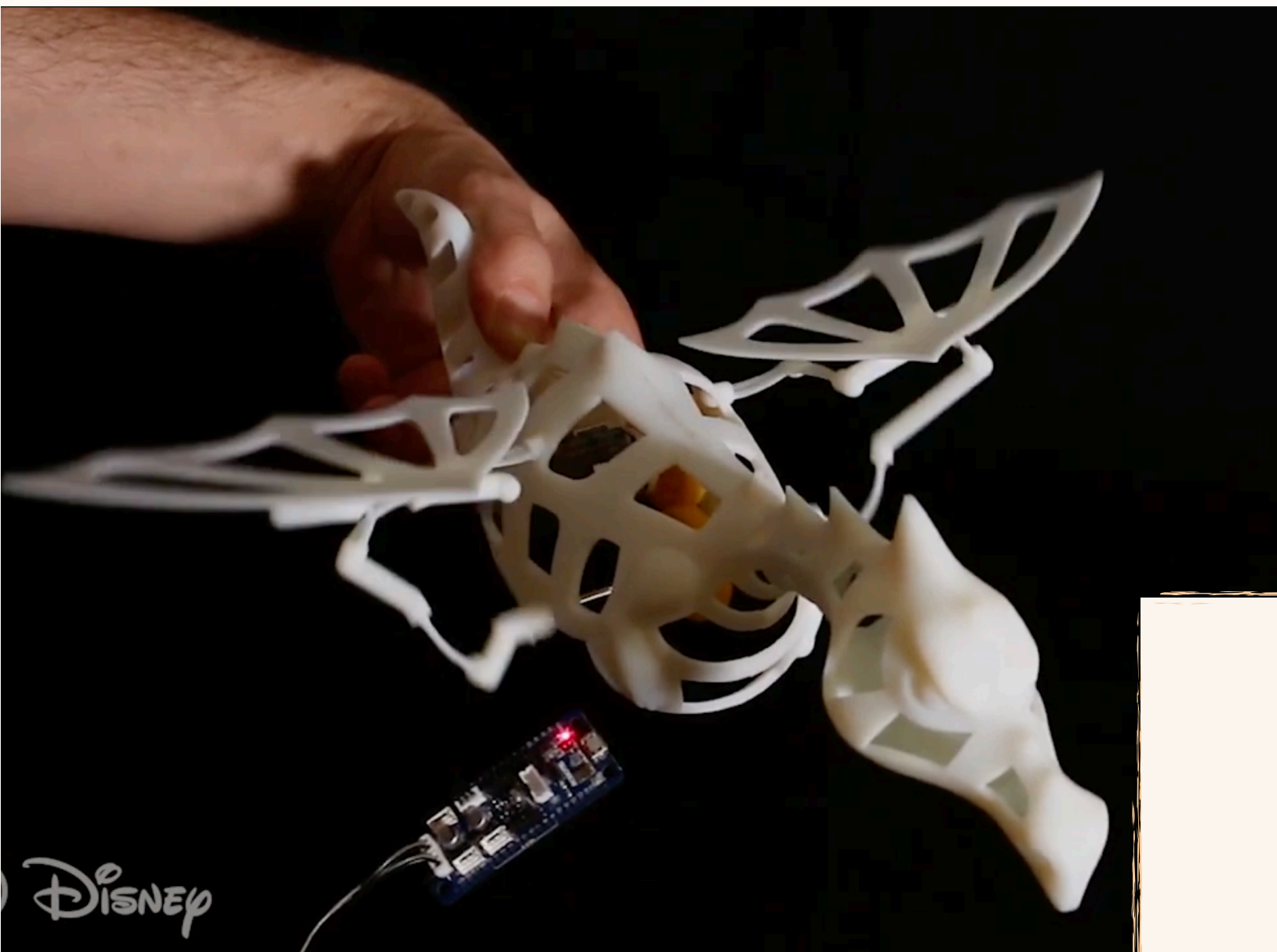
Robotics

[Sukhija et al. 2023]

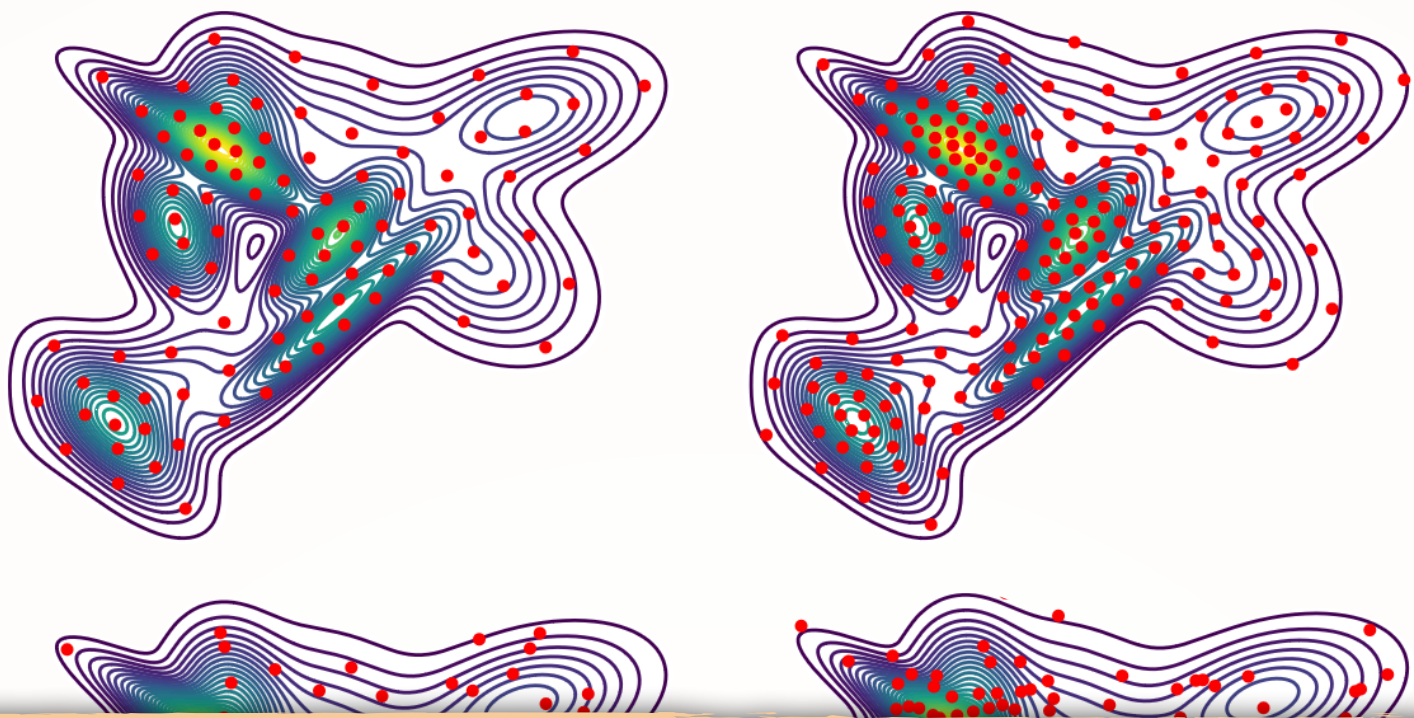


Inverse Rendering

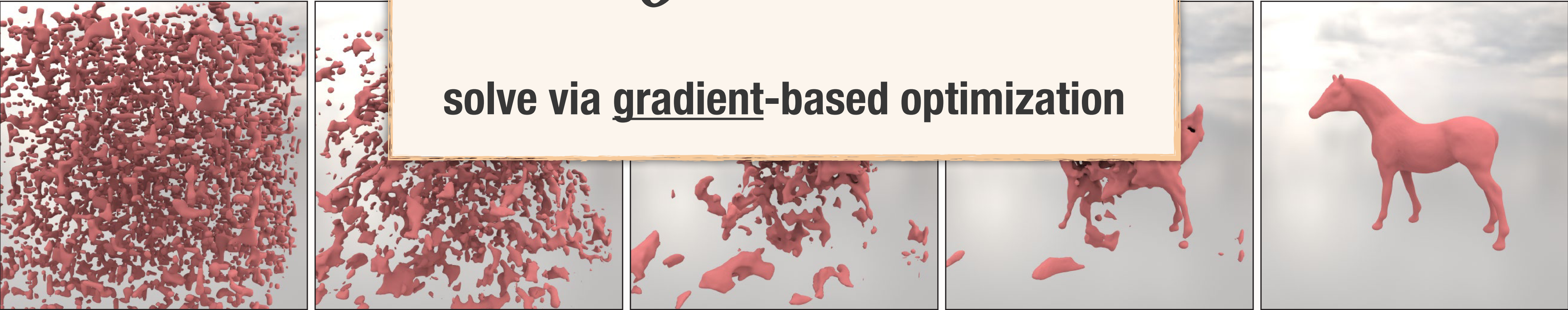
[Zhang et al. 2026]



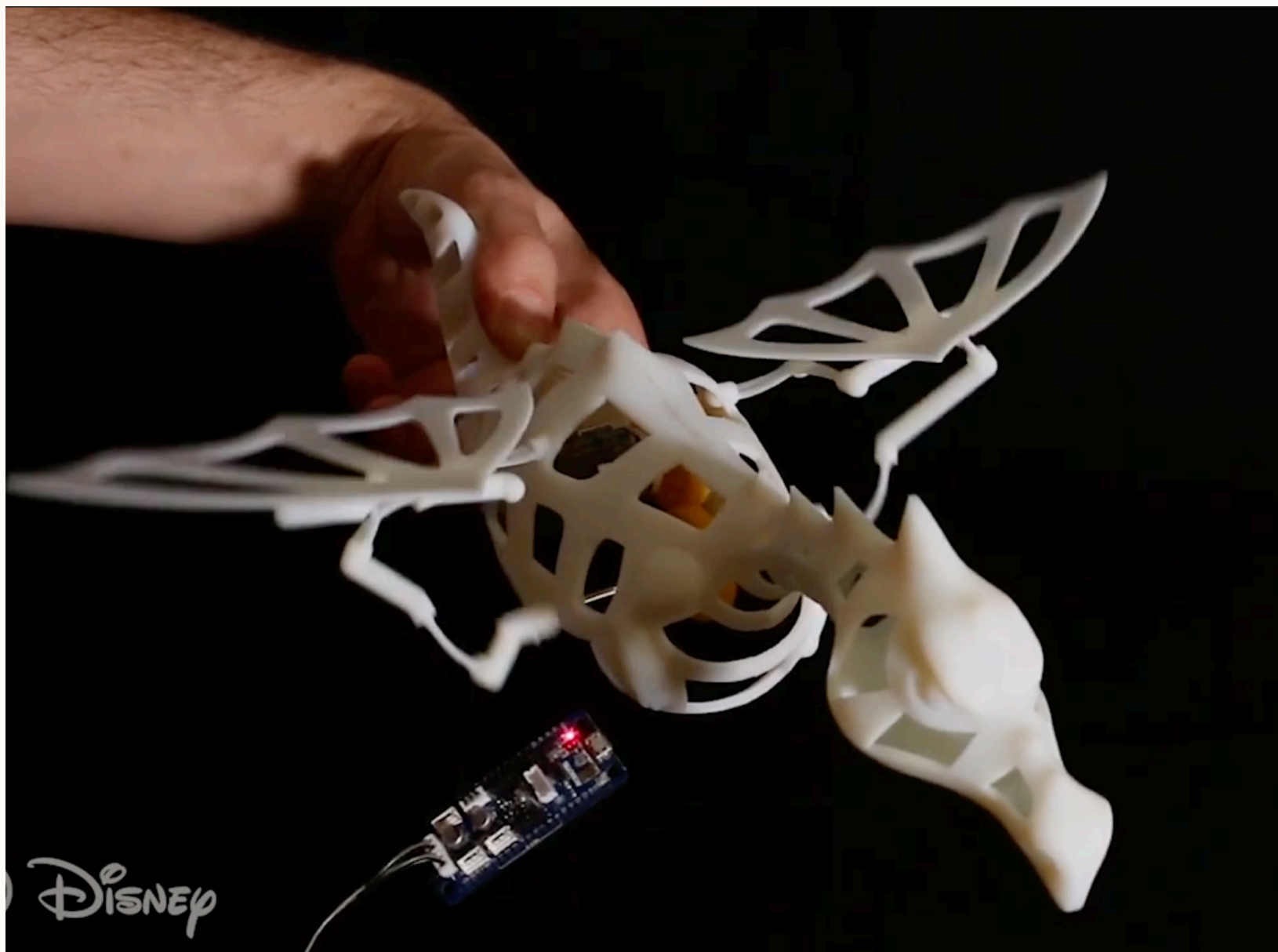
Computational Design
[Megaro et al. 2017]



Robotics
[Sukhija et al. 2023]

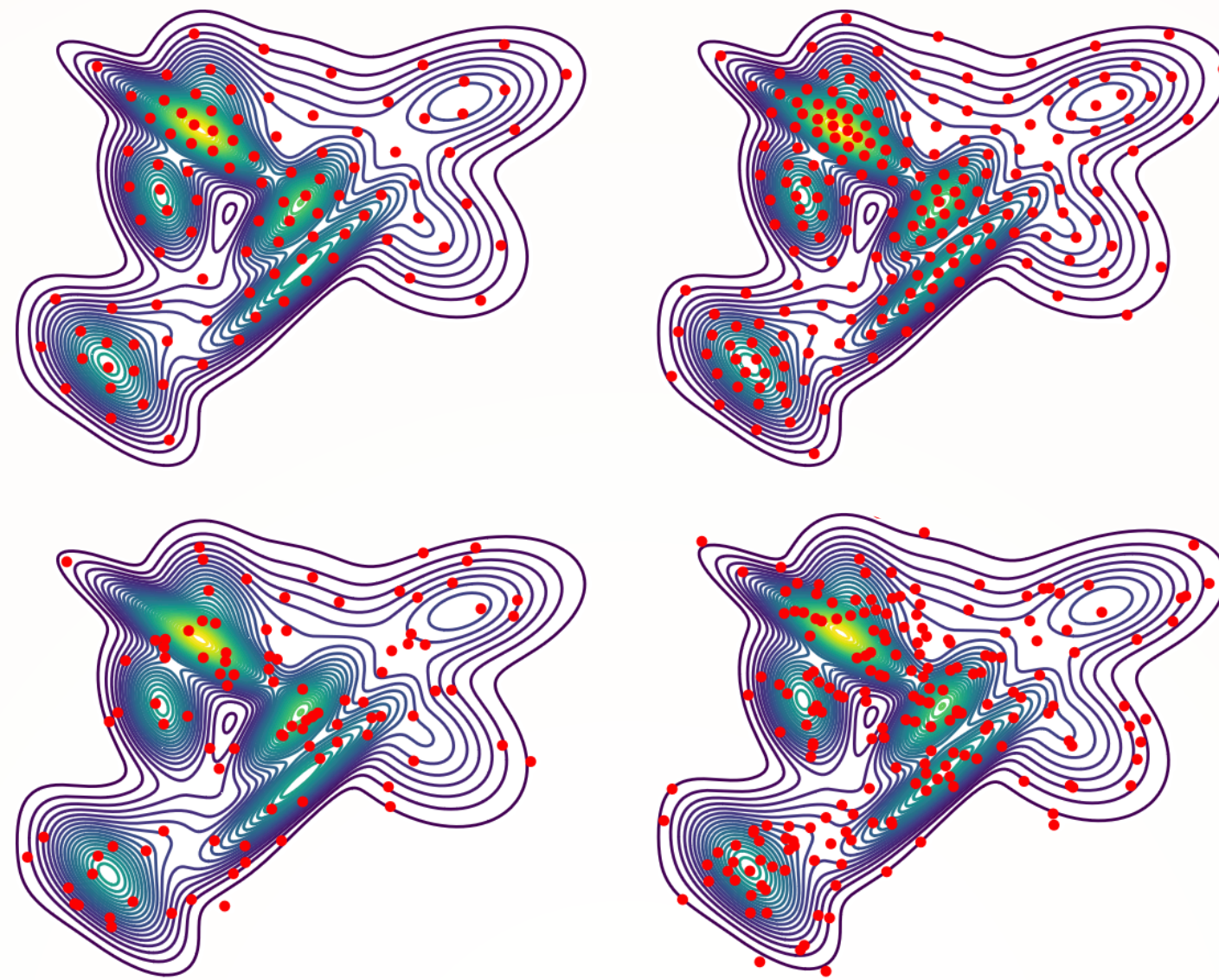


Inverse Rendering
[Zhang et al. 2026]



Computational Design

[Megaro et al. 2017]



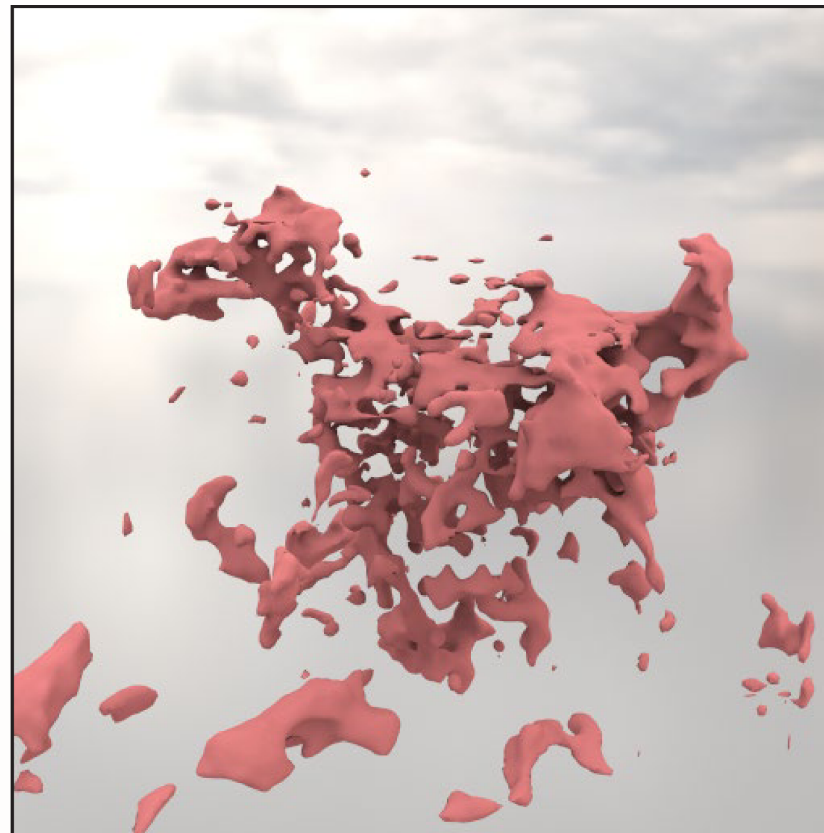
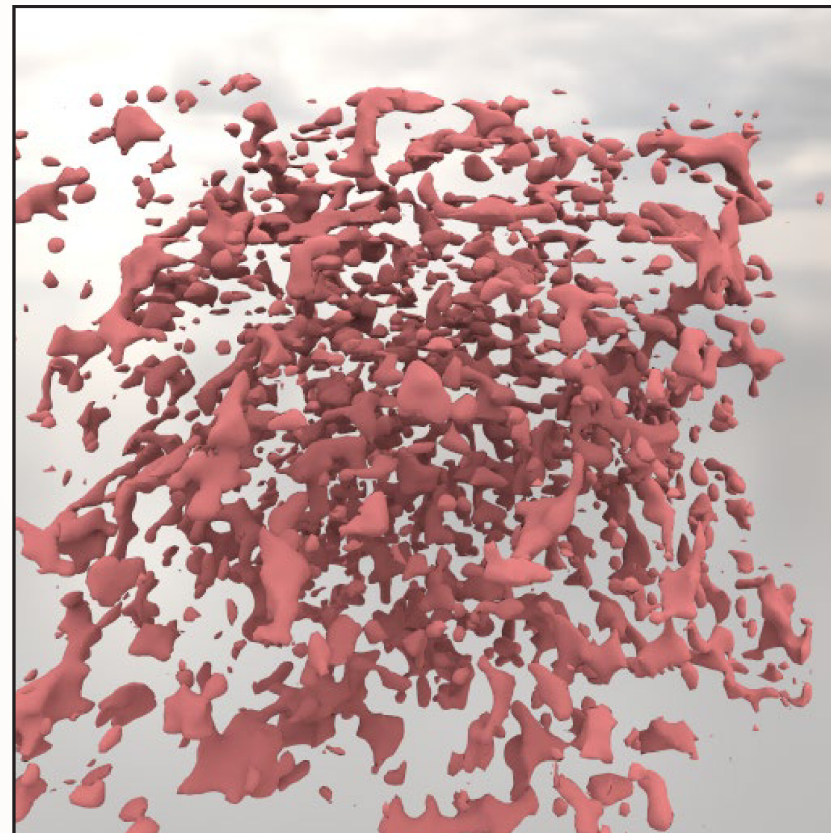
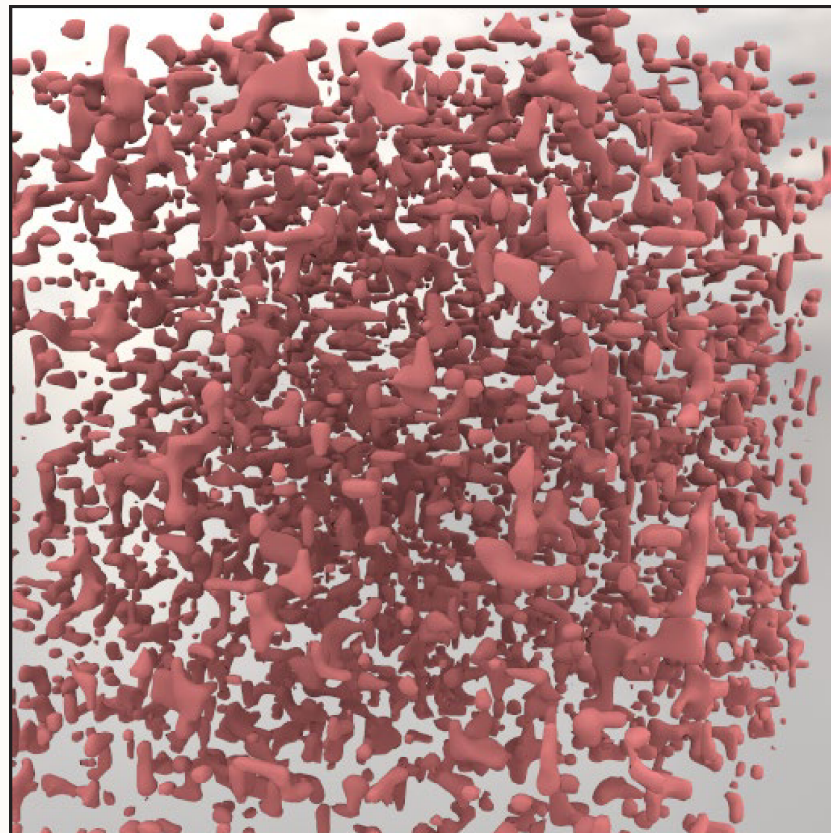
Machine Learning

[Claici et al. 2018]



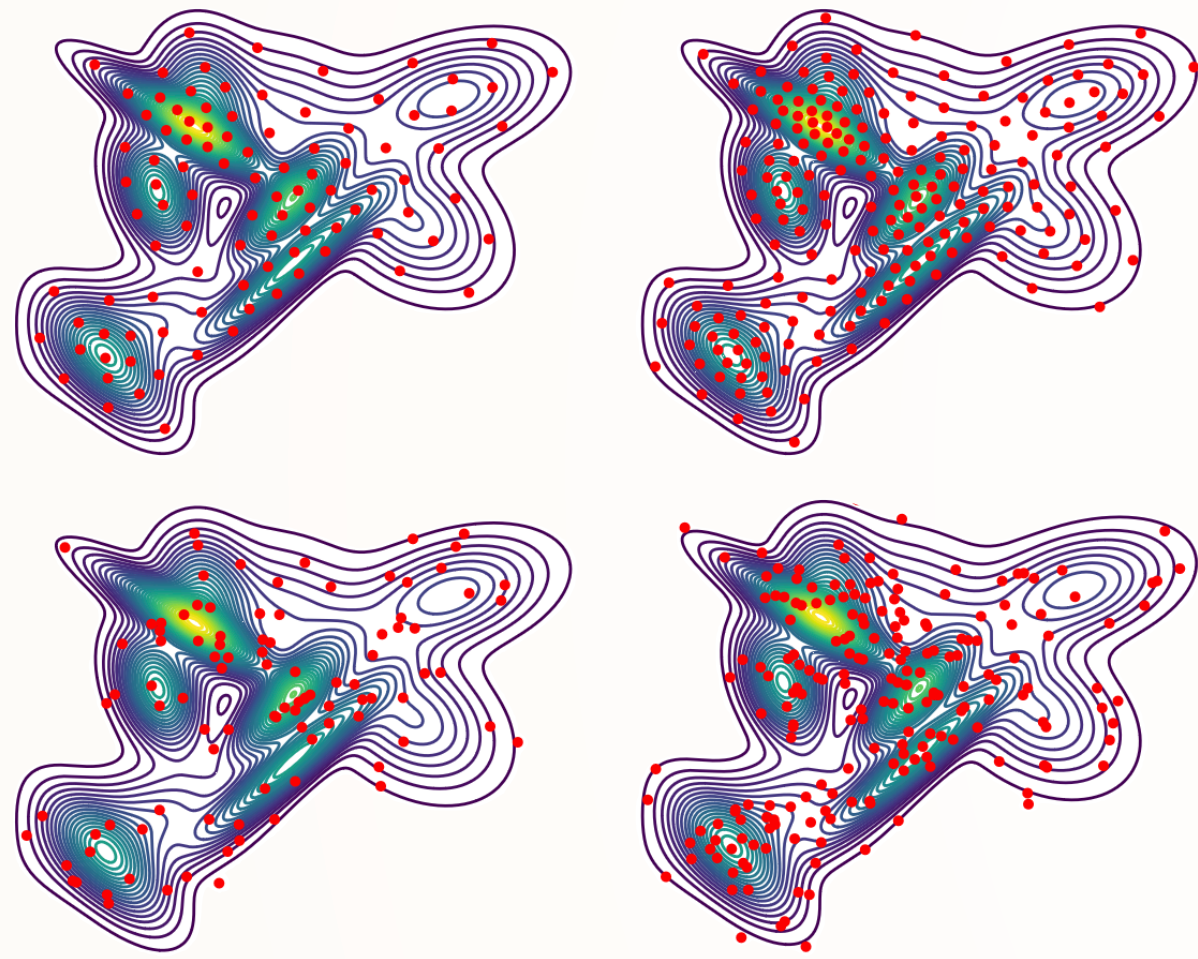
Robotics

[Sukhija et al. 2023]



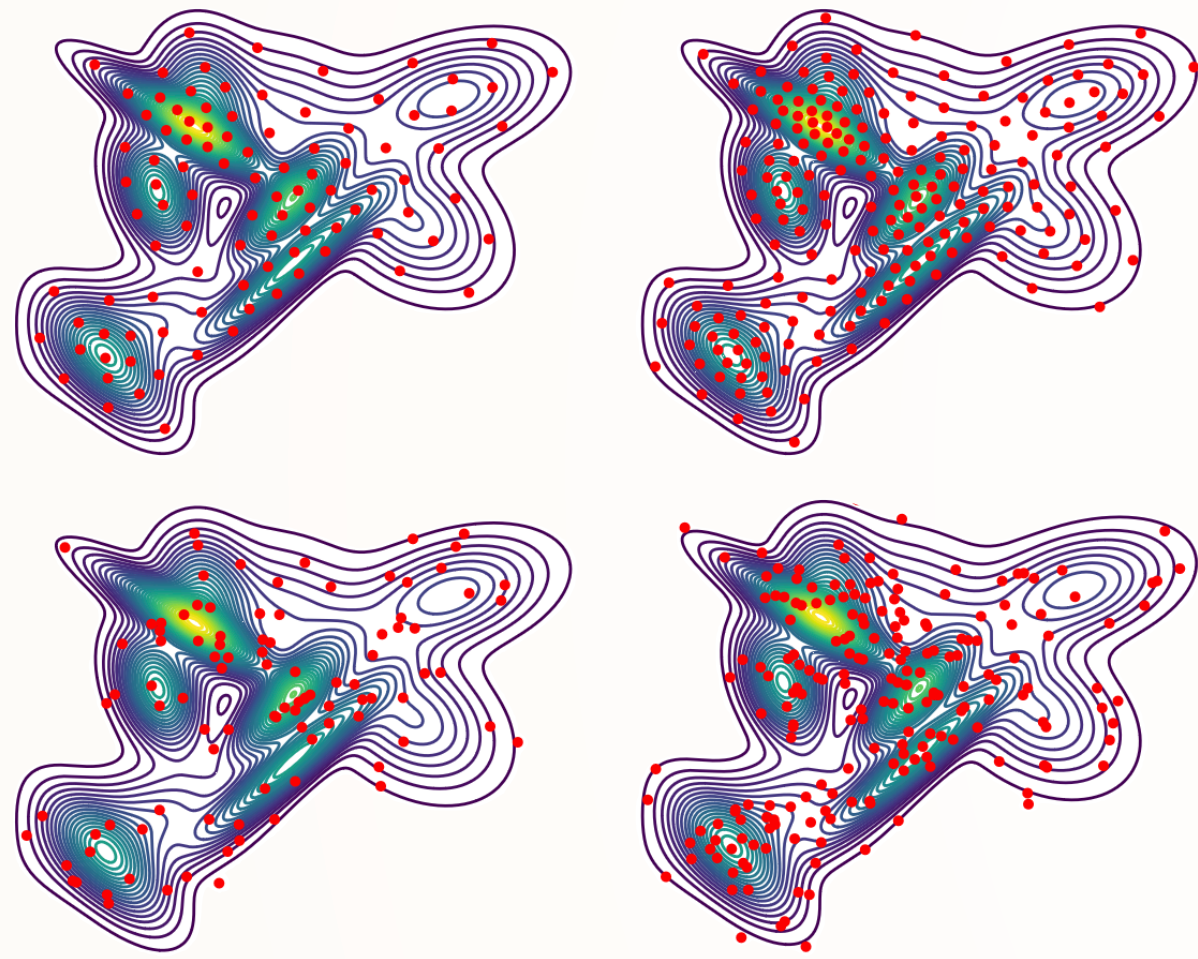
Inverse Rendering

[Zhang et al. 2026]



Machine Learning

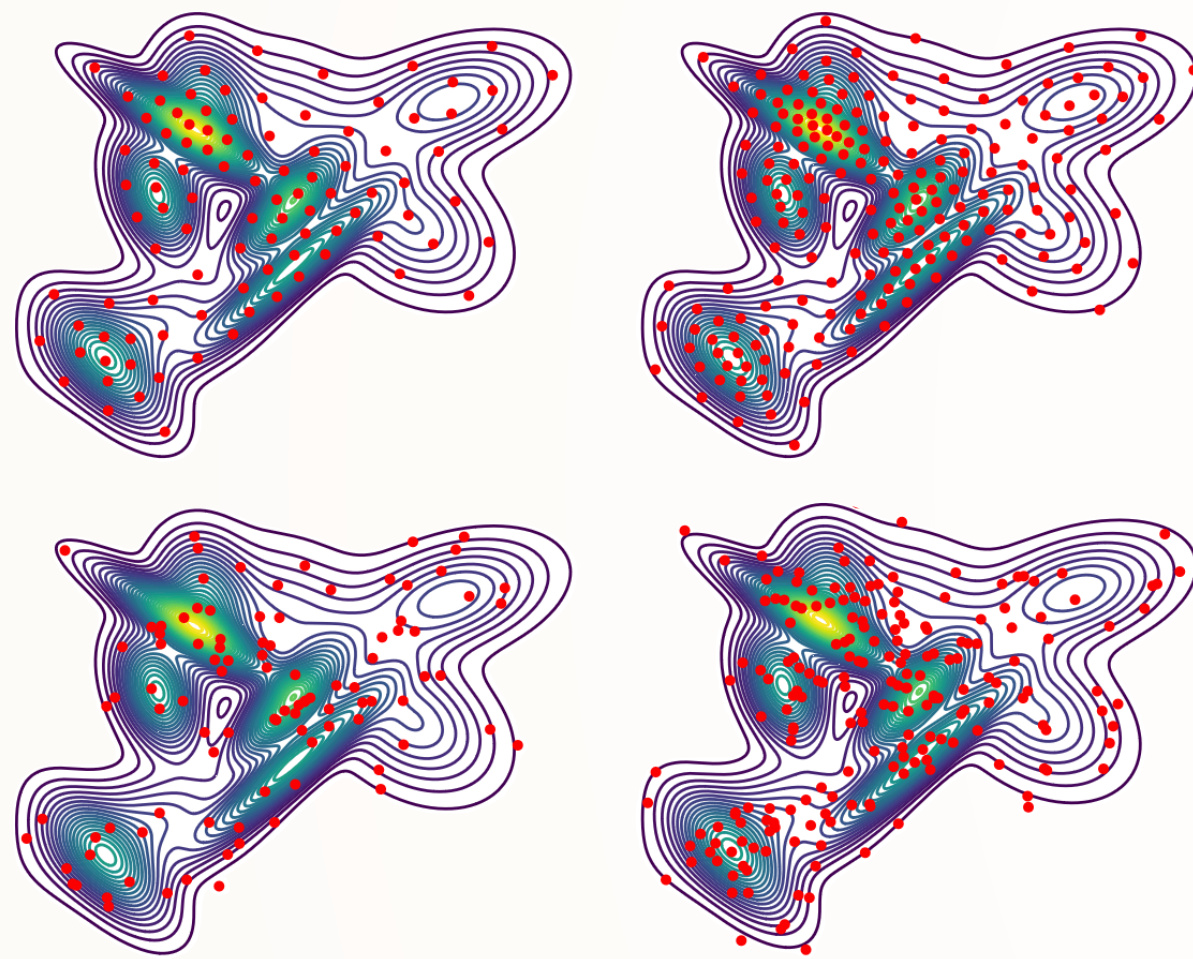
[Claici et al. 2018]



Machine Learning

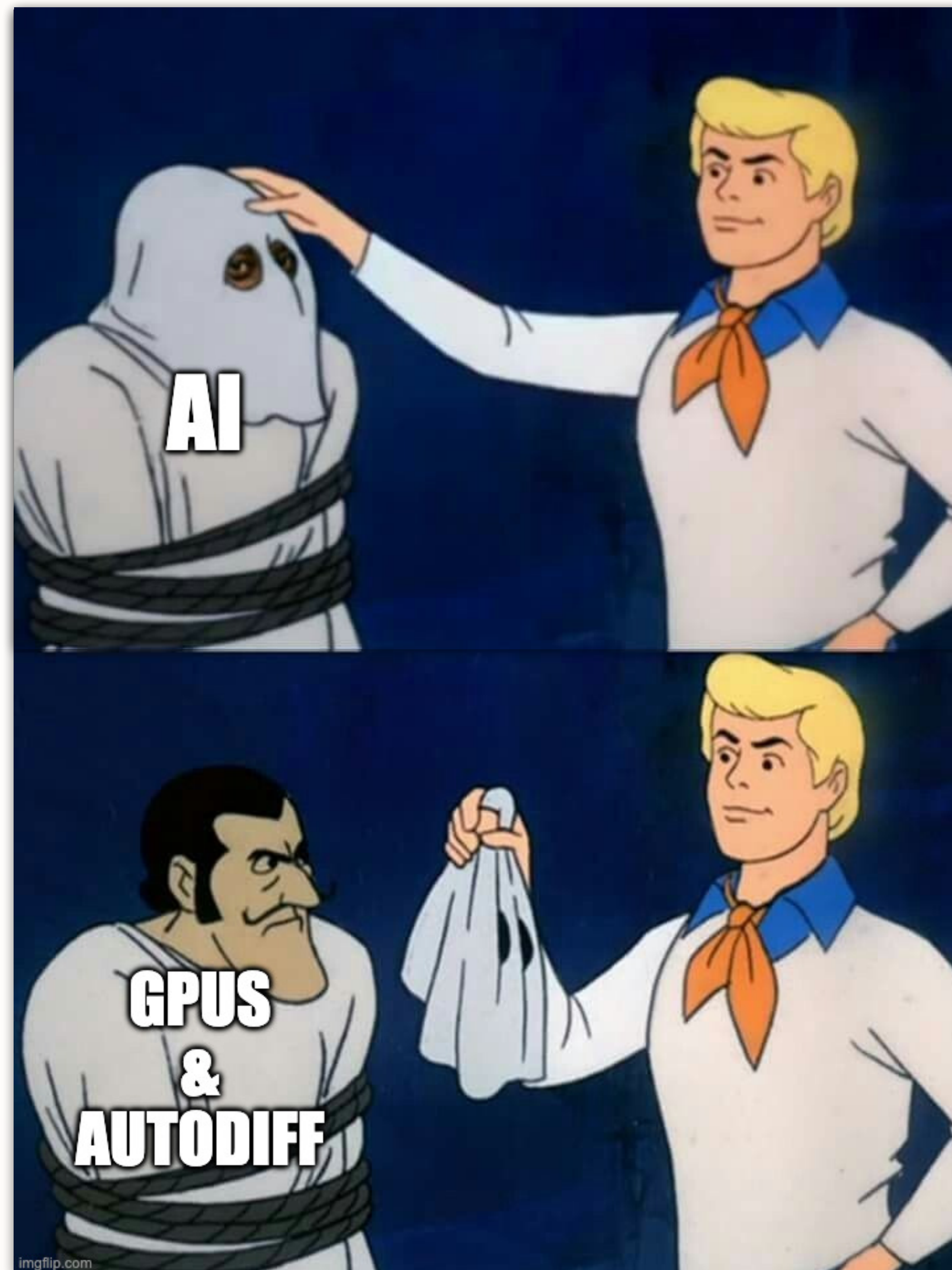
[Claici et al. 2018]





Machine Learning

[Claici et al. 2018]



Differentiable Geometry Processing?

Differentiating

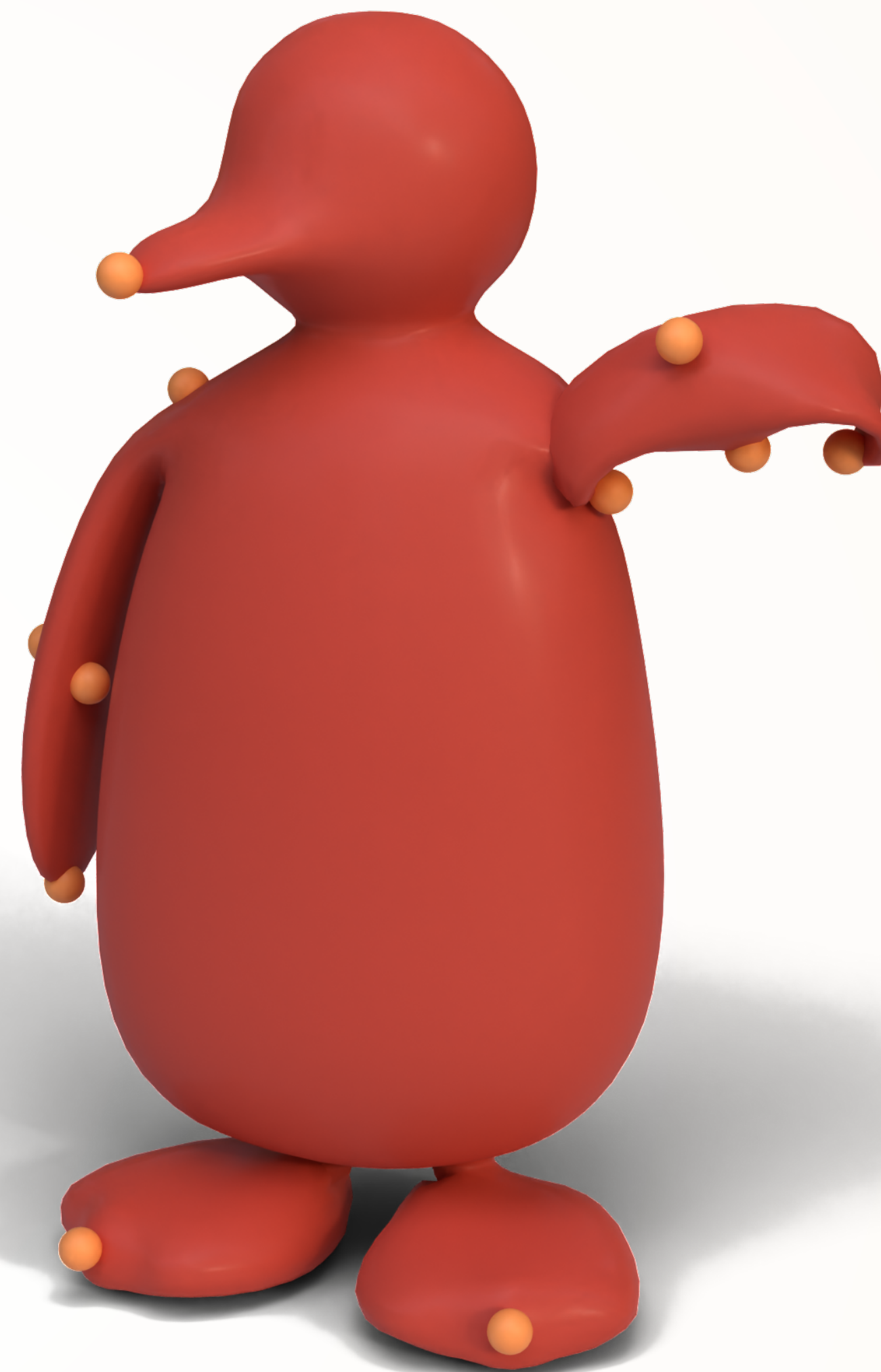
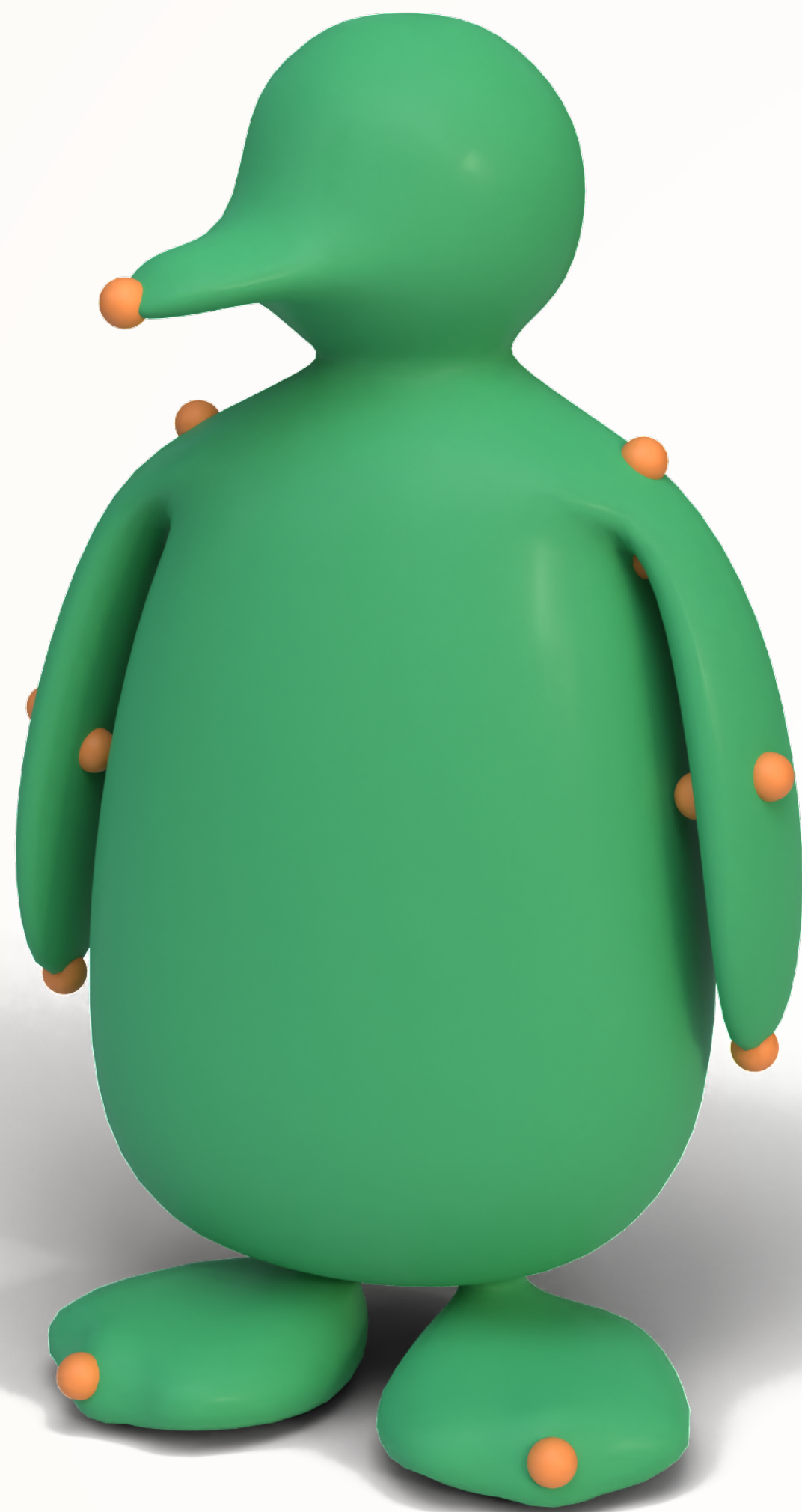
for | through

Geometry Processing

Differentiating

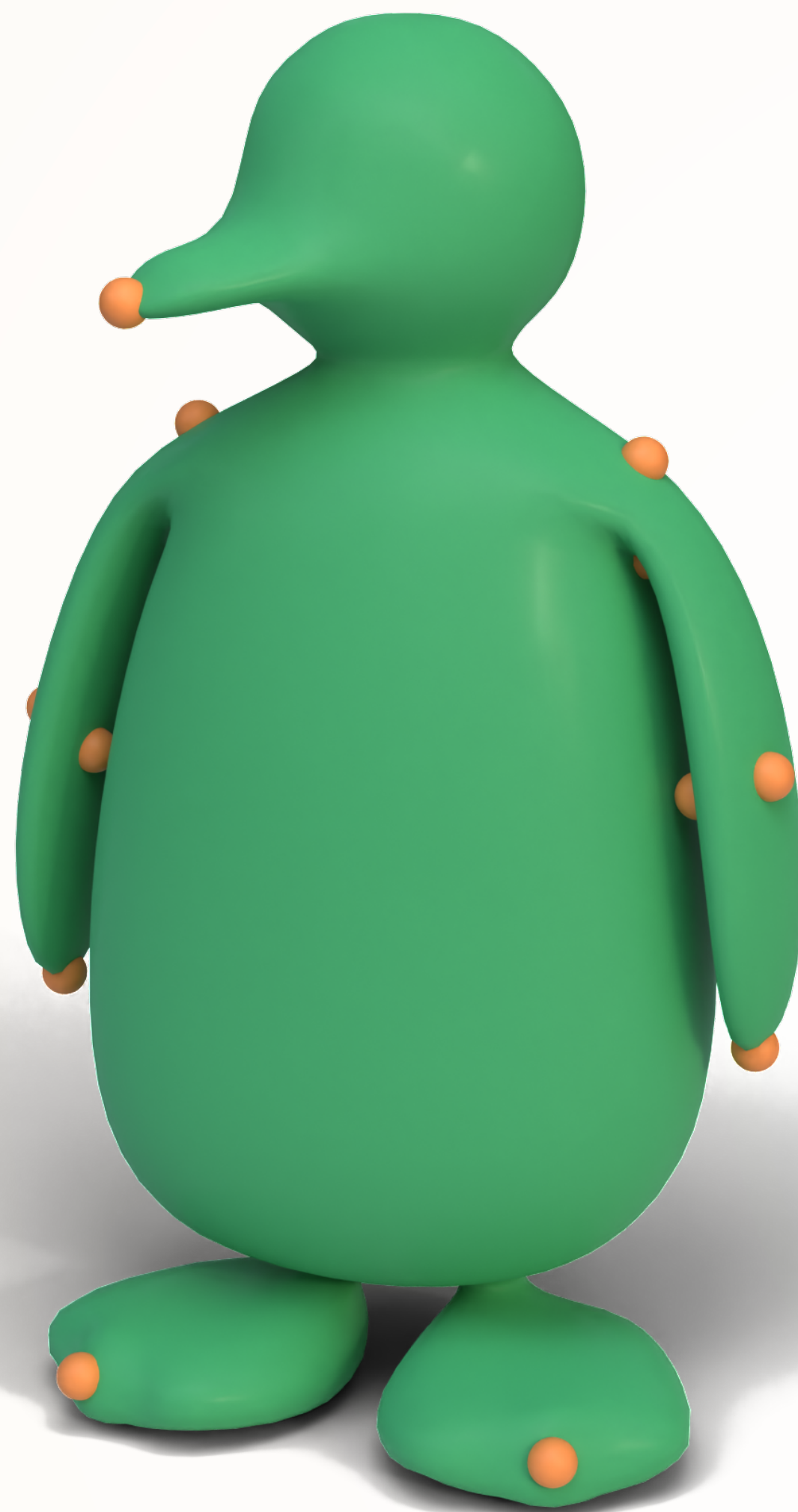
for

Geometry Processing



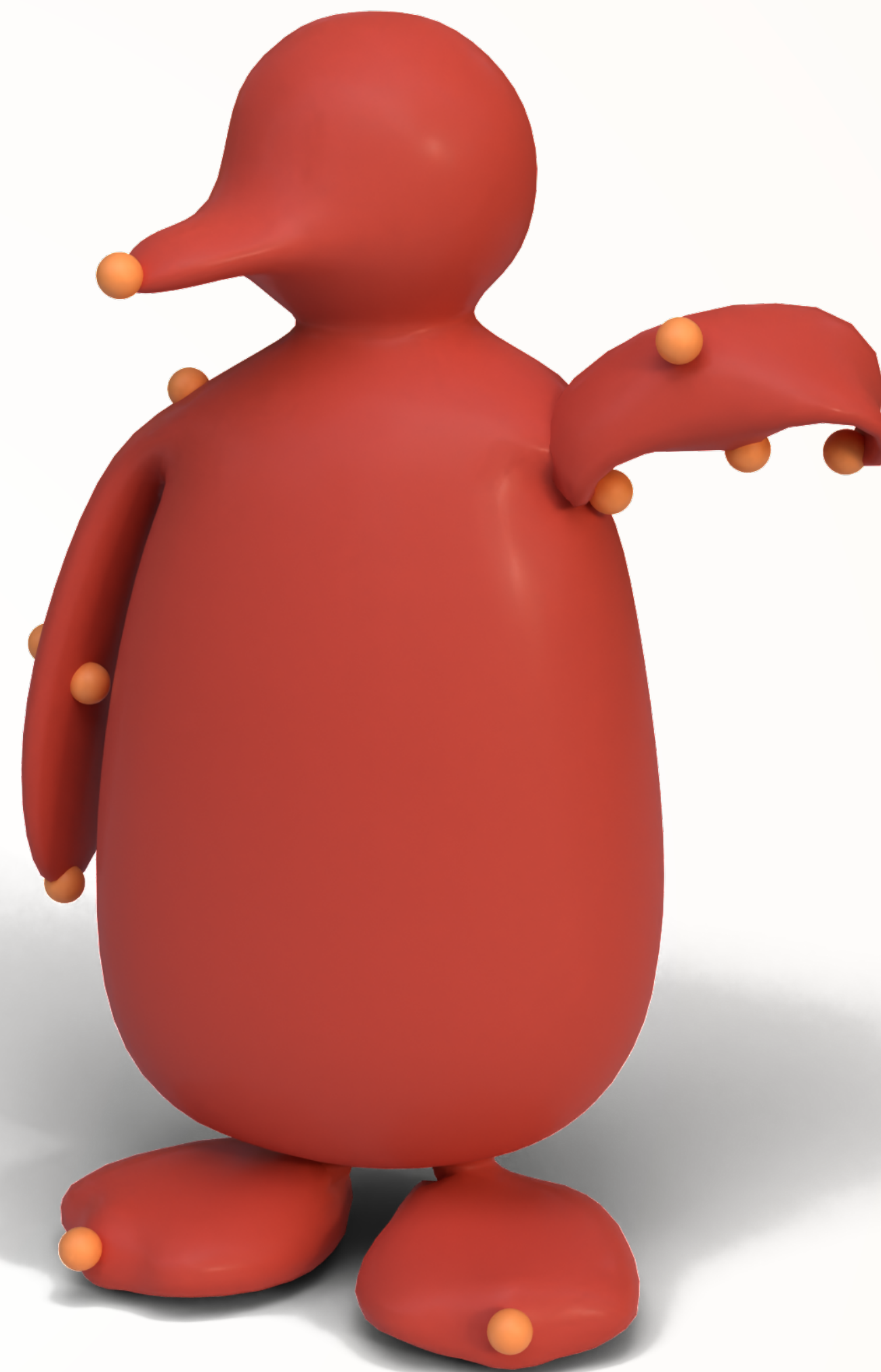
ARAP Deformation

Sorkine and Alexa [2007]



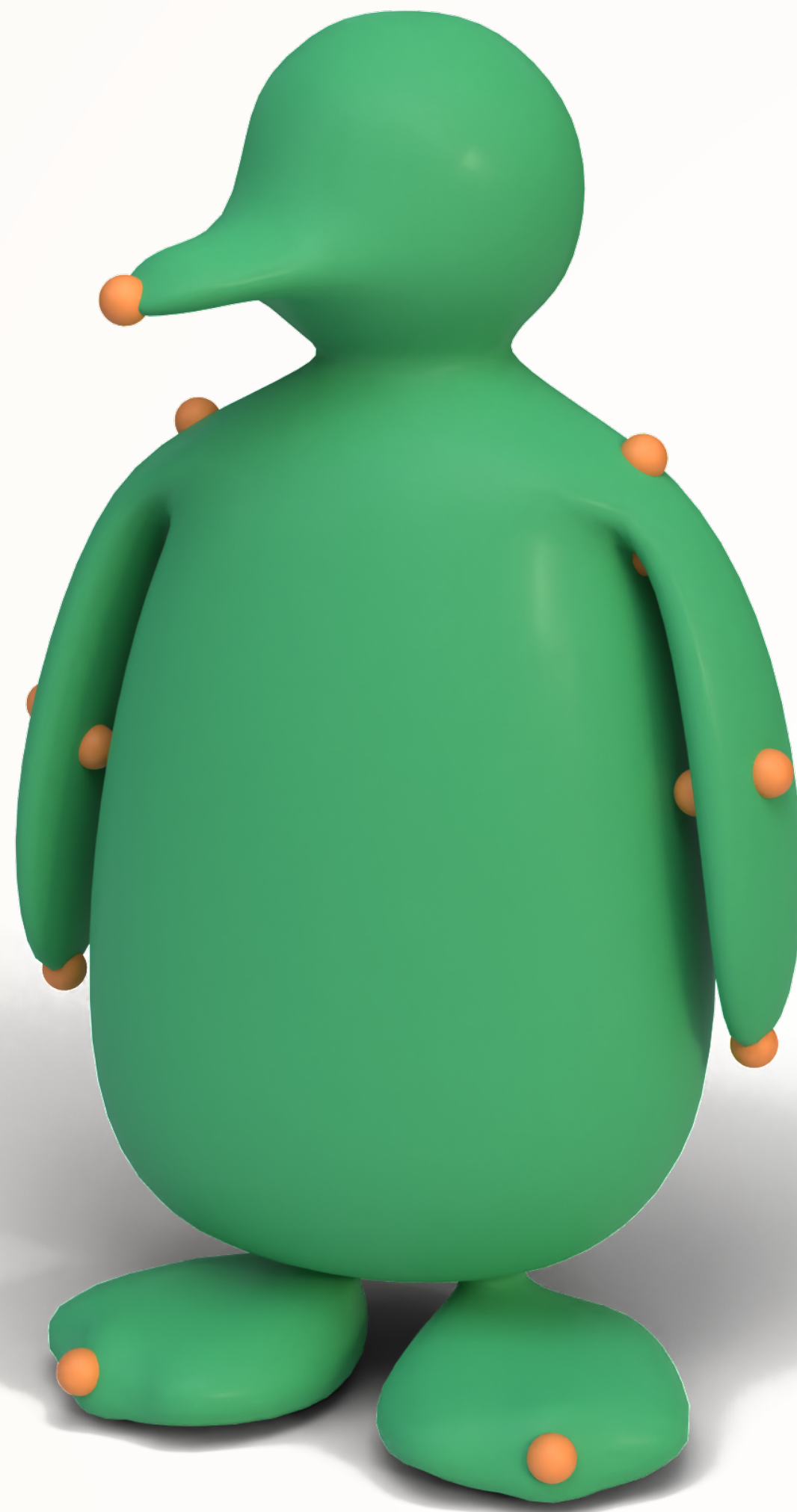
Rest Mesh

$$v_{i \in V}$$



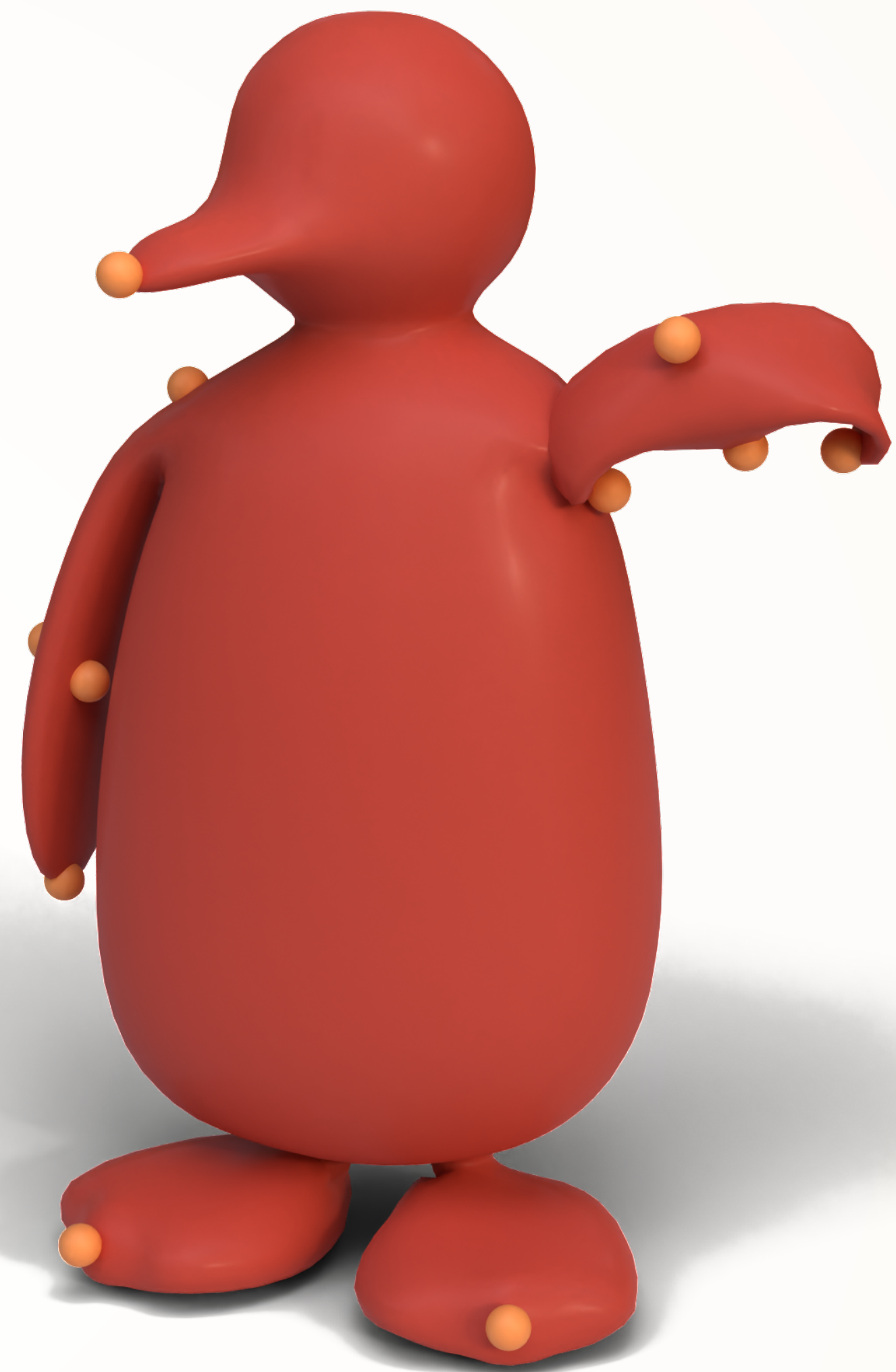
ARAP Deformation

Sorkine and Alexa [2007]



Rest Mesh

$$v_{i \in V}$$



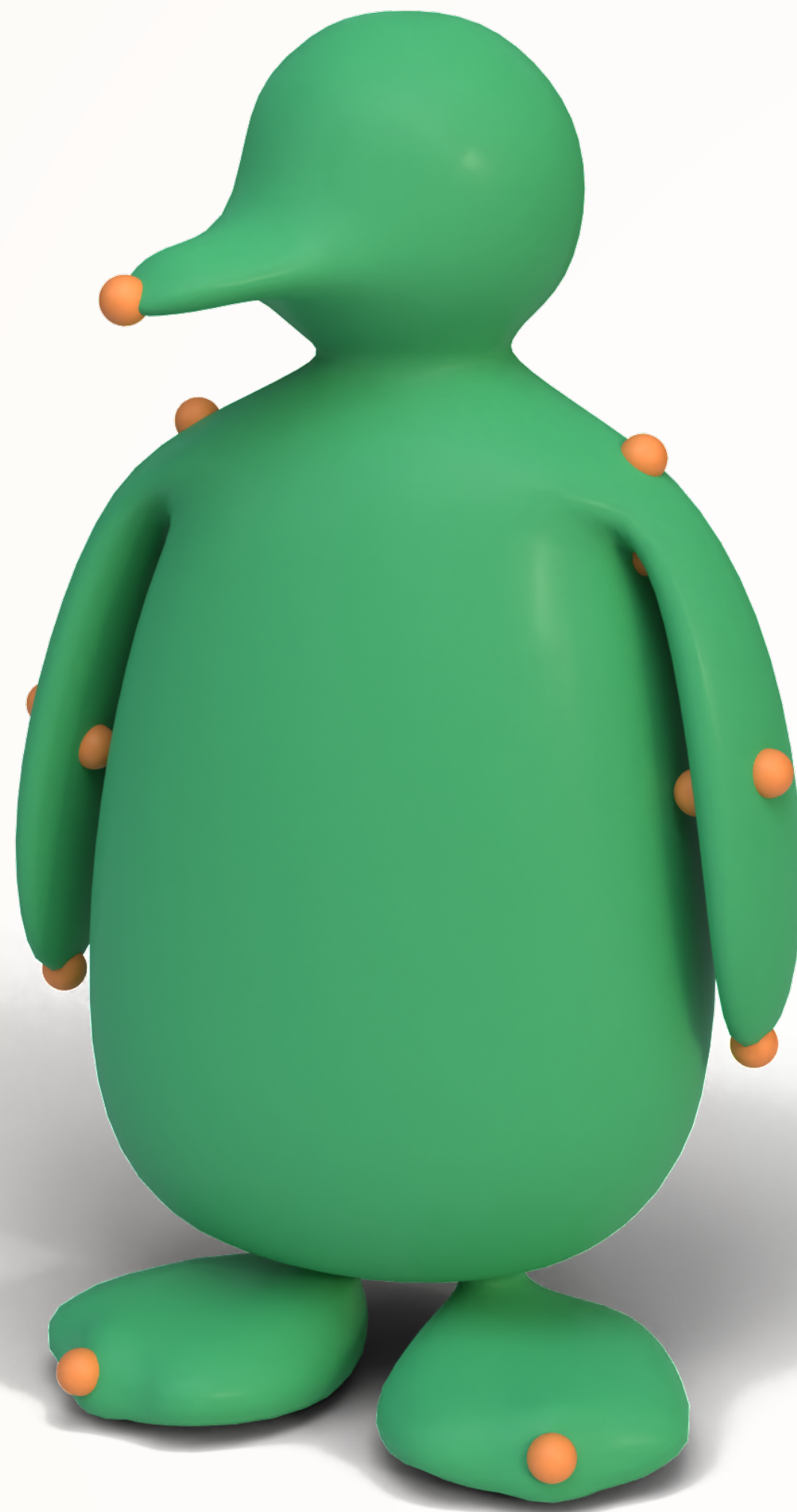
Deformed Mesh

$$y_{i \in V}$$

ARAP Deformation

Sorkine and Alexa [2007]

Control
Handles
 $h_{k \in \mathbb{H}}$



Rest Mesh
 $v_{i \in \mathbb{V}}$

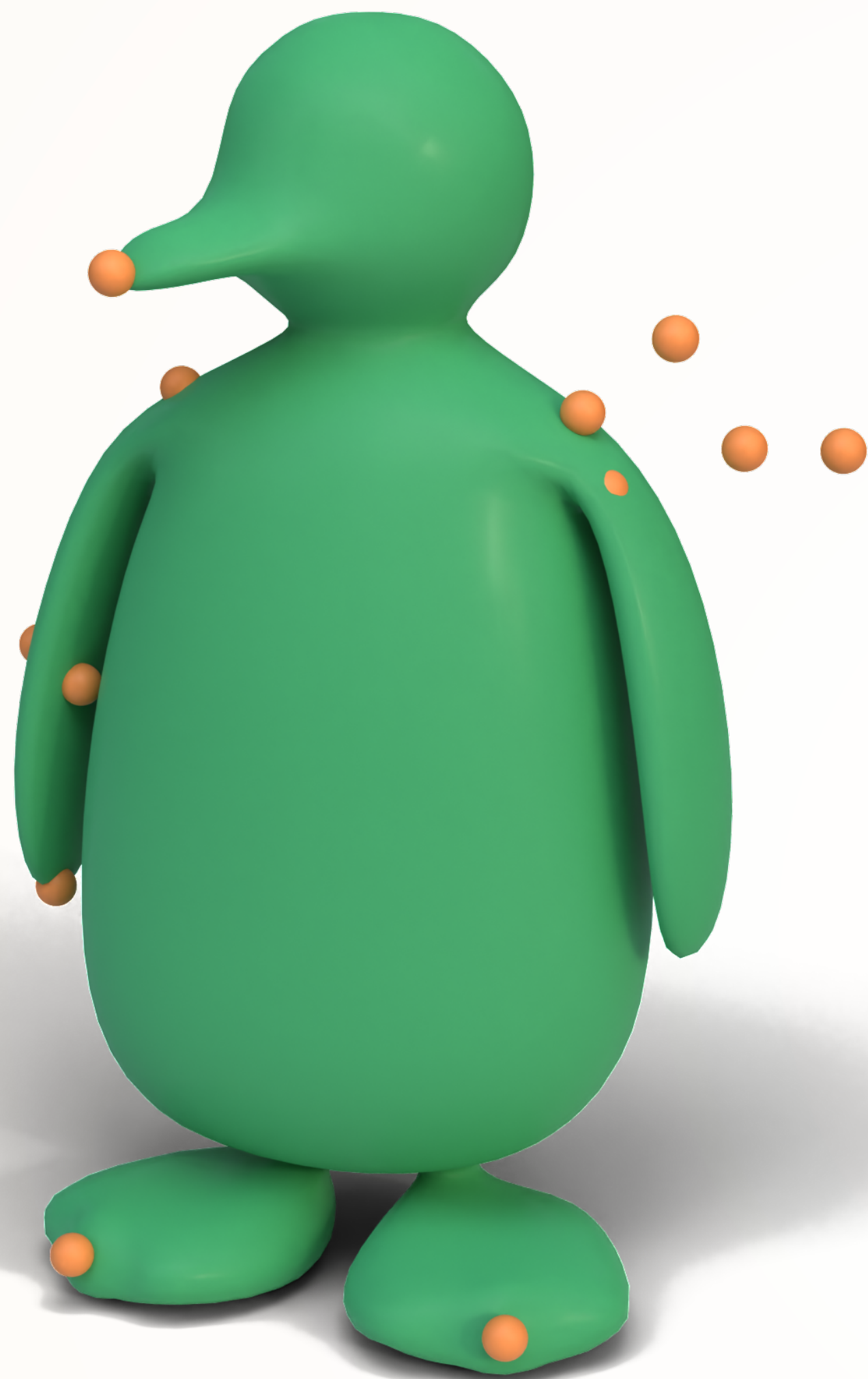


Deformed Mesh
 $y_{i \in \mathbb{V}}$

ARAP Deformation

Sorkine and Alexa [2007]

Control
Handles
 $h_{k \in \mathbb{H}}$



Rest Mesh
 $v_{i \in \mathbb{V}}$

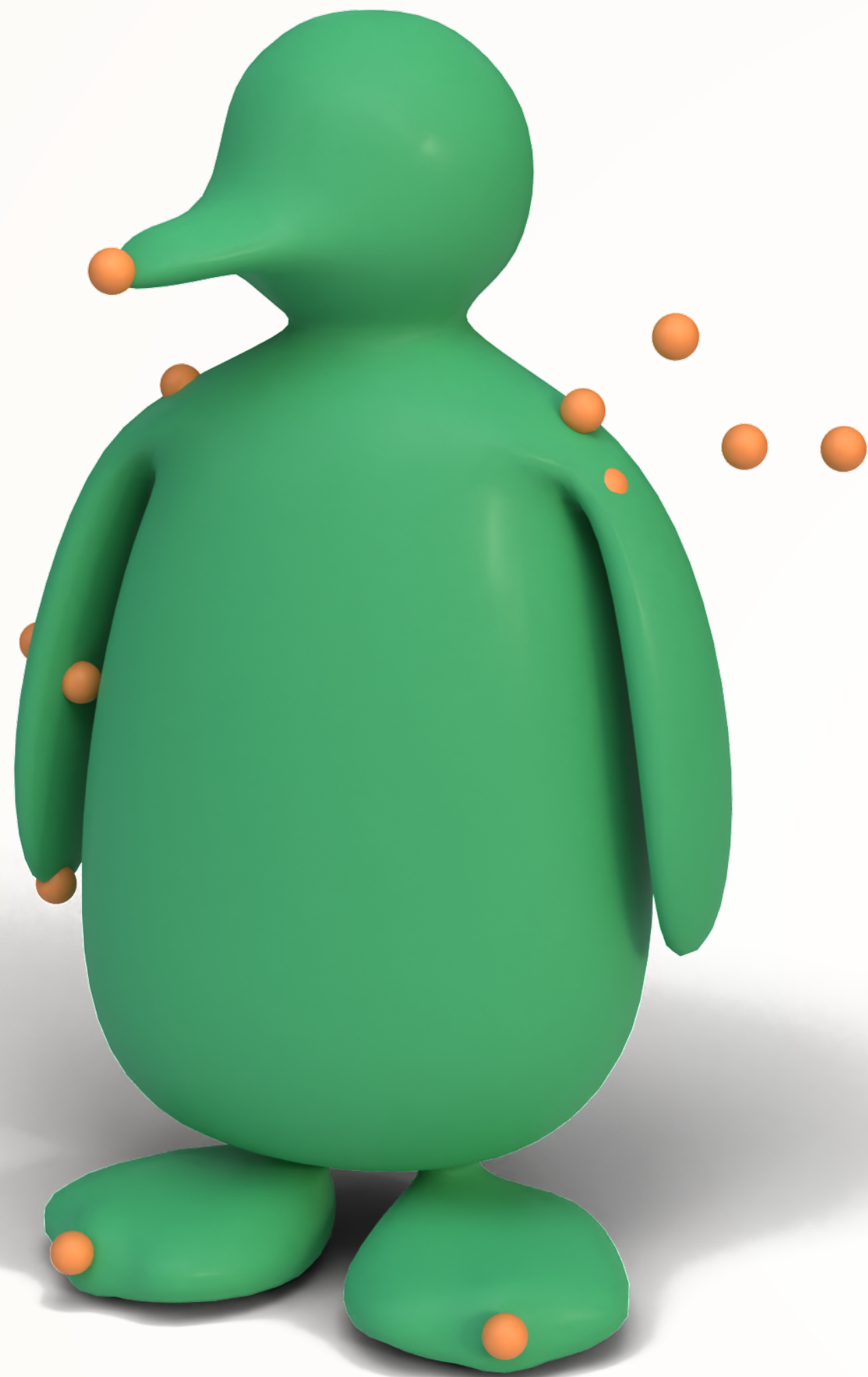


Deformed Mesh
 $y_{i \in \mathbb{V}}$

ARAP Deformation

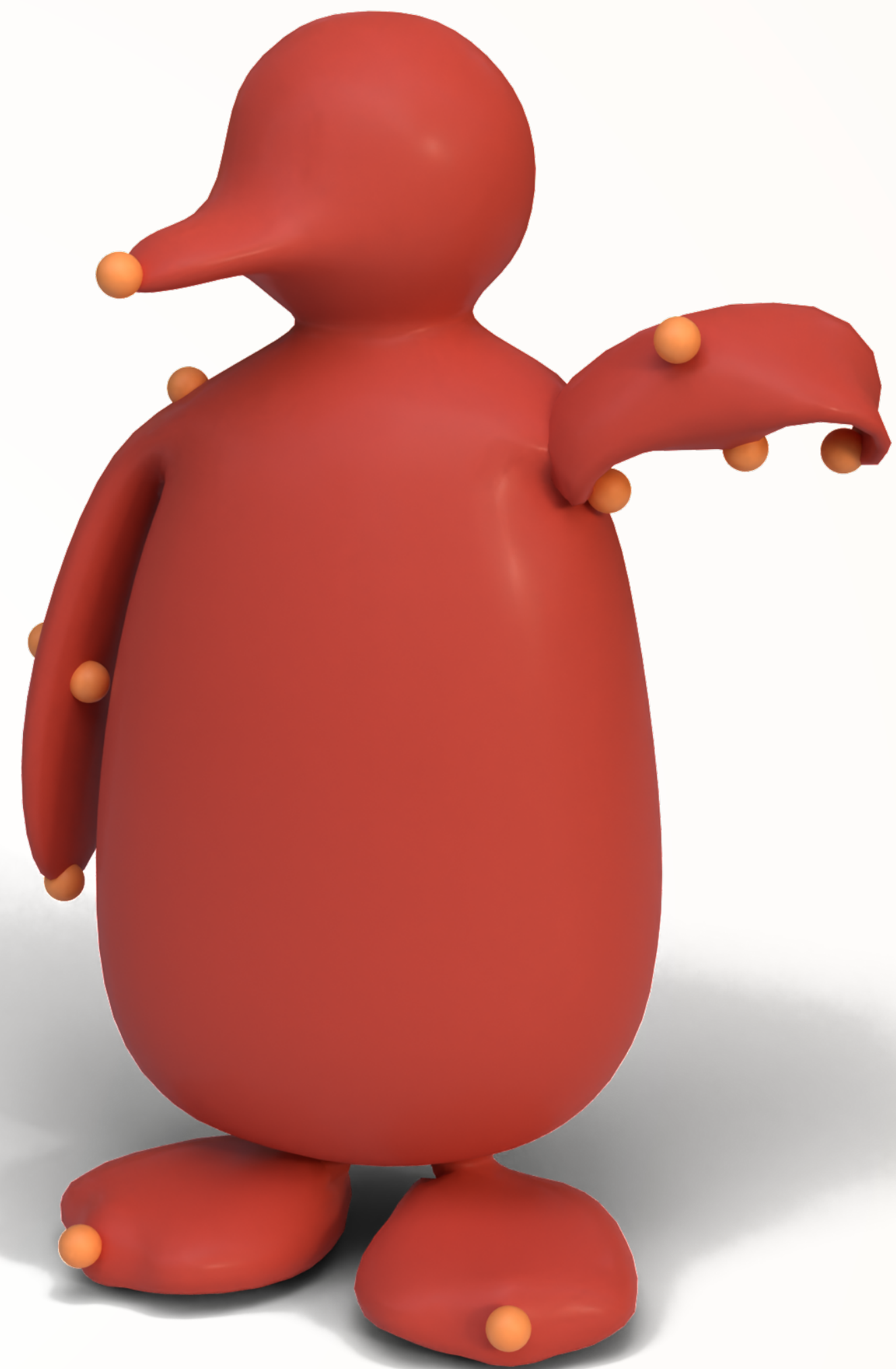
Sorkine and Alexa [2007]

Control
Handles
 $h_{k \in \mathbb{H}}$



Rest Mesh
 $v_{i \in \mathbb{V}}$

$$y \leftarrow g(h)$$



Deformed Mesh
 $y_{i \in \mathbb{V}}$

ARAP Deformation

Sorkine and Alexa [2007]

Control
Handles
 $\mathbf{h}_{k \in \mathbb{H}}$

$$\begin{aligned} & \mathbf{y} \leftarrow g(\mathbf{h}) \\ \operatorname{argmin}_{\mathbf{y}, R_i} & \sum_{(i,j) \in VV} w_{ij} \|(\mathbf{y}_i - \mathbf{y}_j) - R_i(\mathbf{v}_i - \mathbf{v}_j)\|^2, \\ \text{s. t. } & \mathbf{y}_k = \mathbf{h}_k, \forall k \in \mathbb{H}, \\ & R_i \in SO(3). \end{aligned}$$

Rest Mesh
 $\mathbf{v}_{i \in V}$

$$\mathbf{y} \leftarrow g(\mathbf{h})$$

Deformed Mesh
 $\mathbf{y}_{i \in V}$

ARAP Deformation

Sorkine and Alexa [2007]

Control
Handles
 $h_{k \in \mathbb{H}}$

$$\begin{aligned} & \mathbf{y} \leftarrow g(\mathbf{h}) \\ \operatorname{argmin}_{\mathbf{y}, R_i} & \sum_{(i,j) \in VV} w_{ij} \|(\mathbf{y}_i - \mathbf{y}_j) - R_i(\mathbf{v}_i - \mathbf{v}_j)\|^2, \\ \text{s. t. } & \mathbf{y}_k = \mathbf{h}_k, \forall k \in \mathbb{H}, \\ & R_i \in SO(3). \end{aligned}$$

solve “forward problem”
via gradient-based optimization

Rest Mesh
 $\mathbf{v}_{i \in V}$

$$\mathbf{y} \leftarrow g(\mathbf{h})$$

Deformed Mesh
 $\mathbf{y}_{i \in V}$

ARAP Deformation

Sorkine and Alexa [2007]

Differentiating

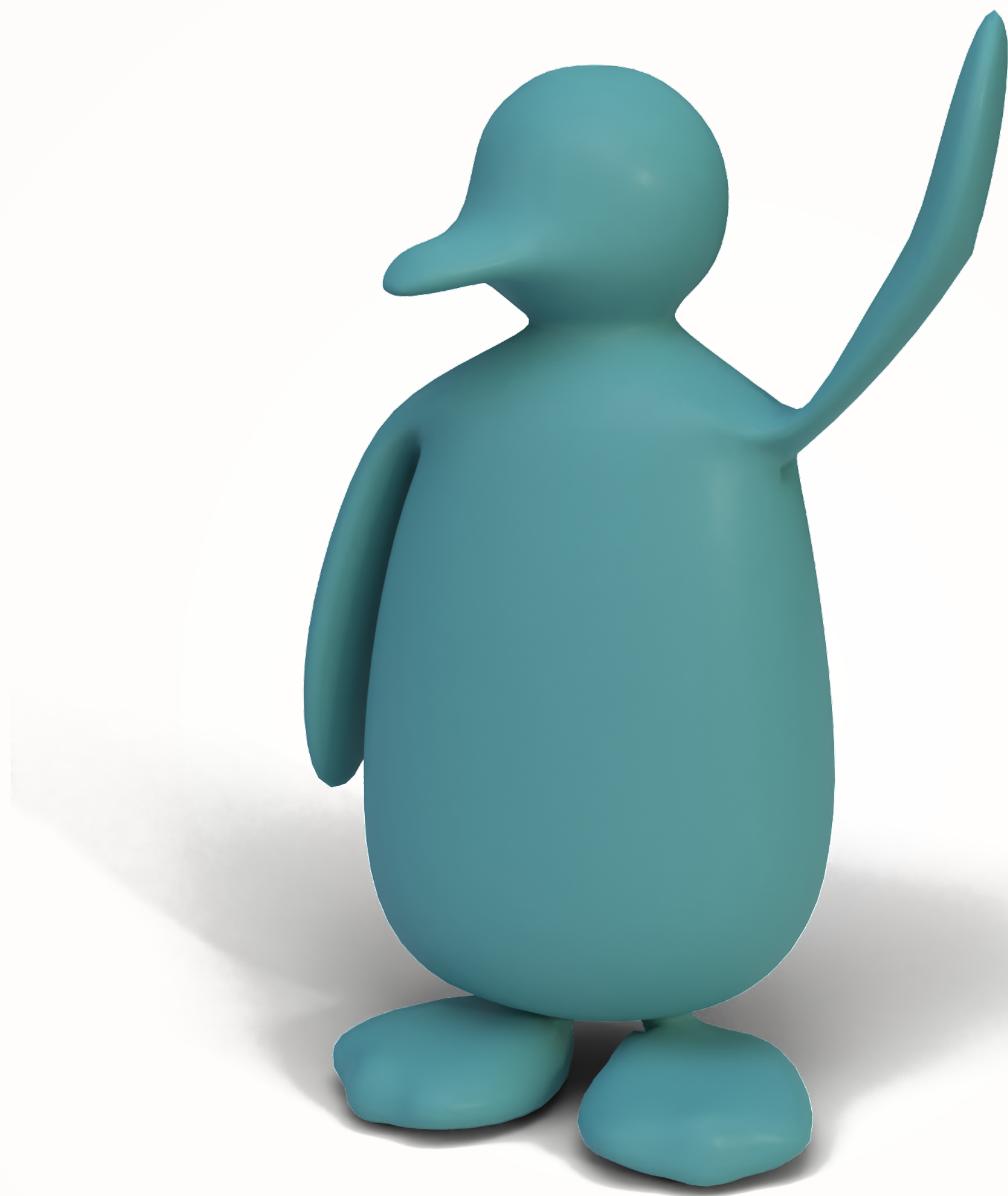
for | through

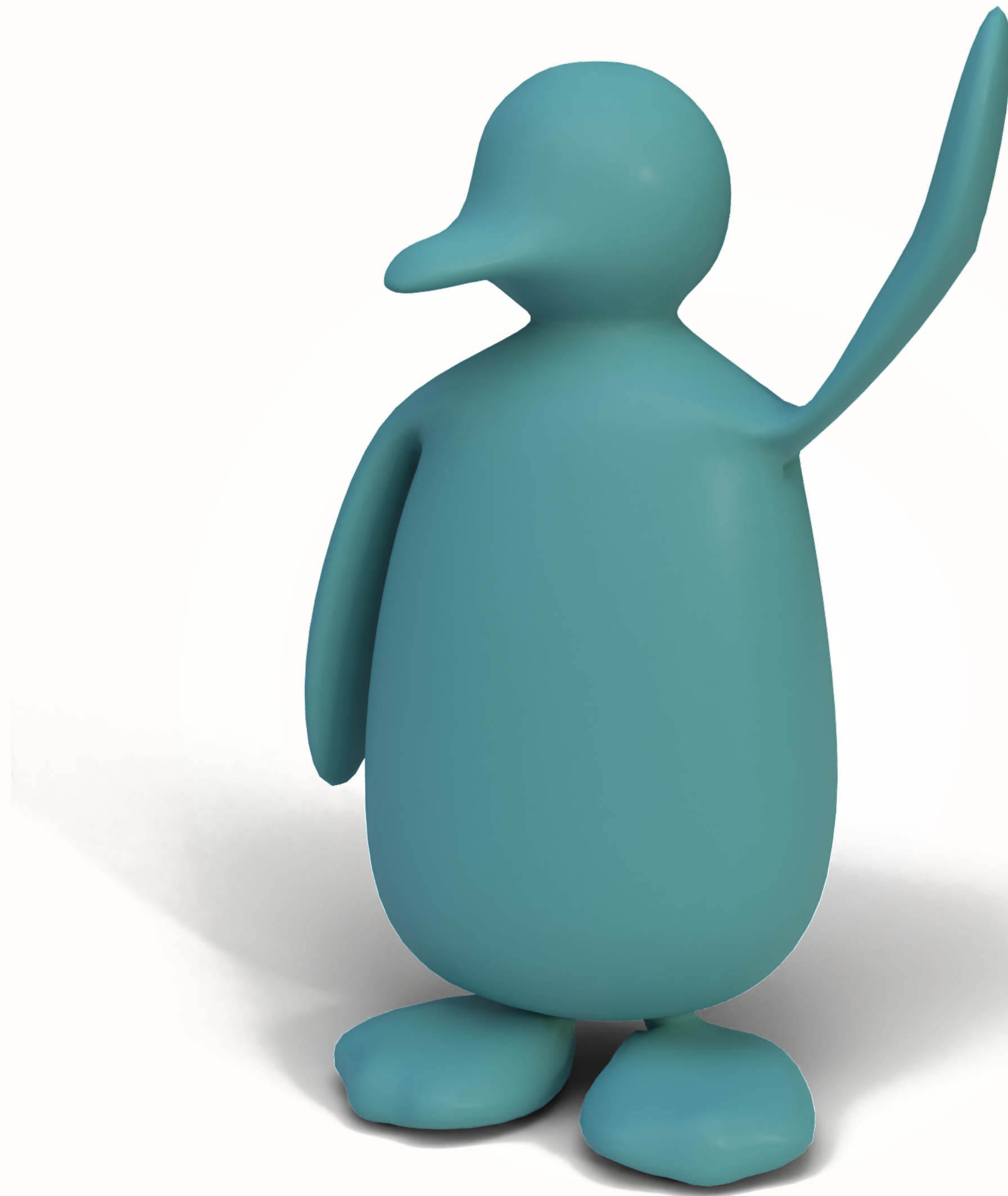
Geometry Processing

Differentiating

through

Geometry Processing





Target Mesh

$t_{i \in \mathbb{V}}$

Mocap Registration

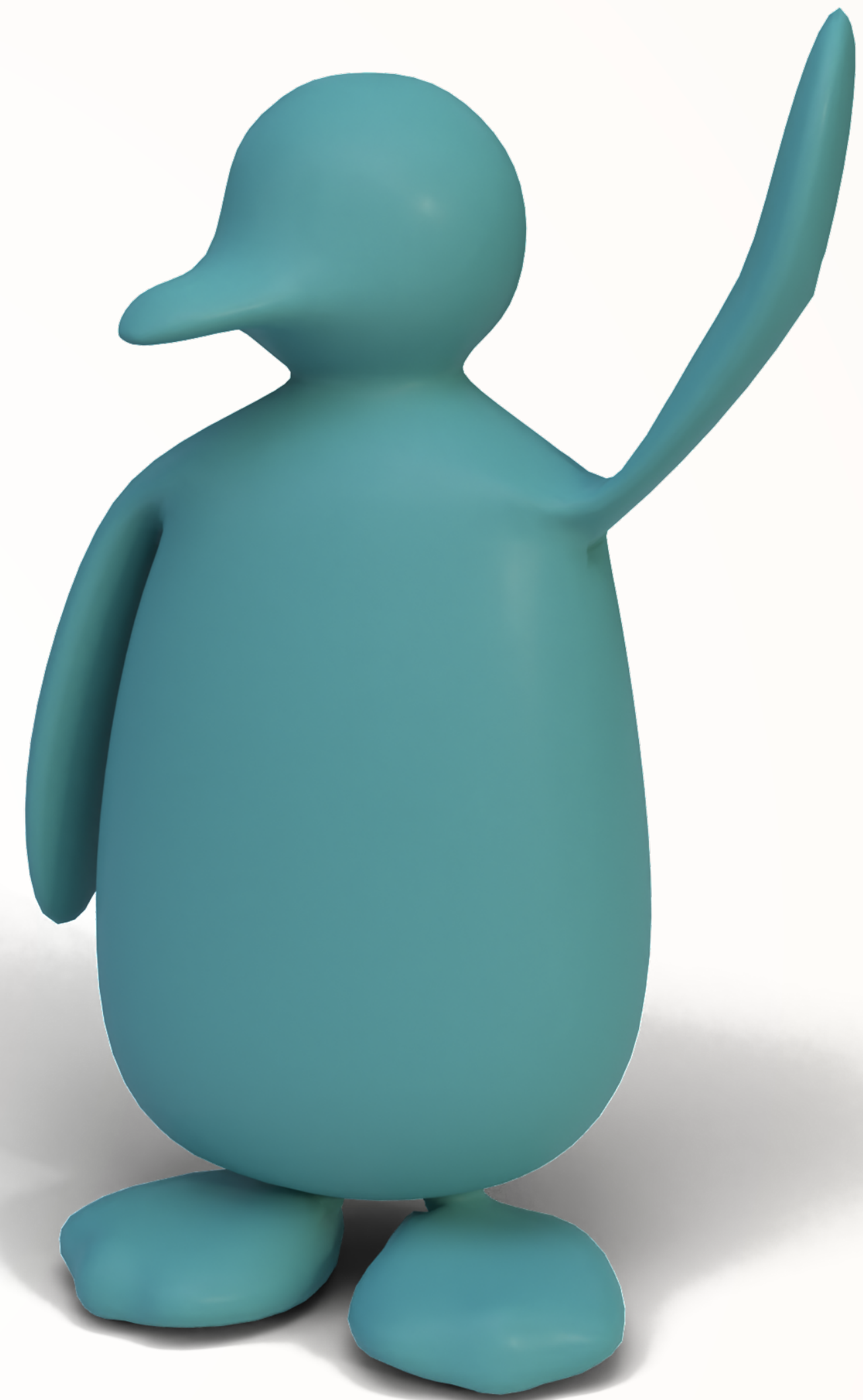


Control
Handles

$$h_{k \in \mathbb{H}}$$

Deformed Mesh

$$y \leftarrow g(h)$$



Target Mesh

$$t_{i \in \mathbb{V}}$$

Mocap Registration

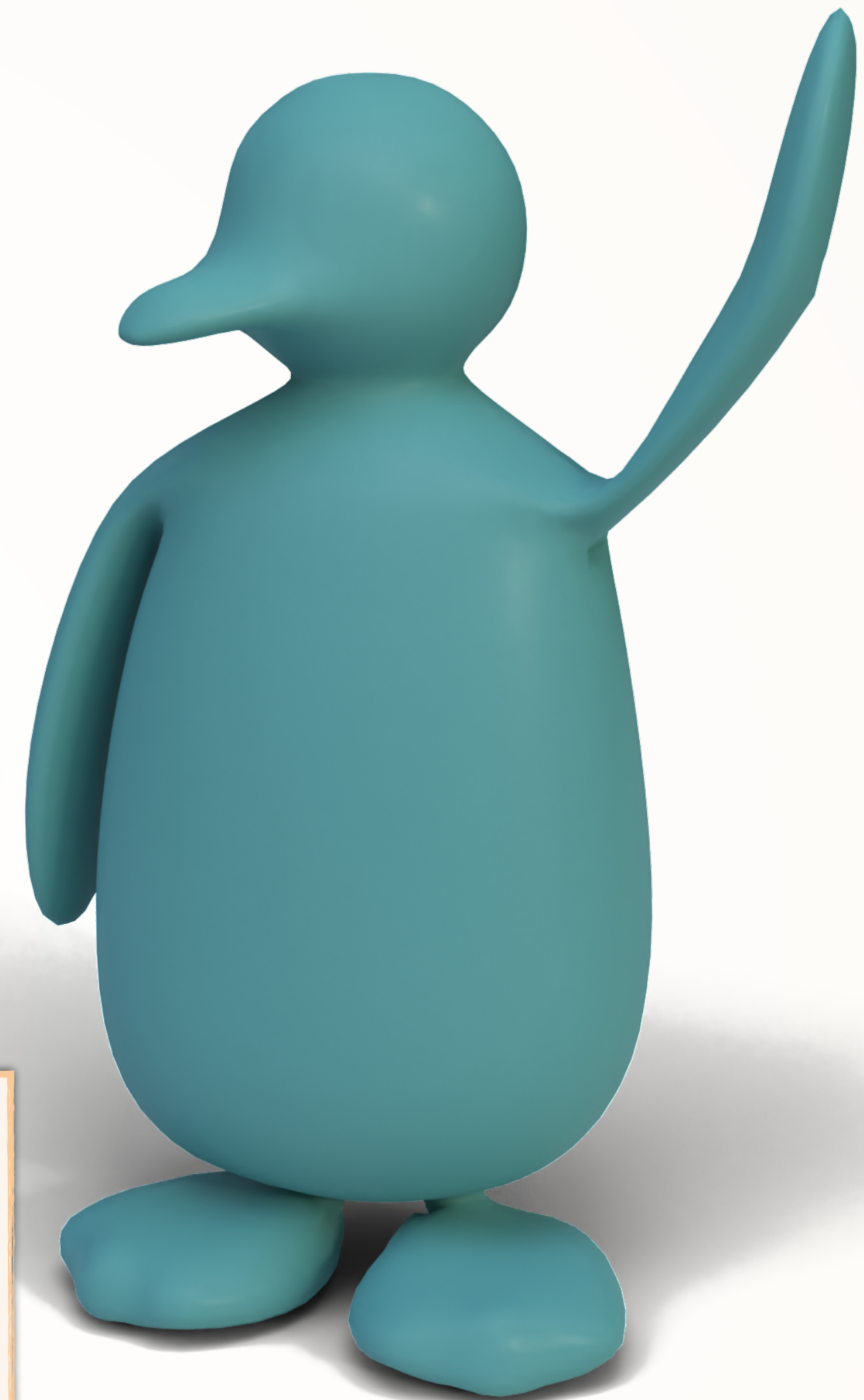


Control
Handles
 $h_{k \in \mathbb{H}}$

Deformed Mesh
 $y \leftarrow g(h)$

Goal: move control handles to
best approximate target mesh.

???



Target Mesh
 $t_{i \in \mathbb{V}}$

Mocap Registration

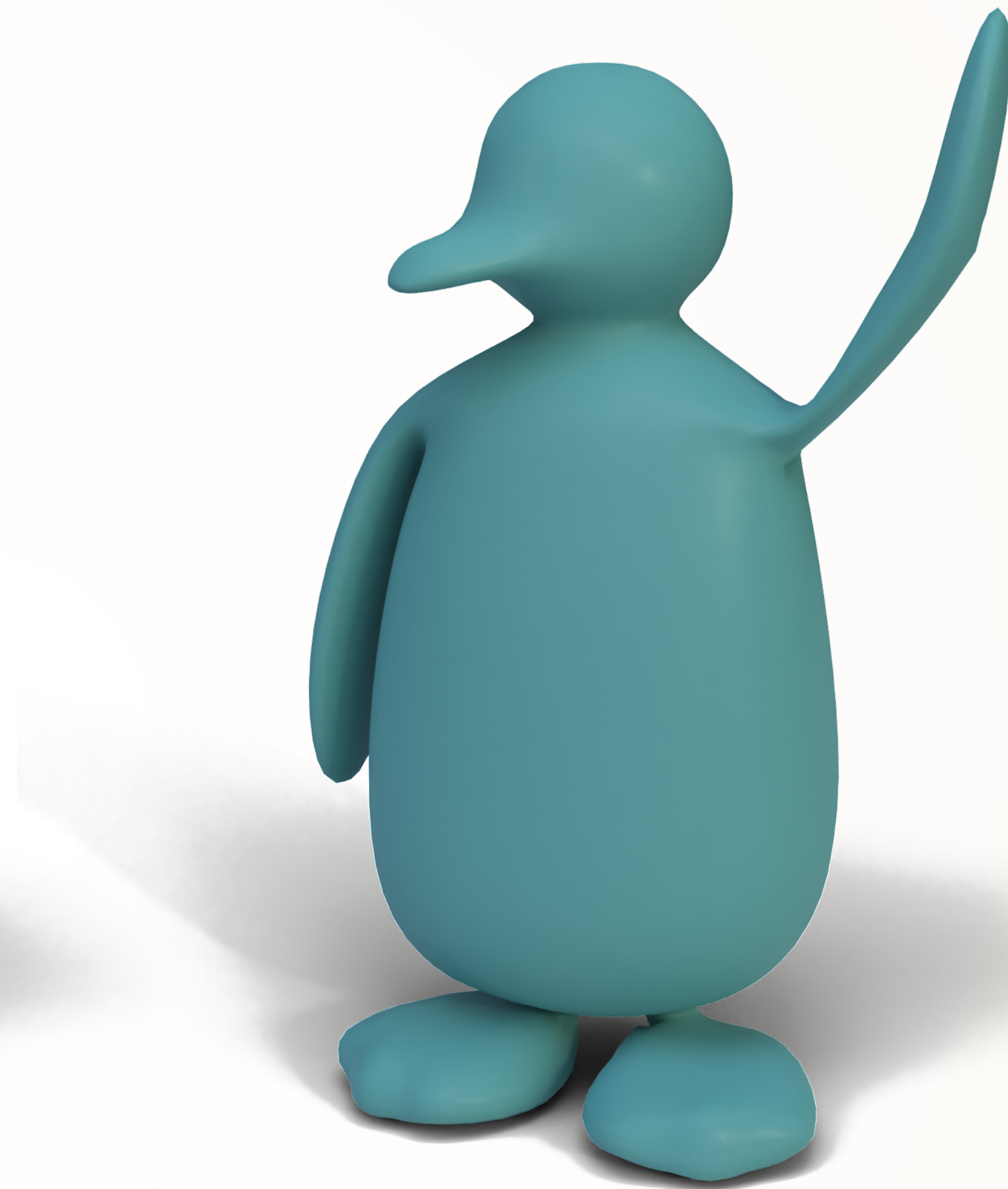


Control
Handles

$$h_{k \in \mathbb{H}}$$

Deformed Mesh

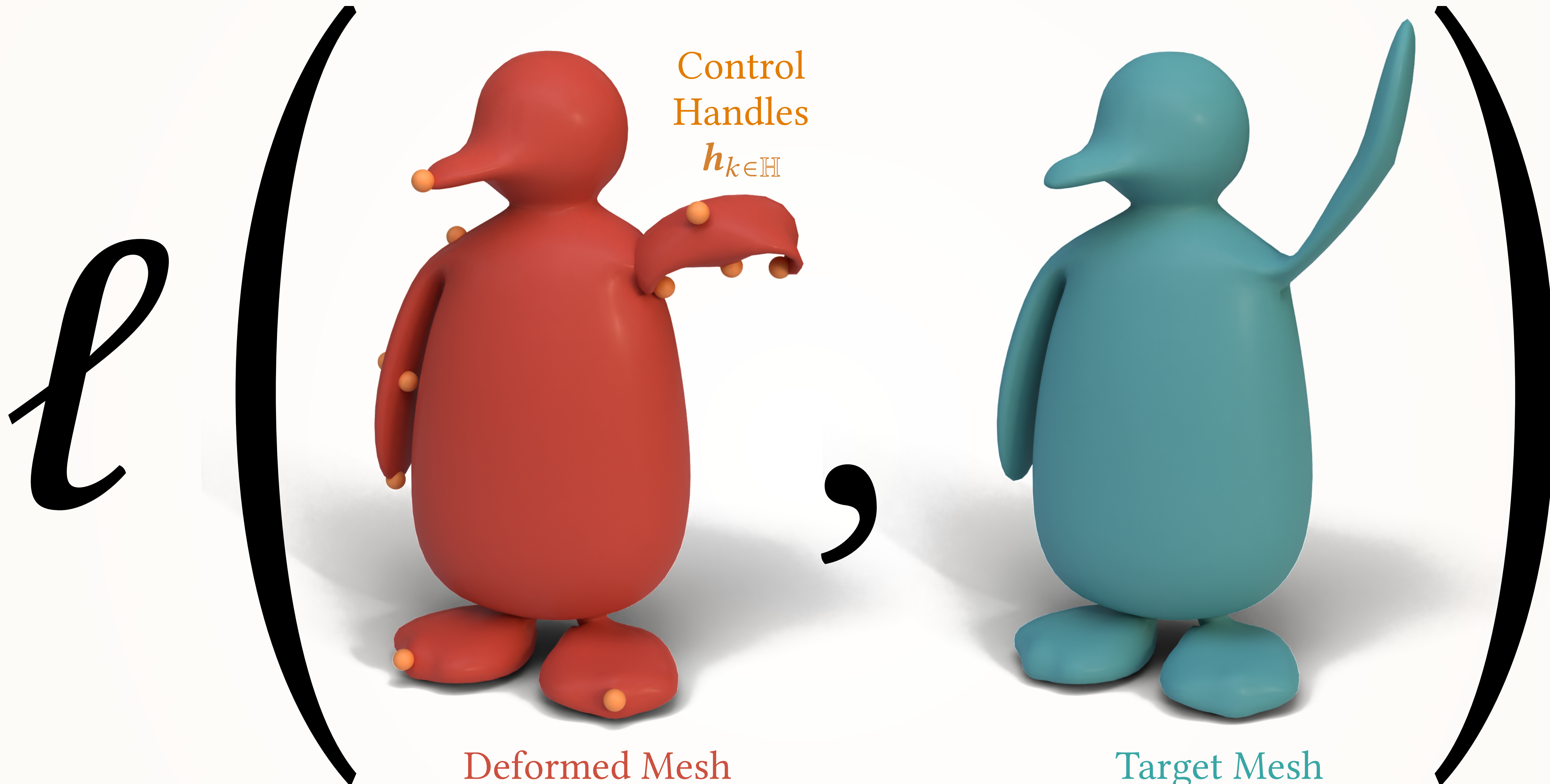
$$y \leftarrow g(h)$$



Target Mesh

$$t_{i \in \mathbb{V}}$$

Mocap Registration



ℓ

Control
Handles

$h_{k \in \mathbb{H}}$

$$\operatorname{argmin}_h \ell(g(h), t)$$

solve “inverse problem”
via gradient-based optimization

[problem: cannot autodiff through g]

Deformed Mesh

$$y \leftarrow g(h)$$

Target Mesh

$$t_{i \in \mathbb{V}}$$

Mocap Registration

Geometry in Style: 3D Stylization via Surface Normal Deformation

Nam Anh Dinh
University of Chicago

Itai Lang
University of Chicago

Hyunwoo Kim
University of Chicago

Oded Stein
University of Southern California

Rana Hanocka
University of Chicago

a 3d render of...



a pineapple-themed

Figure 1. Our method deforms the geometry so that they preserve the identity

Neural Jacobian Fields: Learning Intrinsic Mappings of Arbitrary Meshes

NOAM AIGERMAN, Adobe Research, USA

KUNAL GUPTA*, University of California San Diego, USA

VLADIMIR G. KIM, Adobe Research, USA

SIDDHARTHA CHAUDHURI, Adobe Research, IIT Bombay, India

JUN SAITO, Adobe Research, USA

THIBAUT GROUEIX, Adobe Research, USA

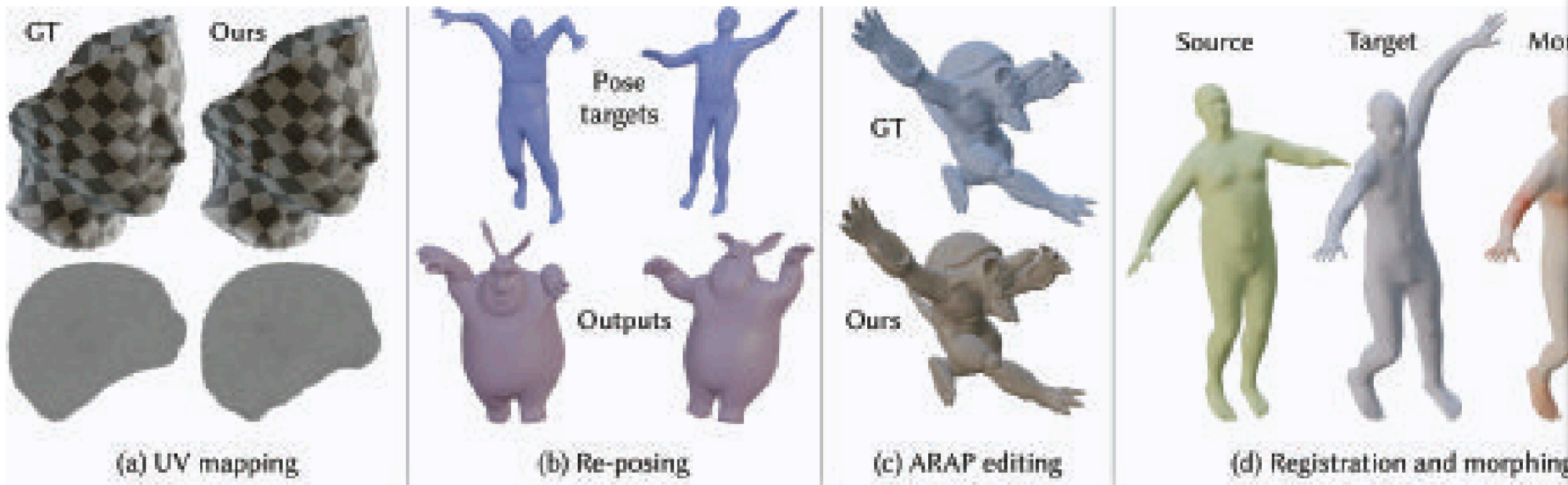


Fig. 1. We present a versatile, detail-preserving and triangulation-agnostic neural framework for learning piecewise-linear mappings of meshes. In the *exact same core framework*, we show, from left to right: (a) learning to UV-map arbitrary surfaces; (b) re-posing a novel character (the bunny), using a model trained only on humans; (c) learning to replicate As-Rigid-As-Possible [Saito et al. 2017] deformations; and (d) morphing a human shape (green) to a target pose and body shape (blue), without input correspondences.

HodgeNet: Learning Spectral Geometry on Triangle Meshes

DMITRIY SMIRNOV and JUSTIN SOLOMON, Massachusetts Institute of Technology, USA



MIT animation dataset. Each mesh in the dataset contains 20,000 faces (10,000 vertices). We show an

Fast Quasi-Harmonic Weights for Geometric Data Interpolation

YU WANG and JUSTIN SOLOMON, Massachusetts Institute of Technology, USA

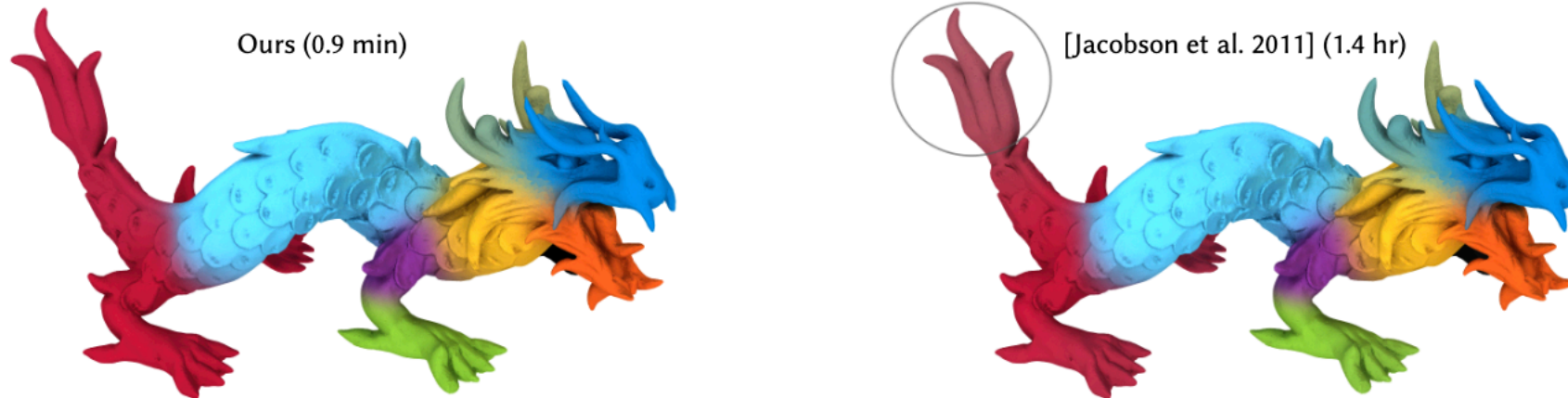


Fig. 1. Color interpolation using our skinning weights and bounded biharmonic weights [Jacobson et al. 2011]. Note the latter weights (and thus the blended color) have a local extrema at the tail of the Asian dragon. Our weights are free from local extrema and have a slightly lower smoothness energy. More importantly, our weights are many orders of magnitude faster to compute. Previous methods can take hours on this model, which has more than one million tetrahedra. Our method takes less than a minute, with potential for further speedups.

We propose *quasi-harmonic weights* for interpolating geometric data, which are orders of magnitude faster to compute than state-of-the-art. Currently, interpolation (or, skinning) weights are obtained by solving large-scale constrained optimization problems with explicit constraints to suppress oscillative patterns, yielding smooth weights only after a substantial amount of computation time. As an alternative, our weights are obtained as minima of an unconstrained problem that can be optimized quickly using straight-forward numerical techniques. We consider weights that can be obtained as solutions to a parameterized family of second-order elliptic partial differential equations. By leveraging the maximum principle and careful parameterization, we pose weight computation as an inverse problem of recovering

however, motivates automatic computation of skinning weights as a critical way to make the animation pipeline more efficient. Beyond their original application in animation, however, automatic skinning weight computation has found application in graphics pipelines beyond animation as a generic tool to interpolate geometric and physical quantities smoothly across geometric domains.

The problem of computing skinning weights amounts to designing a *partition of unity*, or set of functions that sums to 1 at every point on the domain, that satisfies a few properties. There is one skinning function per control handle of a deforming shape, which typically equals 1 at the handle and decays as we move further away

Who is this course for?

- The **machine learning curious** geometers.

Who is this course for?

- The **machine learning curious** geometers.
- The **geometry curious** machine learning practitioners.

Who is this course for?

- The **machine learning curious** geometers.
- The **geometry curious** machine learning practitioners.
- Interest in **theory and practice** of differentiable programming.

- The **machine learning curious** geometers.
- The **geometry curious** machine learning practitioners.
- Interest in **theory and practice** of differentiable programming.
- Familiar with n-dimensional array Python libraries.

- The **machine learning curious** geometers.
- The **geometry curious** machine learning practitioners.
- Interest in **theory and practice** of differentiable programming.
- Familiar with n-dimensional array Python libraries.
- Somewhat familiar with “forward” geometry processing.

Who is this course for?

- **Chapter 1:** Introduction to Automatic Differentiation
- **Chapter 2:** Differentiable Geometry Processing in Python
- **Chapter 3:** Introduction to The Adjoint Method
- **Chapter 4:** Inverse Geometry Processing in Python

📖 README



iskra

iskra ✨ Modern Geometry Processing

Lightweight geometry processing library that is a one-stop shop for all your geometric needs. Iskra is:

- modern and Python-first,
- simple by default, powerful when needed,
- fully differentiable and compatible with machine learning,
- CPU and GPU enabled,
- actively maintained.

Support the project by starring it!  Stars 

⚠️ Caution

This is a pre-release. We are actively working on a user guide, documentation, and setting up on PyPI.

README

 **iskra**

iskra ✨ Modern Geometry Processing

Lightweight geometry processing library that is a one-stop shop for all your geometric needs. Iskra is:

- modern and Python-first,
- simple by default, powerful when needed,
- fully differentiable and compatible with machine learning,
- CPU and GPU enabled,
- actively maintained.

Support the project by starring it!  Stars  repo not found

 **Caution**

This is a pre-release. We are actively working on a user guide, documentation, and setting up on PyPI.

☆ Star 1 ▼



Introduction to Automatic Differentiation

Gradient-based Optimization

We focus on problems where we minimize a loss/energy

$$\min_{\theta} \ell(\theta),$$

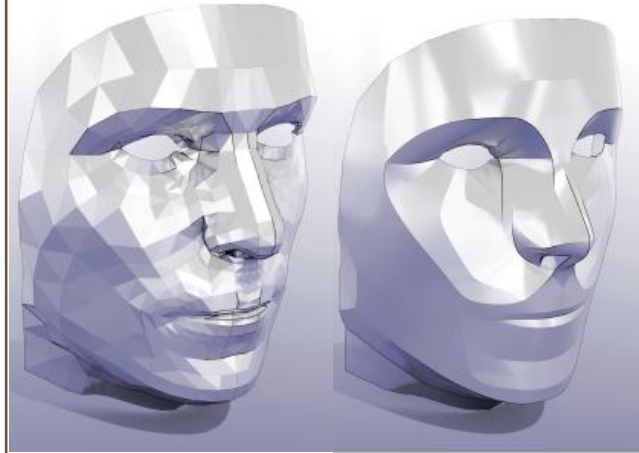
where θ are the optimization variables and L is the loss/energy function

Parametrization



[Rabinovich et al., 2017]

Developable Surfaces



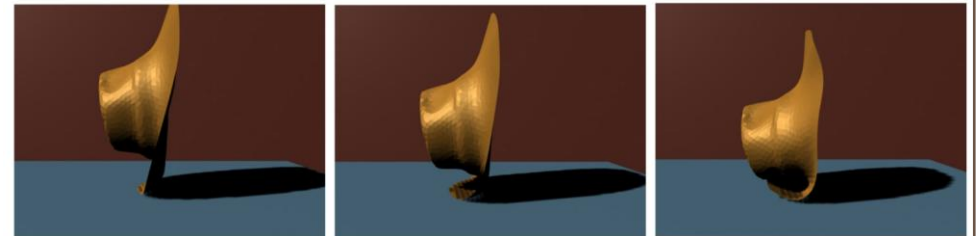
[Stein et al., 2018]

*Frame Field
Optimization*



[Schmidt et al., 2022]

Shell Simulation

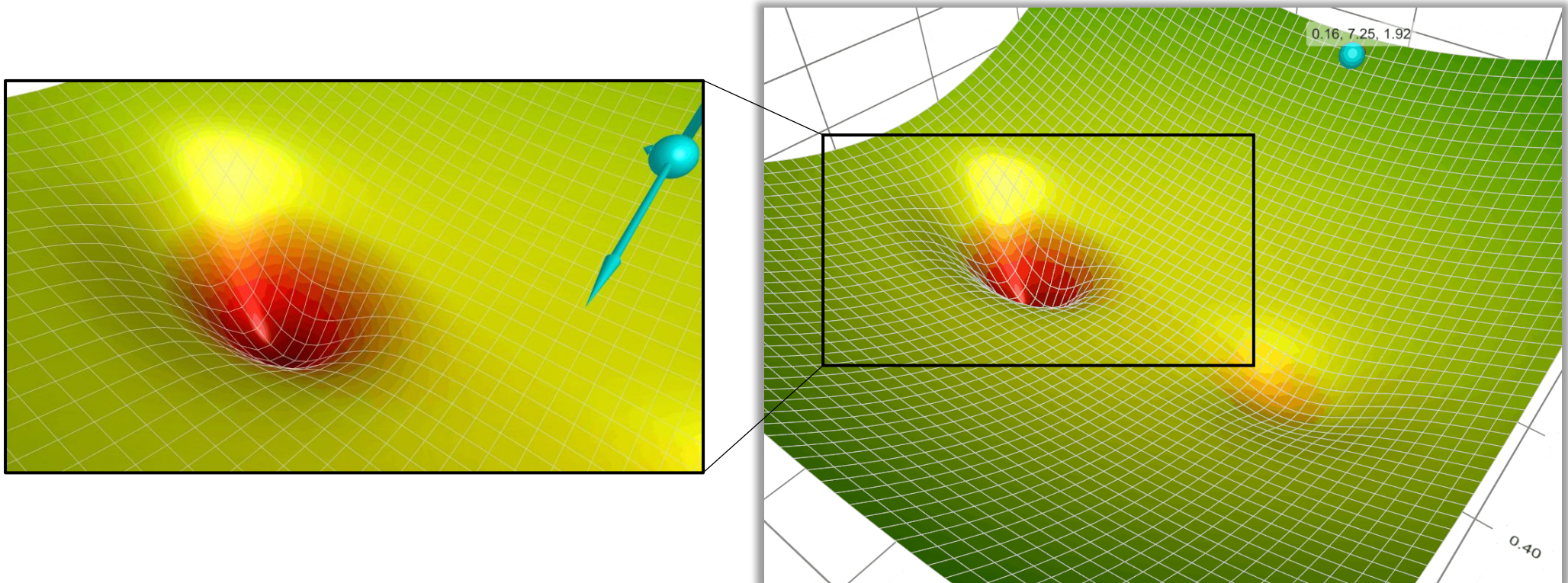


[Grinspun et al., 2003]

Gradient descent

Gradient descent moves the parameters θ in the negative-gradient direction

$$\theta_{k+1} = \theta_k - \eta \nabla \ell(\theta_k)$$



Gradients and Jacobians

Scalar function:

$$L: \mathbb{R}^n \rightarrow \mathbb{R}$$

$$\nabla L \in \mathbb{R}^n$$

$$\nabla L(\theta) = \begin{bmatrix} \frac{\partial L}{\partial \theta_1} \\ \vdots \\ \frac{\partial L}{\partial \theta_n} \end{bmatrix}$$

Vector function:

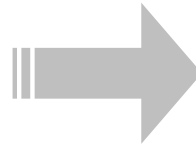
$$F: \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$J_F = \frac{\partial F}{\partial \theta} \in \mathbb{R}^{m \times n}$$

$$J_F = \frac{\partial F}{\partial \theta} = \begin{bmatrix} \nabla f_1^T \\ \vdots \\ \nabla f_m^T \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \cdots & \frac{\partial f_1}{\partial \theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial \theta_1} & \cdots & \frac{\partial f_m}{\partial \theta_n} \end{bmatrix}$$

Sparse Jacobians

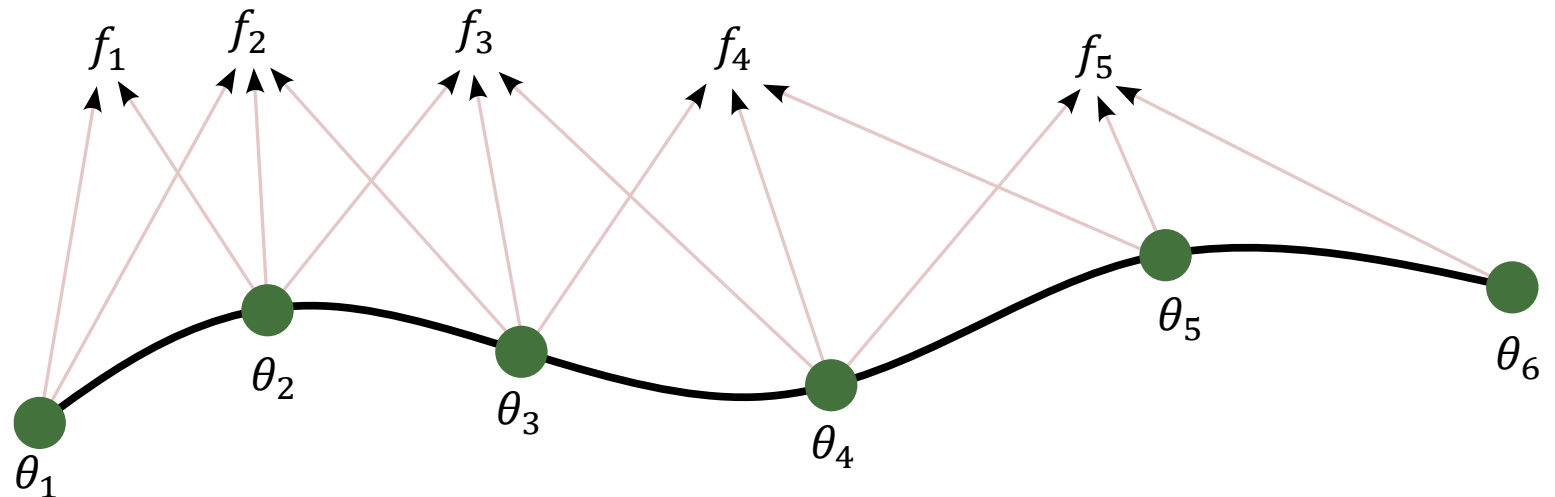
$$F(\theta) = \begin{bmatrix} f_1 = 2(\theta_2^3 - \theta_1^2) \\ f_2 = 3(\theta_2^3 - \theta_1^2) + 2(\theta_3^3 - \theta_2^2) \\ f_3 = 3(\theta_3^3 - \theta_2^2) + 2(\theta_4^3 - \theta_3^2) \\ \vdots \\ f_m = 3(\theta_n^3 - \theta_{n-1}^2) \end{bmatrix}$$



$$J_F = \begin{bmatrix} \times & \times & & & & \\ \times & \times & \times & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \times & \times & \times \\ & & & & \times & \times \end{bmatrix}$$

Each component of F depends on just at most three components of θ

Each row of the Jacobian contains only at most three nonzero elements



Jacobian-vector Product (JVP)

JVP is the product of a Jacobian matrix $J \in \mathbb{R}^{m \times n}$ and a vector $v \in \mathbb{R}^n$

$$Jv \in \mathbb{R}^m \quad \Rightarrow \quad \begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \dots & \frac{\partial f_1}{\partial \theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial \theta_1} & \dots & \frac{\partial f_m}{\partial \theta_n} \end{bmatrix} \begin{bmatrix} v_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} v_1 \frac{\partial f_1}{\partial \theta_1} \\ \vdots \\ \vdots \\ v_1 \frac{\partial f_m}{\partial \theta_1} \end{bmatrix}$$

JVP tells us how the output F changes when the input θ changes by amount/move in direction v

Jacobian-vector Product (JVP)

JVP is the product of a Jacobian matrix $J \in \mathbb{R}^{m \times n}$ and a vector $v \in \mathbb{R}^n$

$$Jv \in \mathbb{R}^m \quad \Rightarrow \quad \begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \dots & \frac{\partial f_1}{\partial \theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial \theta_1} & \dots & \frac{\partial f_m}{\partial \theta_n} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} v_1 \frac{\partial f_1}{\partial \theta_1} + v_2 \frac{\partial f_1}{\partial \theta_2} + \dots v_n \frac{\partial f_1}{\partial \theta_n} \\ \vdots \\ v_1 \frac{\partial f_m}{\partial \theta_1} + v_2 \frac{\partial f_m}{\partial \theta_2} + \dots v_n \frac{\partial f_m}{\partial \theta_n} \end{bmatrix}$$

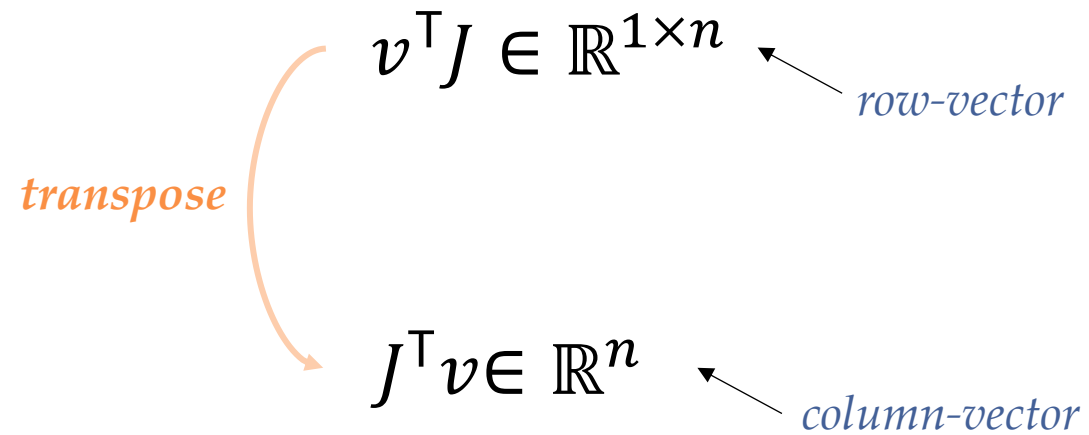
Directional derivative of F along v

JVP tells us how the output F changes when the input θ changes by amount/move in direction v

JVP propagates perturbations forward.

Vector-Jacobian Product (VJP)

VJP is the product of a vector $v \in \mathbb{R}^{m \times 1}$ and the Jacobian matrix $J \in \mathbb{R}^{m \times n}$



What does VJP mean?

Consider scalar function (ℓ) defined through a vector-valued intermediate function:

$$\ell = K(F(\theta))$$

$$F: \mathbb{R}^n \rightarrow \mathbb{R}^m, \theta \in \mathbb{R}^n, F(\theta) \in \mathbb{R}^m, K: \mathbb{R}^m \rightarrow \mathbb{R}$$

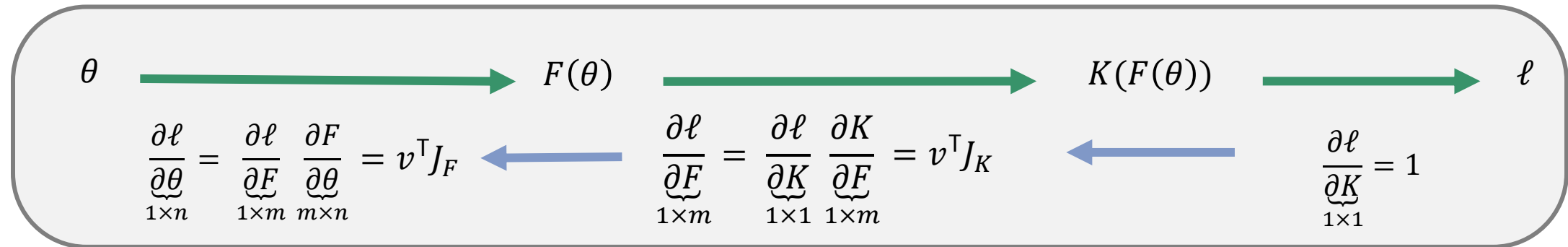


What does VJP mean?

Consider scalar function (ℓ) defined through a vector-valued intermediate function:

$$\ell = K(F(\theta))$$

$$F: \mathbb{R}^n \rightarrow \mathbb{R}^m, \theta \in \mathbb{R}^n, F(\theta) \in \mathbb{R}^m, K: \mathbb{R}^m \rightarrow \mathbb{R}$$

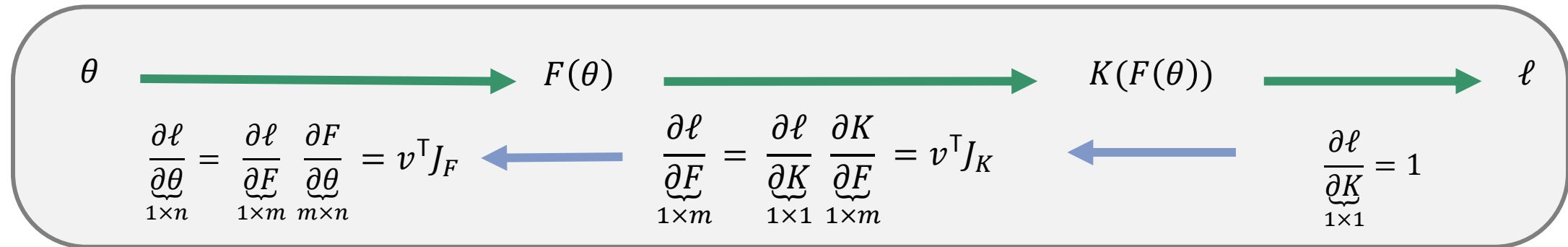


What does VJP mean?

Consider scalar function (ℓ) defined through a vector-valued intermediate function:

$$\ell = K(F(\theta))$$

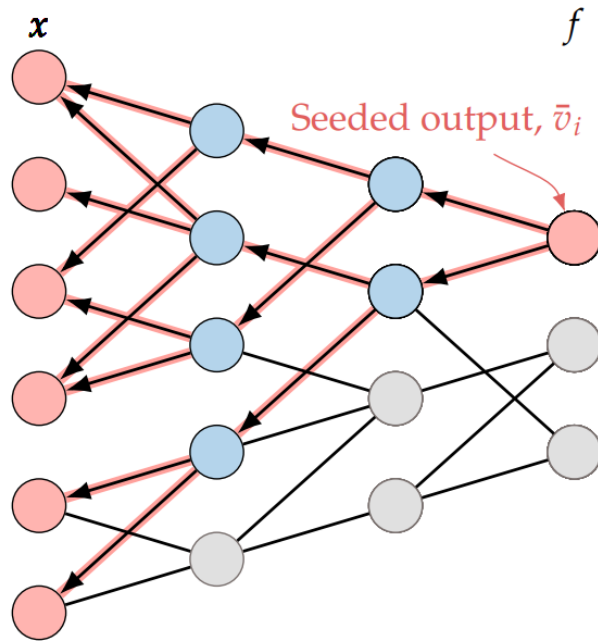
$$F: \mathbb{R}^n \rightarrow \mathbb{R}^m, \theta \in \mathbb{R}^n, F(\theta) \in \mathbb{R}^m, K: \mathbb{R}^m \rightarrow \mathbb{R}$$



VJP propagates sensitivities backward

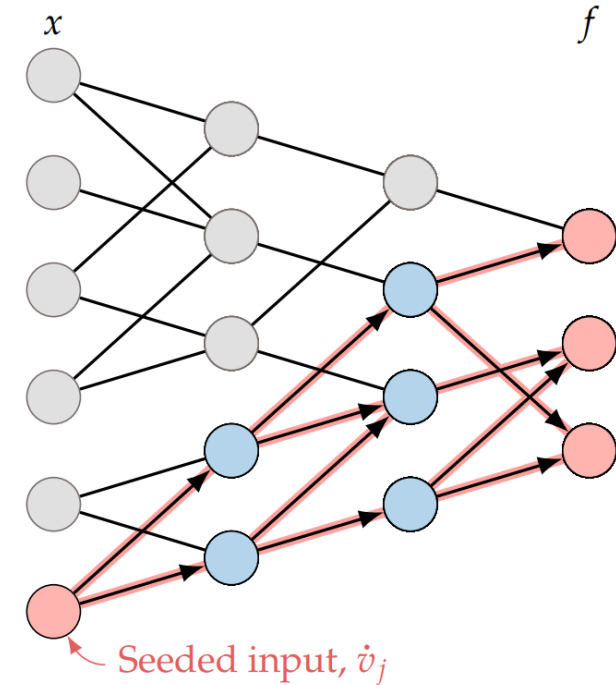
Automatic differentiation: reverse *vs.* forward mode

Reverse Mode (VJP)



Cost $\propto$ #outputs

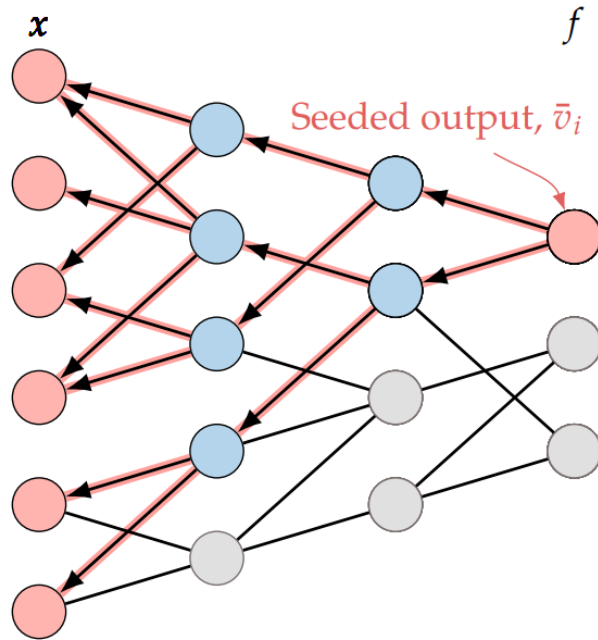
Forward Mode (JVP)



Cost $\propto$ #inputs

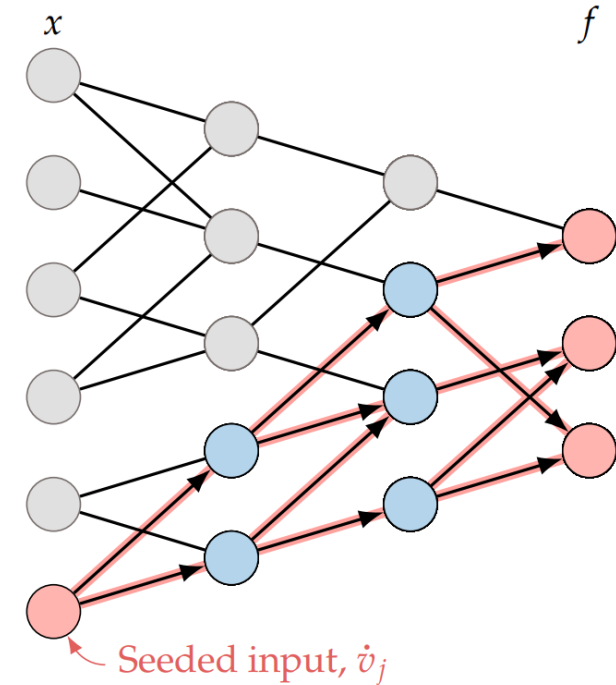
Automatic differentiation: reverse *vs.* forward mode

Reverse Mode (VJP)



Cost $\propto$ #outputs

Forward Mode (JVP)



Cost $\propto$ #inputs

Common case: many params, one loss $\rightarrow$ reverse wins

PyTorch Scalar Example

$$f(\theta) = (\theta_1 \theta_2 \sin \theta_3 + e^{\theta_1 \theta_2}) / \theta_3$$

```
import torch
theta_1 = torch.tensor(2.0, requires_grad=True)
theta_2 = torch.tensor(3.0, requires_grad=True)
theta_3 = torch.tensor(4.0, requires_grad=True)
f = (theta_1 * theta_2 * torch.sin(theta_3) + torch.exp(theta_1 * theta_2)) / theta_3

grads = torch.autograd.grad(f, [theta_1, theta_2, theta_3])

print(grads[0]) #  $\frac{\partial f}{\partial \theta_1}$ 
print(grads[1]) #  $\frac{\partial f}{\partial \theta_2}$ 
print(grads[2]) #  $\frac{\partial f}{\partial \theta_3}$ 
```

PyTorch Scalar Example with .backward()

$$f(\theta) = (\theta_1 \theta_2 \sin \theta_3 + e^{\theta_1 \theta_2}) / \theta_3$$

```
import torch
theta_1 = torch.tensor(2.0, requires_grad=True)
theta_2 = torch.tensor(3.0, requires_grad=True)
theta_3 = torch.tensor(4.0, requires_grad=True)
f = (theta_1 * theta_2 * torch.sin(theta_3) +
     torch.exp(theta_1 * theta_2)) / theta_3
```

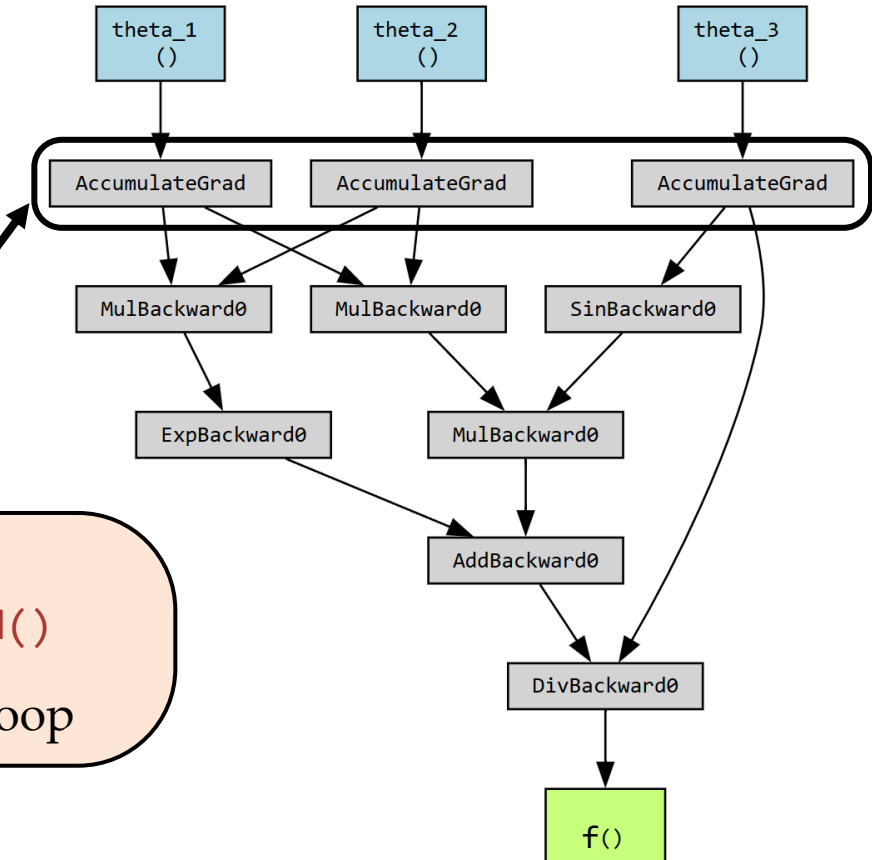
f.backward()

```
print(theta_1.grad) #  $\frac{\partial f}{\partial \theta_1}$ 
```

```
print(theta_2.grad) #  $\frac{\partial f}{\partial \theta_2}$ 
```

```
print(theta_3.grad) #  $\frac{\partial f}{\partial \theta_3}$ 
```

this is why we do
`optimizer.zero_grad()`
in the optimization loop



PyTorch Vector Function and Jacobian

$$F: \mathbb{R}^2 \rightarrow \mathbb{R}^2 \quad F(\theta) = \begin{bmatrix} x^2 + y \\ \sin(xy) \end{bmatrix} \quad \theta = \begin{bmatrix} x \\ y \end{bmatrix}$$

```
import torch

def F(theta):
    x, y = theta
    return torch.stack([x**2 + y,
                        torch.sin(x * y)])

theta = torch.tensor([2.0, 3.0],
                      requires_grad=True)

J = torch.autograd.functional.jacobian(F, theta)
print(J)
```

dense / materialized

JVP and VJP in PyTorch

Jacobian-vector product

```
from torch.func import jvp
theta = torch.tensor([2.0, 3.0])
v = torch.tensor([1.0, 0.5])

value, jvp_value = jvp(F, (theta,), (v,))
```

Vector-Jacobian product

```
theta = torch.tensor([2.0, 3.0], requires_grad=True)
w = torch.tensor([1.0, 2.0])
value = F(theta)
vjp_value, = torch.autograd.grad(value, theta, w)
```


- **Chapter 1:** Introduction to Automatic Differentiation
- **Chapter 2:** Differentiable Geometry Processing in Python
- **Chapter 3:** Introduction to The Adjoint Method
- **Chapter 4:** Inverse Geometry Processing in Python

Differentiable Geometry Processing In Python

Chapter 2


```
import torch
```

```
import torch  
from iskra.mesh import Mesh
```

```
import torch

from iskra.mesh import Mesh

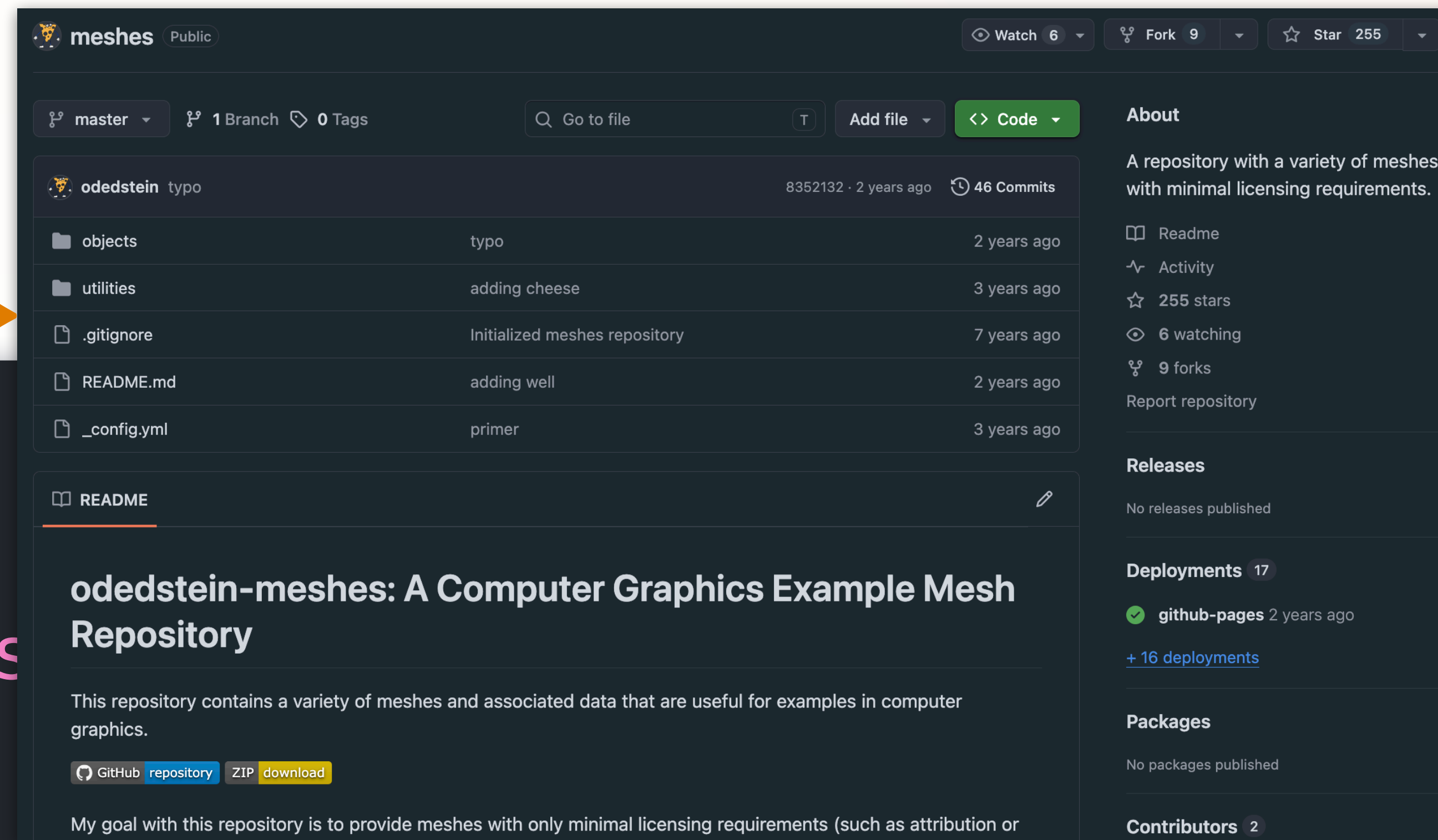
mesh, _ = Mesh.from_path(
    "oded://objects/koala/koala_low_resolution.obj",
    device="cpu"
)
```



```

import torch
from iskra.mesh import Mesh
mesh, _ = Mesh.from_path(
    "oded://objects/koala/koala_low_resolution.obj",
    device="cpu"
)

```



```
import torch

from iskra.mesh import Mesh

mesh, _ = Mesh.from_path(
    "oded://objects/koala/koala_low_resolution.obj",
    device="cpu"
)
```

```
import torch

from iskra.mesh import Mesh

mesh, _ = Mesh.from_path(
    "oded://objects/koala/koala_low_resolution.obj",
    device="cpu"
)

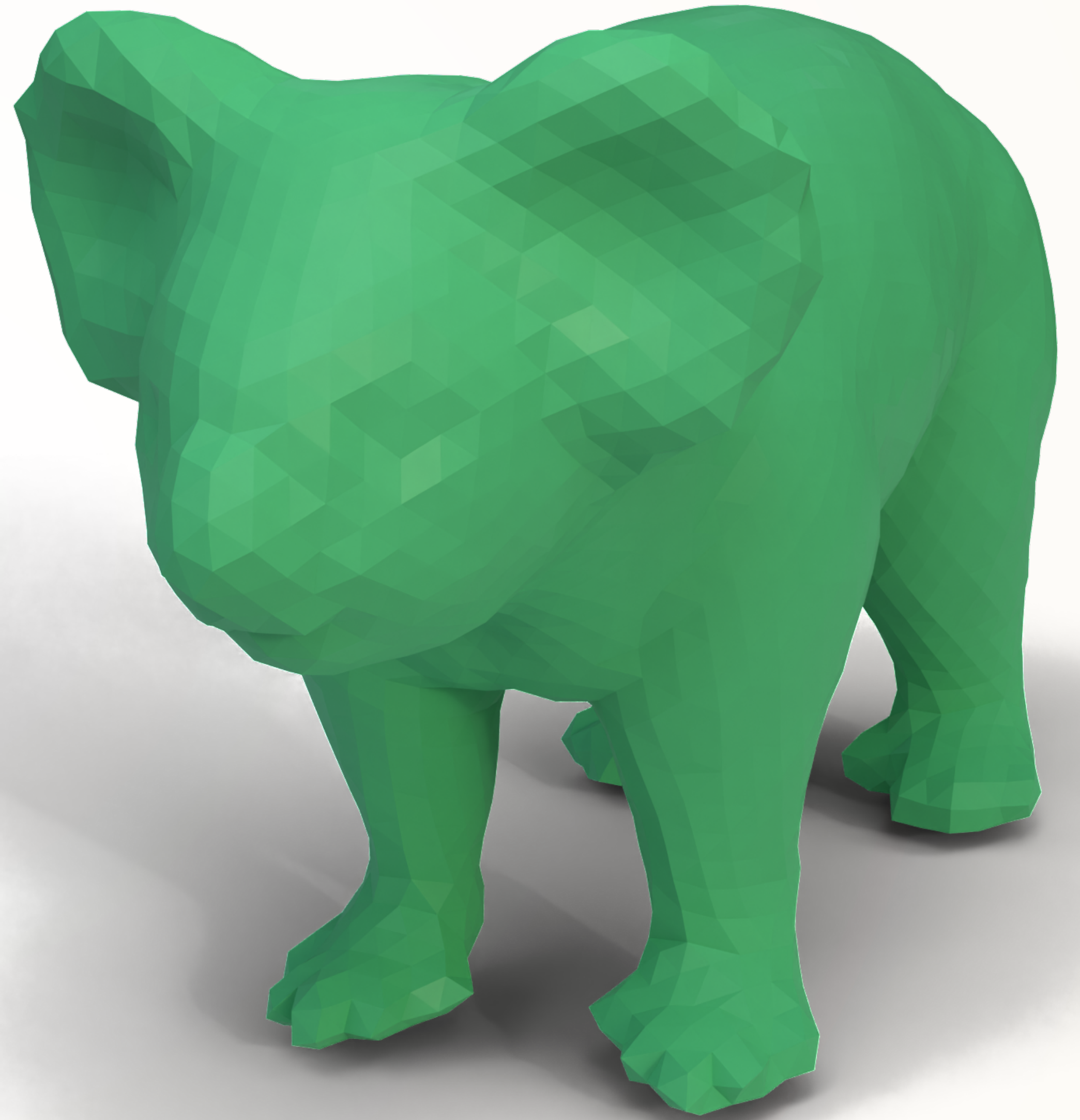
verts = mesh.vertices # [V, 3]
faces = mesh.faces # [F, 3]
```



```
import torch

from iskra.mesh import Mesh

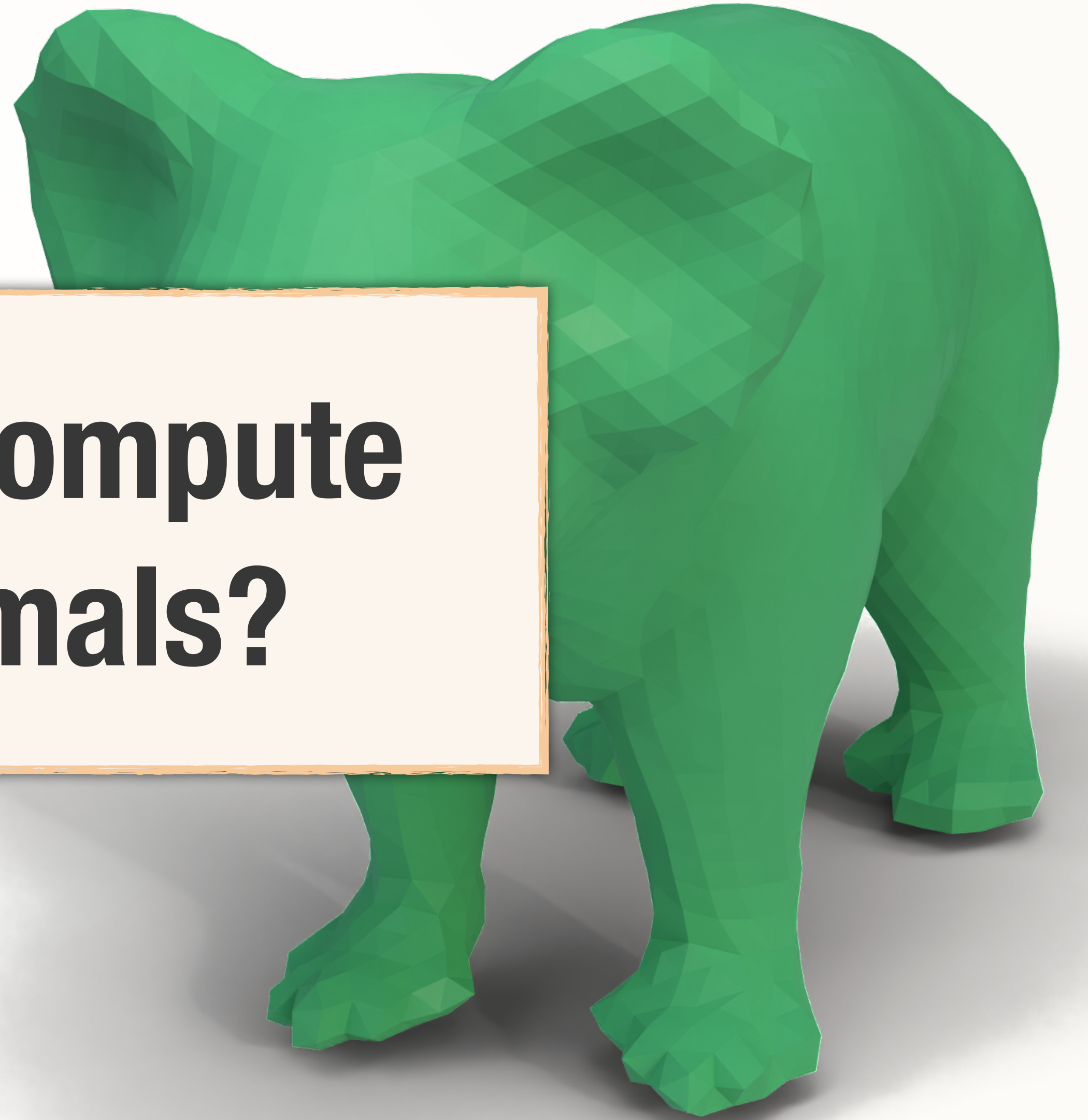
mesh, _ = Mesh.from_path(
    "oded://objects/koala/koala_low_resolution.obj",
    device="cpu"
)
verts = mesh.vertices # [V, 3]
faces = mesh.faces # [F, 3]
```

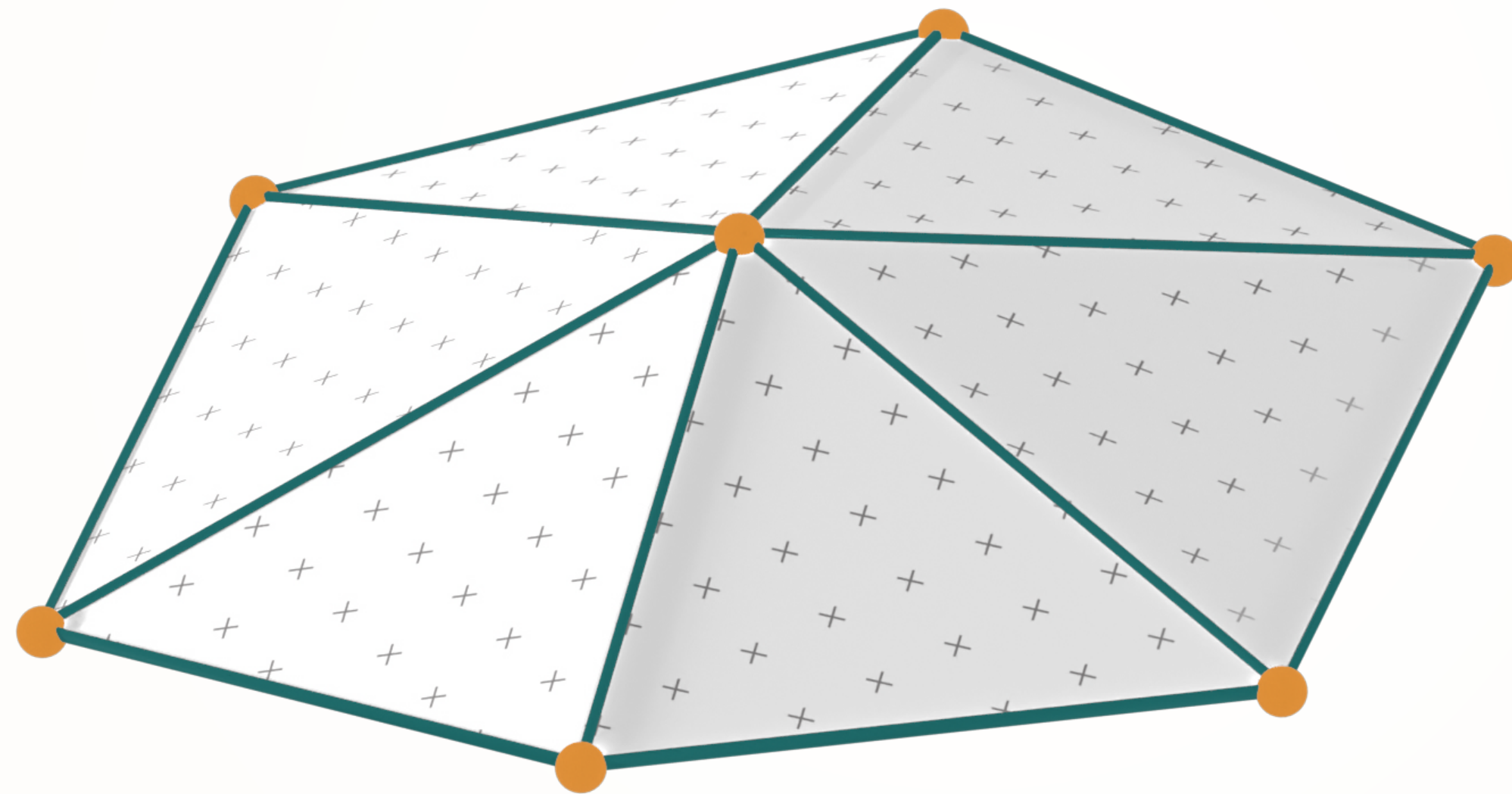



```
import torch

from iskra.mesh import
mesh, _ = Mesh.from_path(
    "oded://objects/koi",
    device="cpu"
)
verts = mesh.vertices
faces = mesh.faces #
```

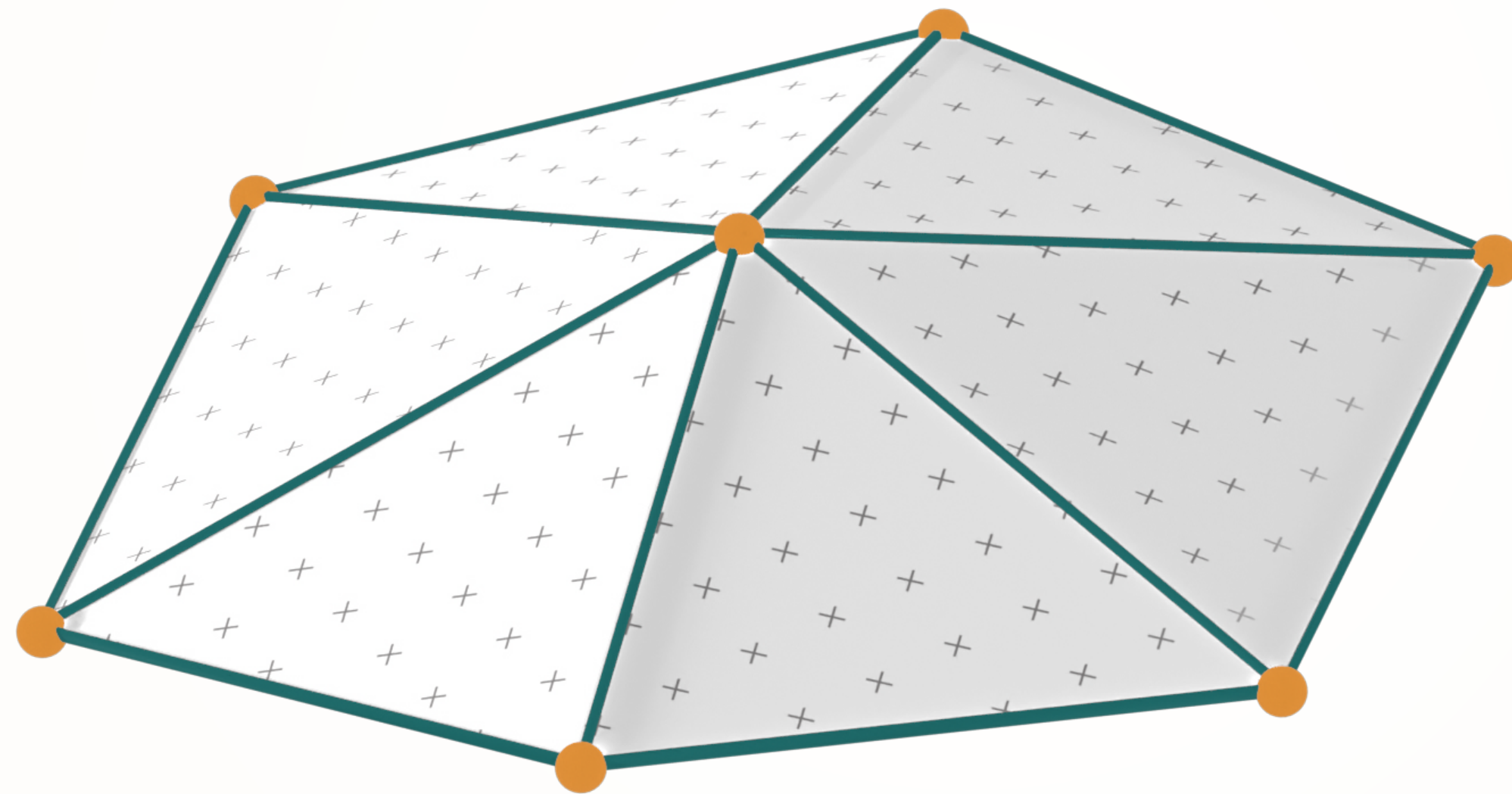
How do we compute vertex normals?





Scatter-Gather

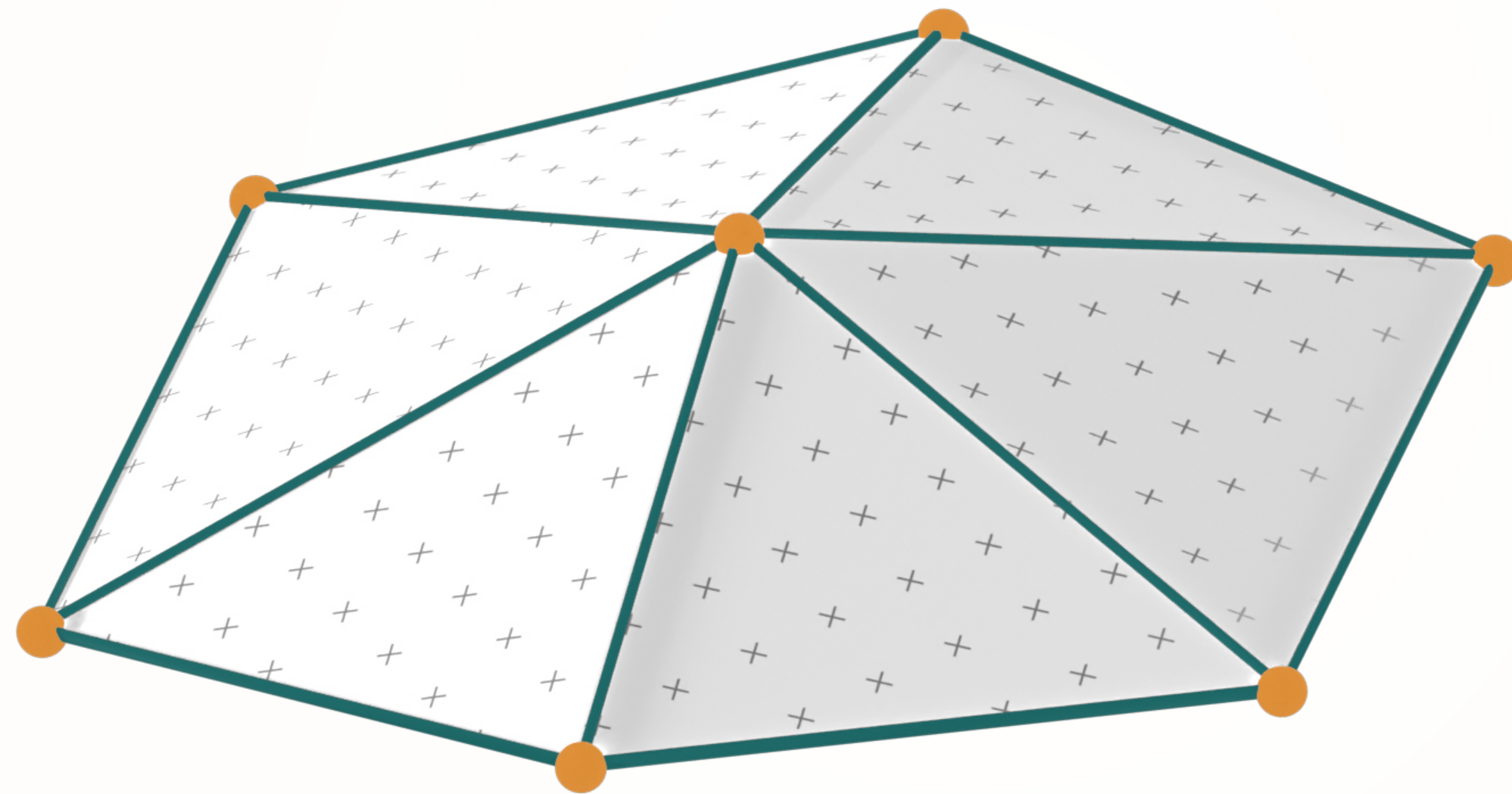
cf. Yu [2022], Dodik [2026]



Scatter-Gather

cf. Yu [2022], Dodik [2026]

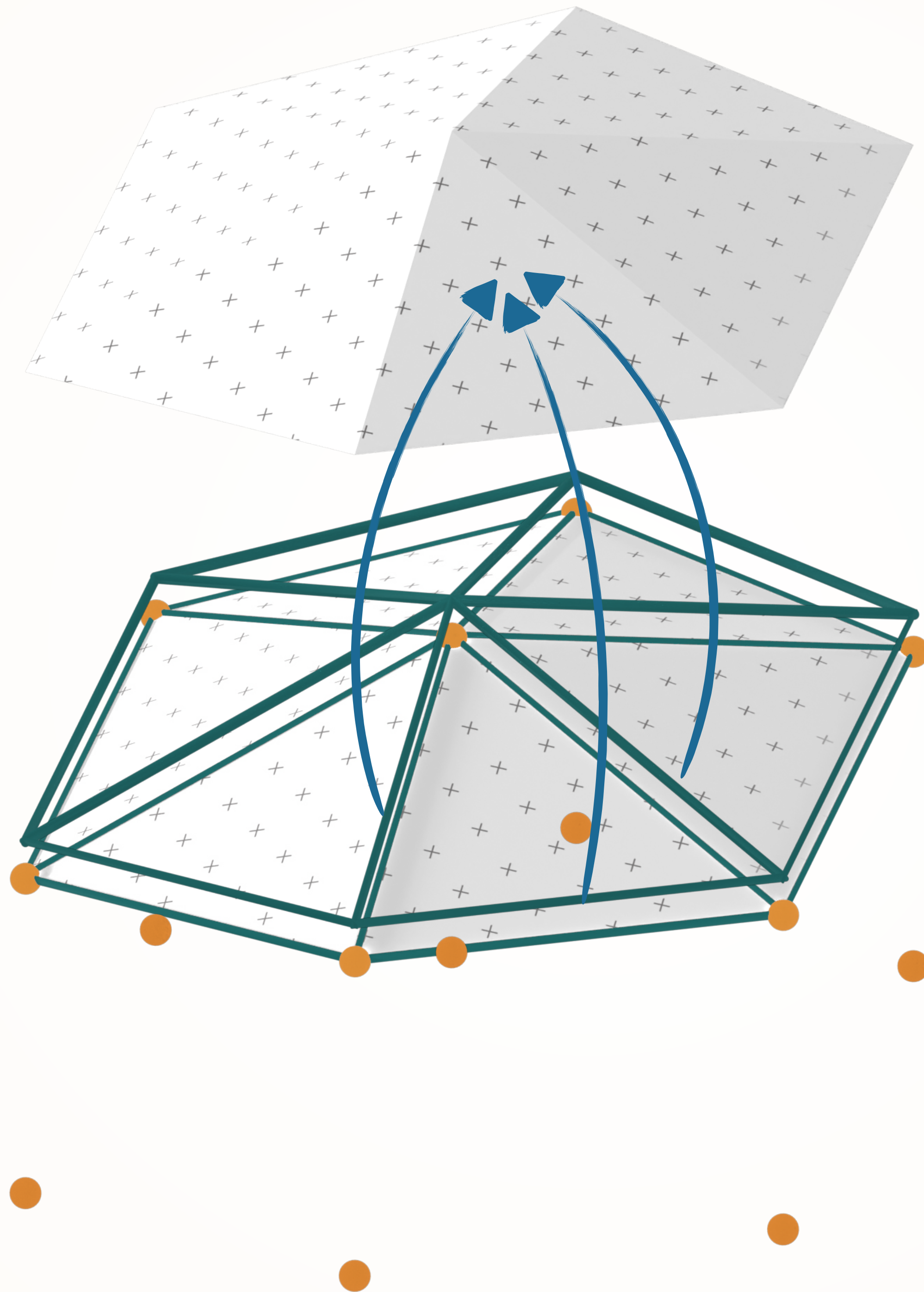
faces 0 faces 1 faces 2



Scatter-Gather

cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2

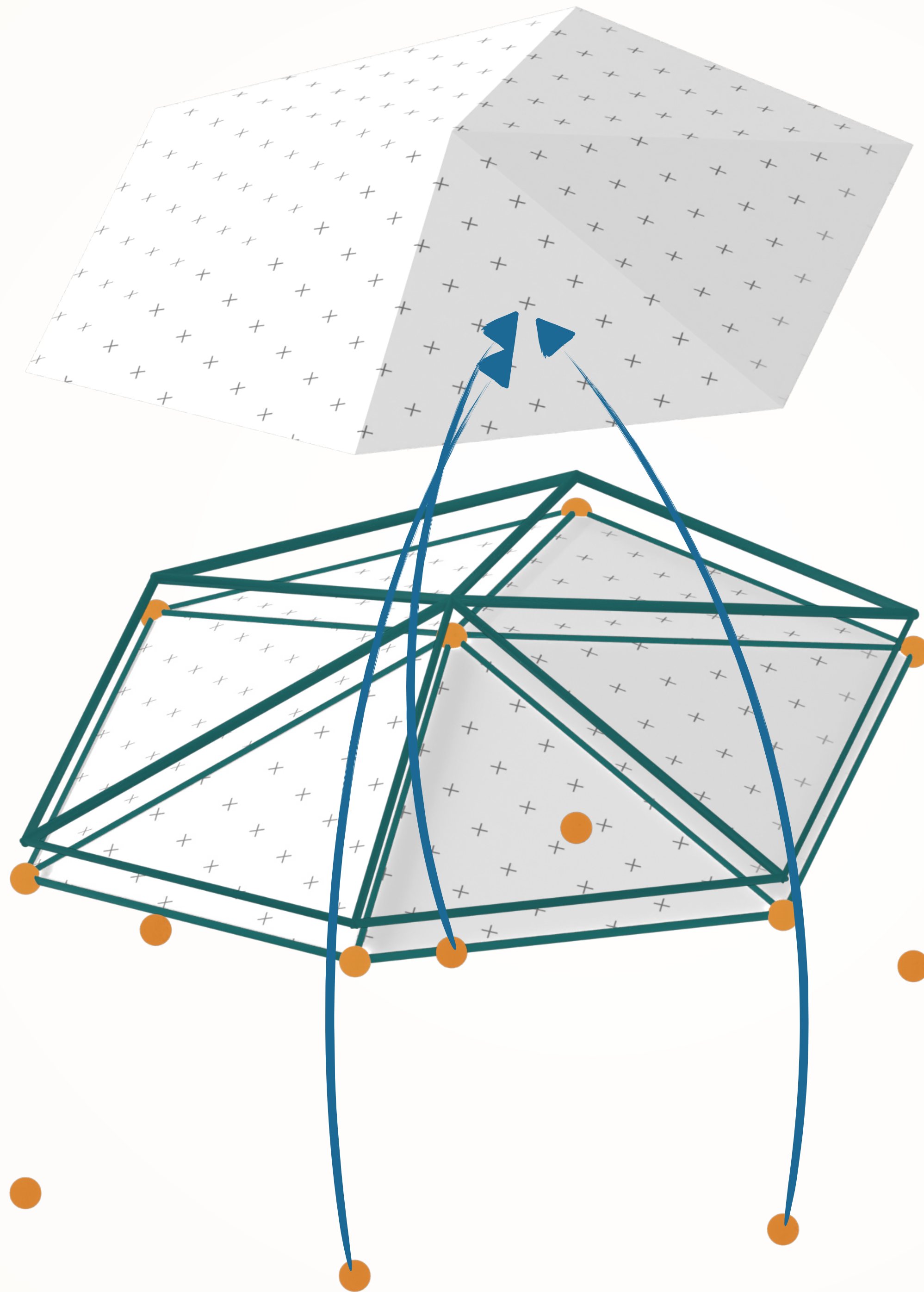


gather
1 on 2

Scatter-Gather

cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2

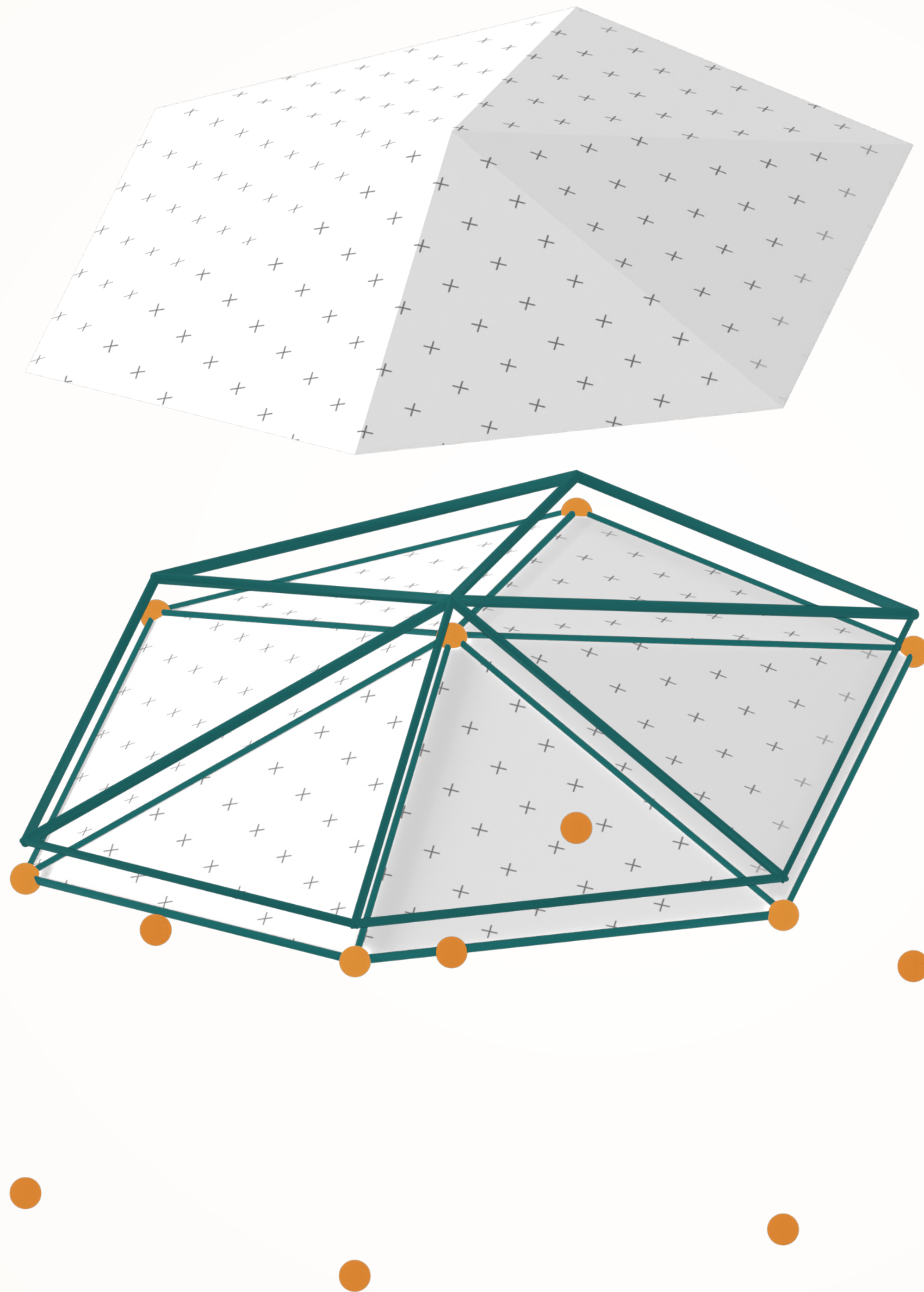


gather
0 on 2

Scatter-Gather

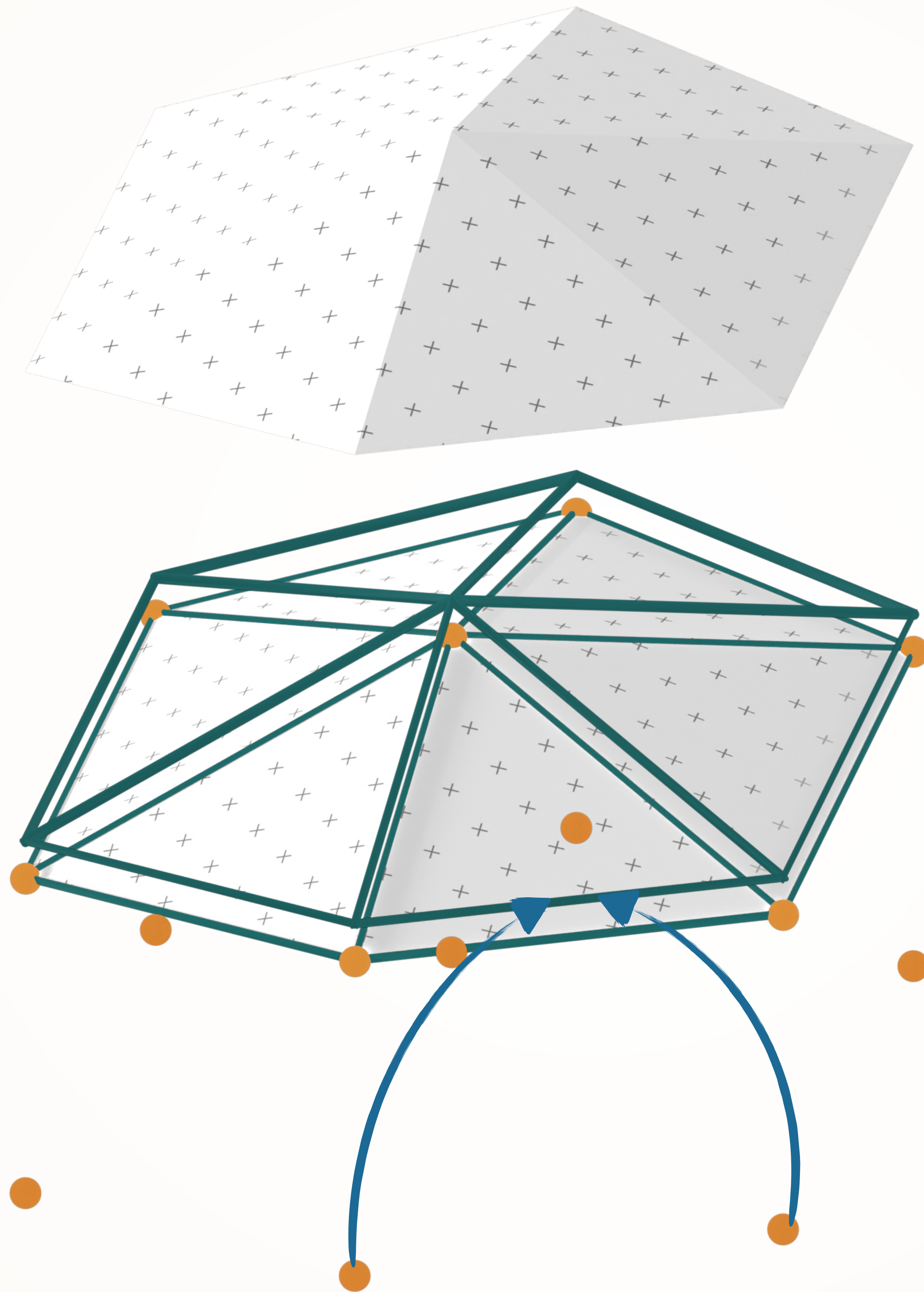
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2



Scatter-Gather
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2

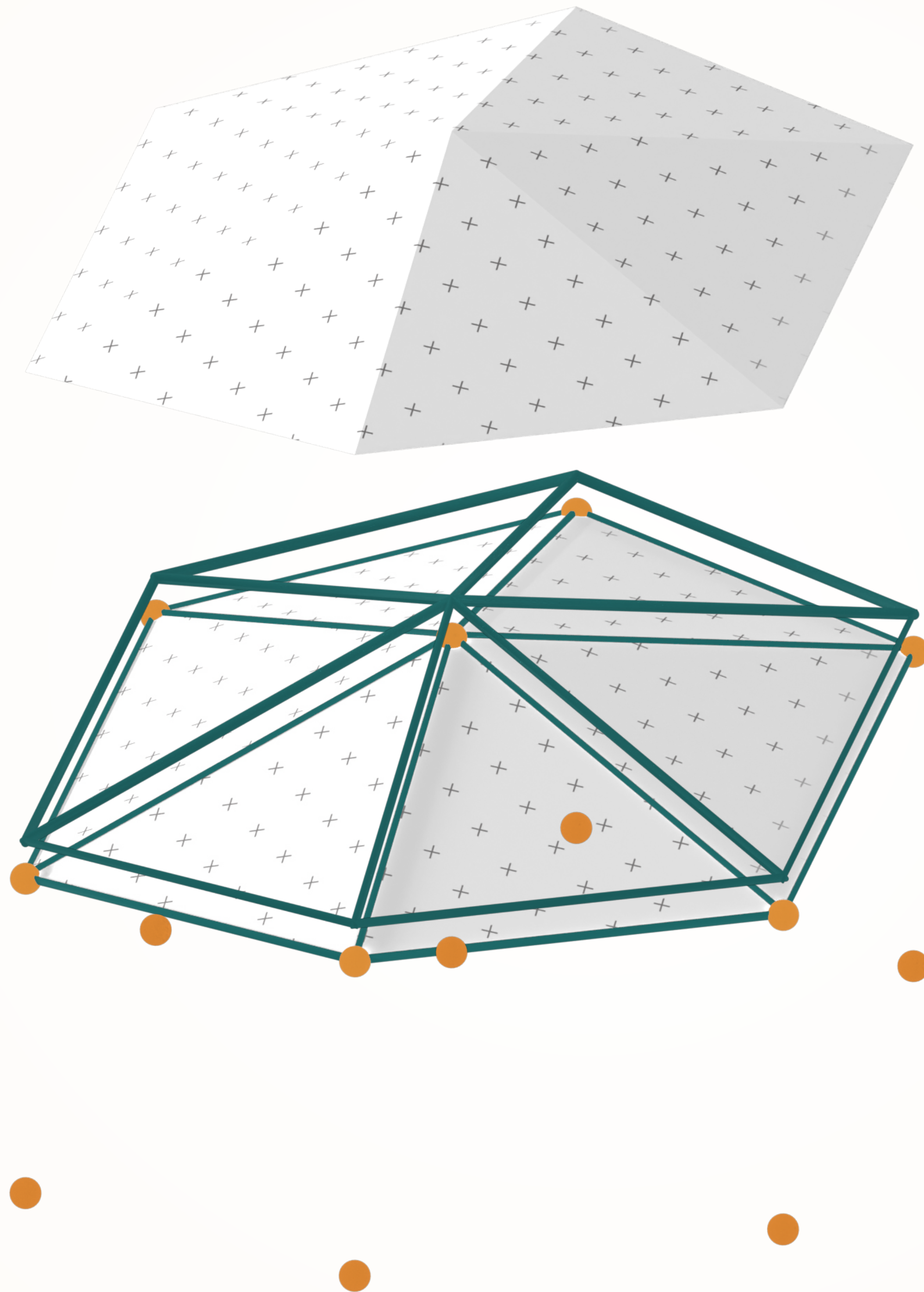


gather
0 on 1

Scatter-Gather

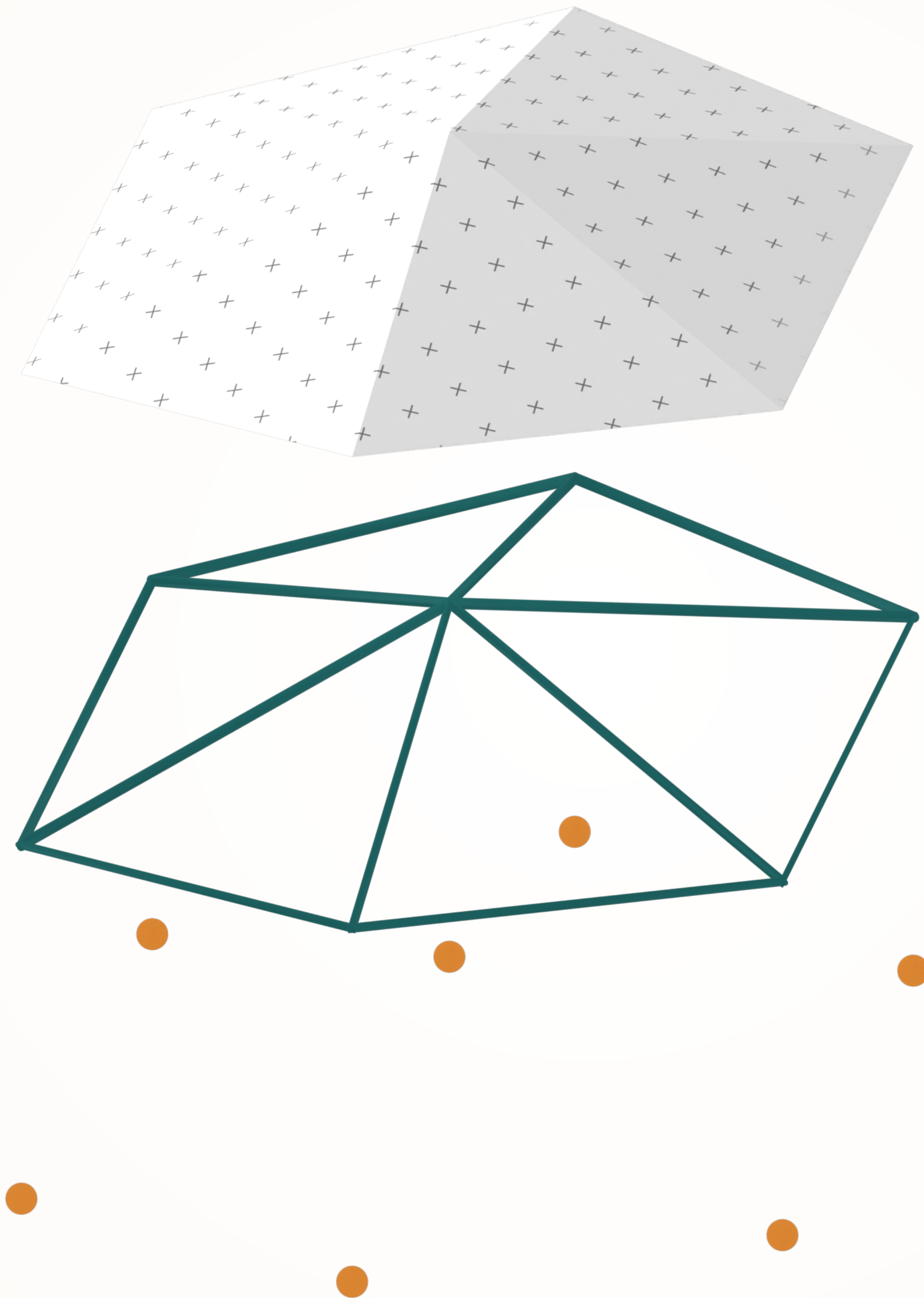
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2



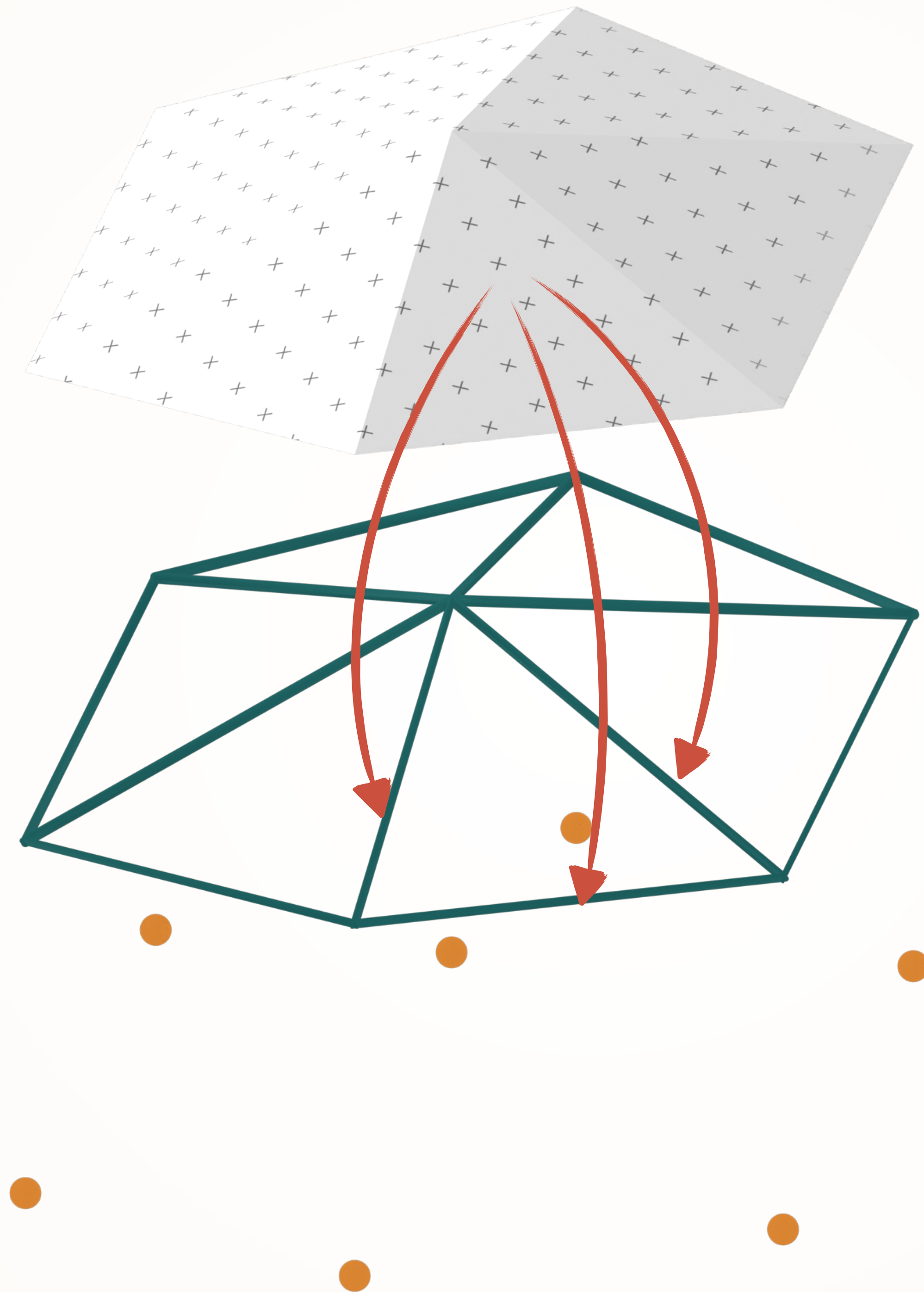
Scatter-Gather
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2



Scatter-Gather
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2

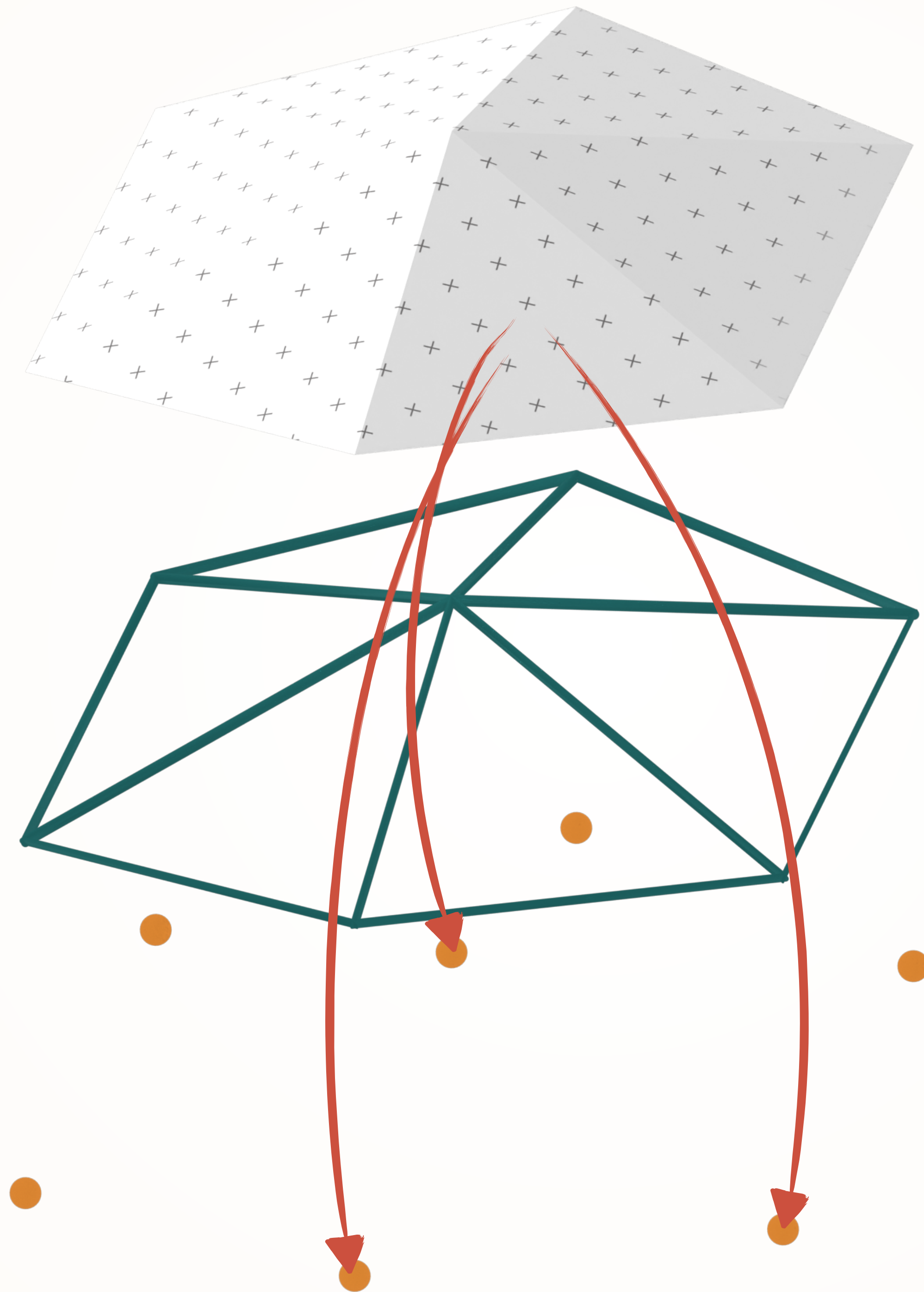


scatter
2 to 1

Scatter-Gather

cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2

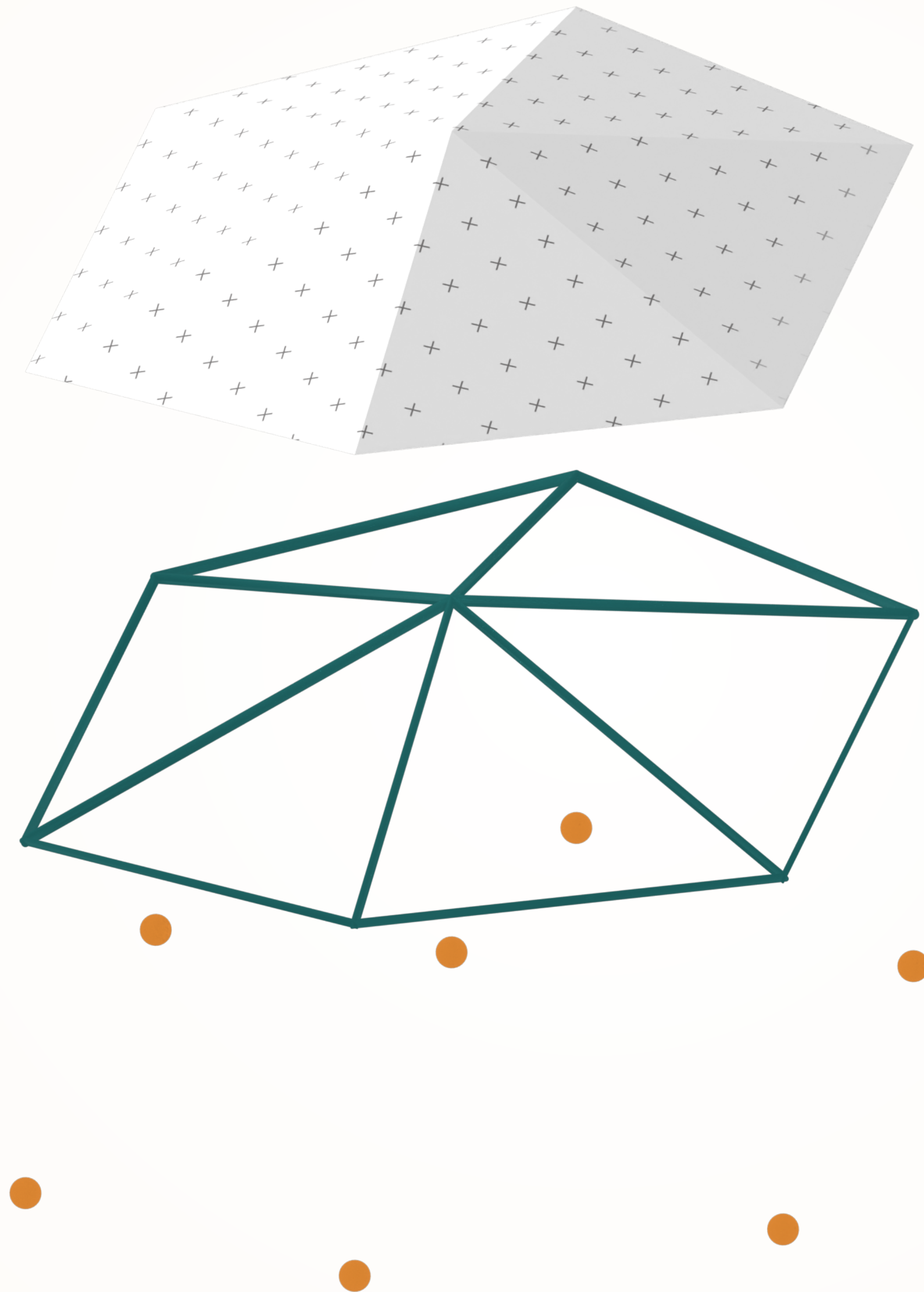


scatter
2 to 0

Scatter-Gather

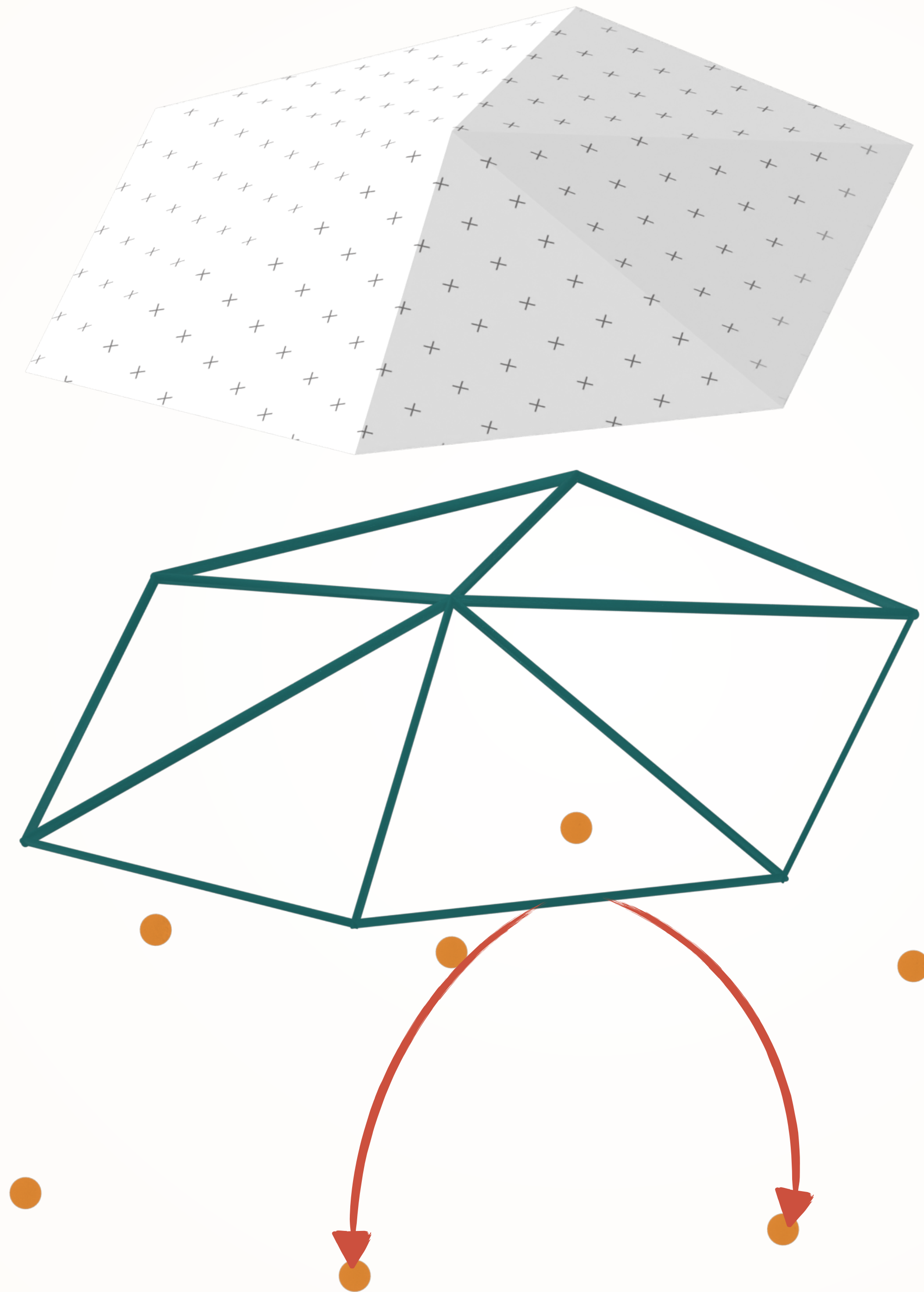
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2



Scatter-Gather
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2

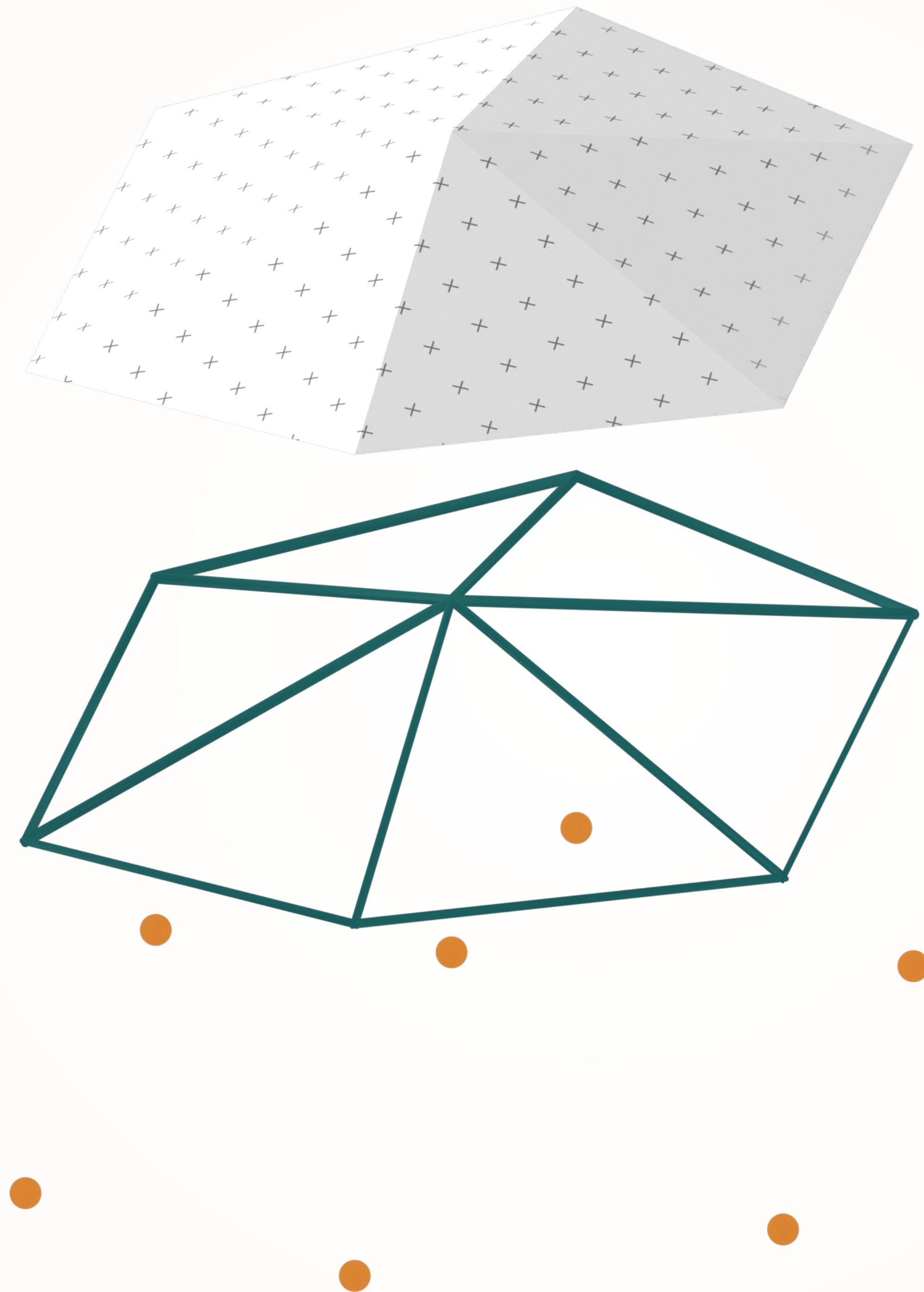


scatter
1 to 0

Scatter-Gather

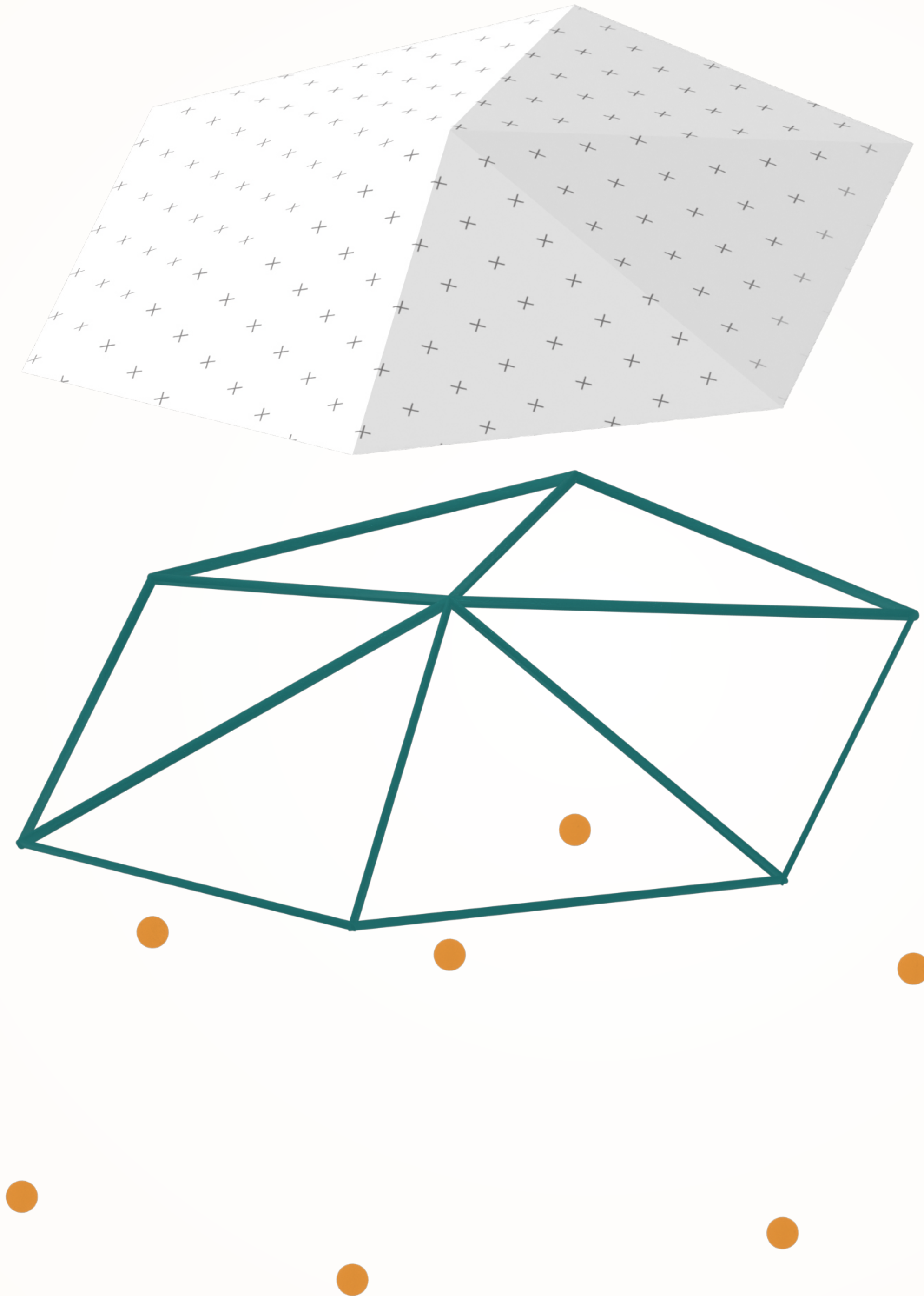
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2



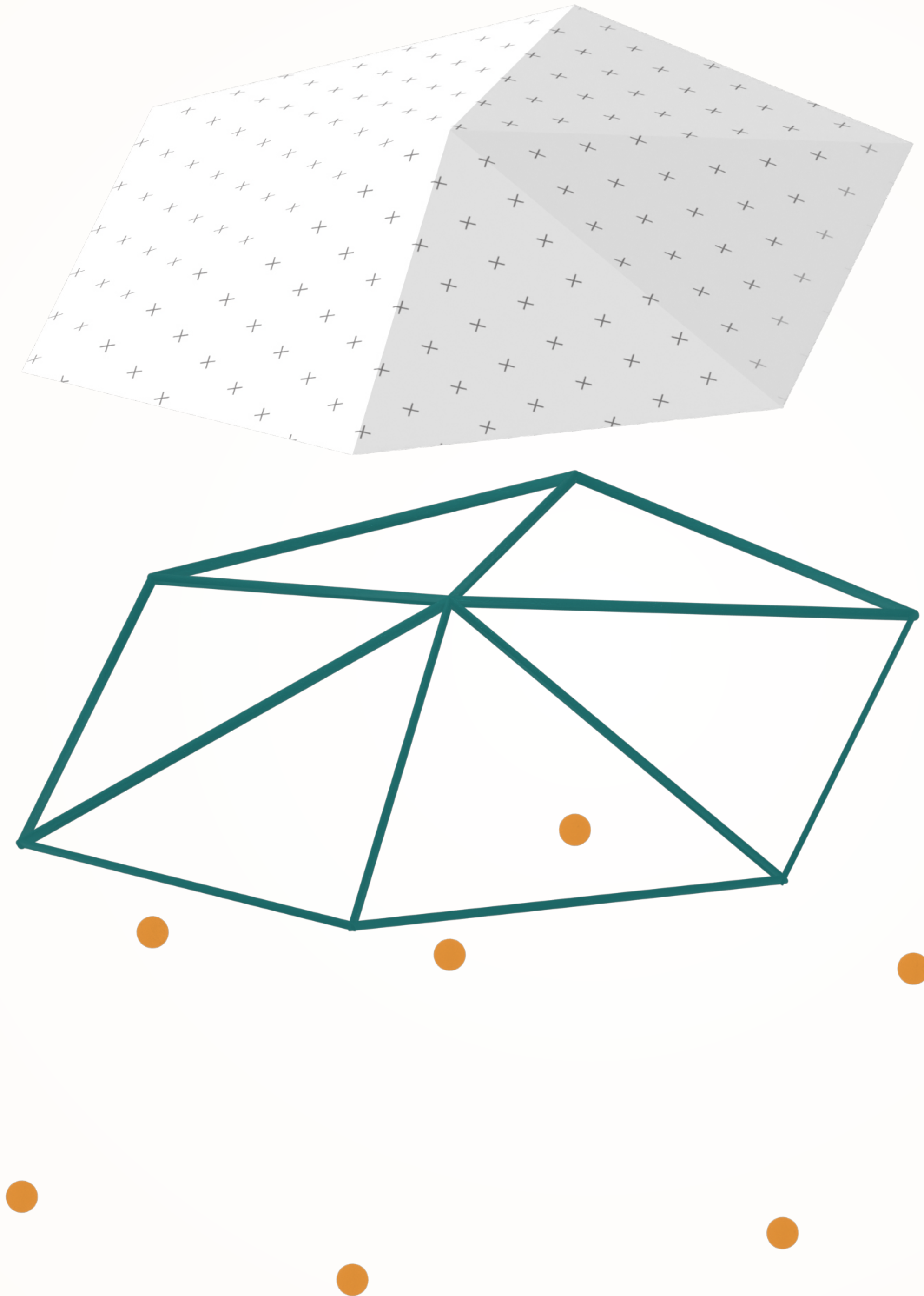
Scatter-Gather
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2



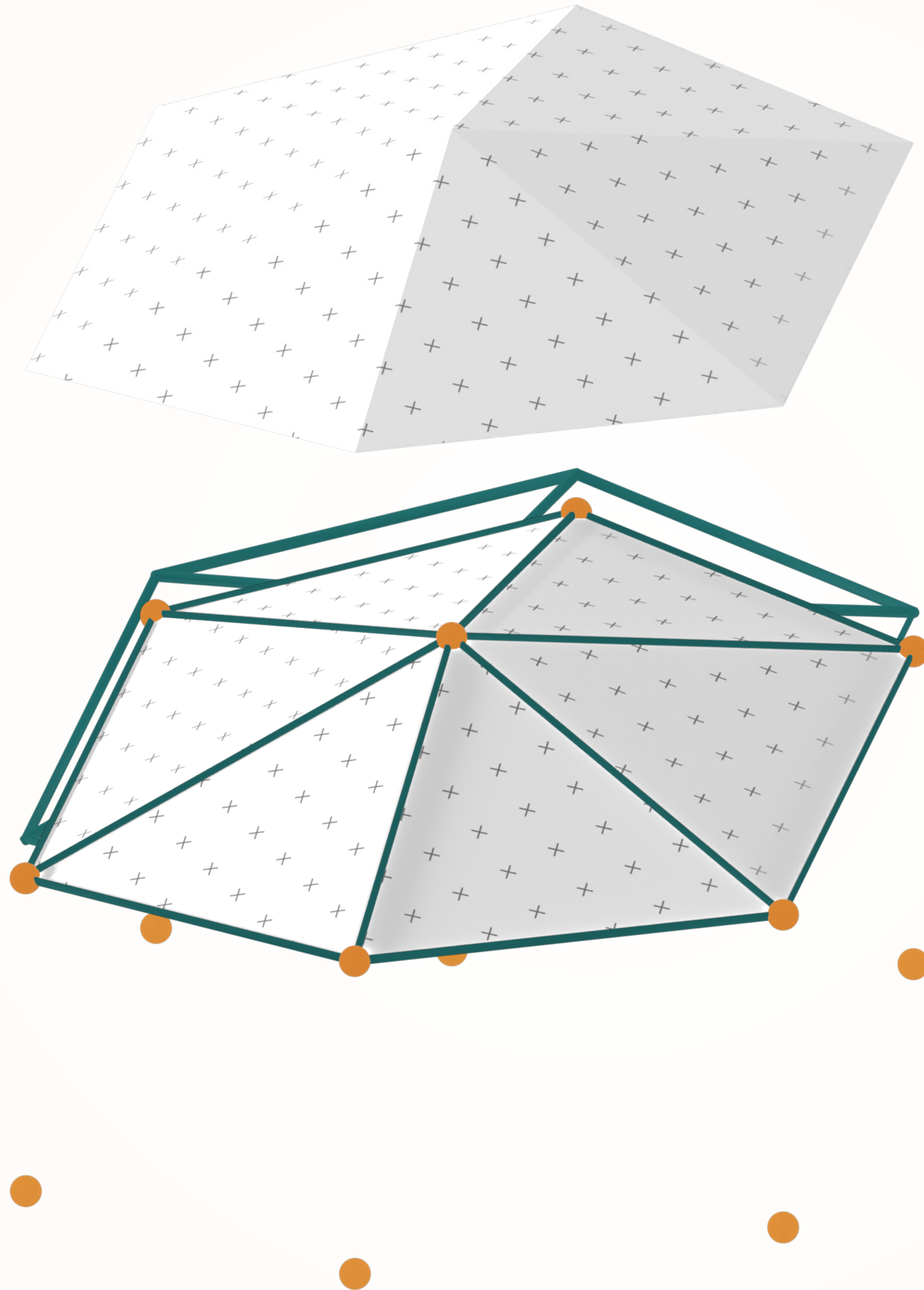
Scatter-Gather
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2

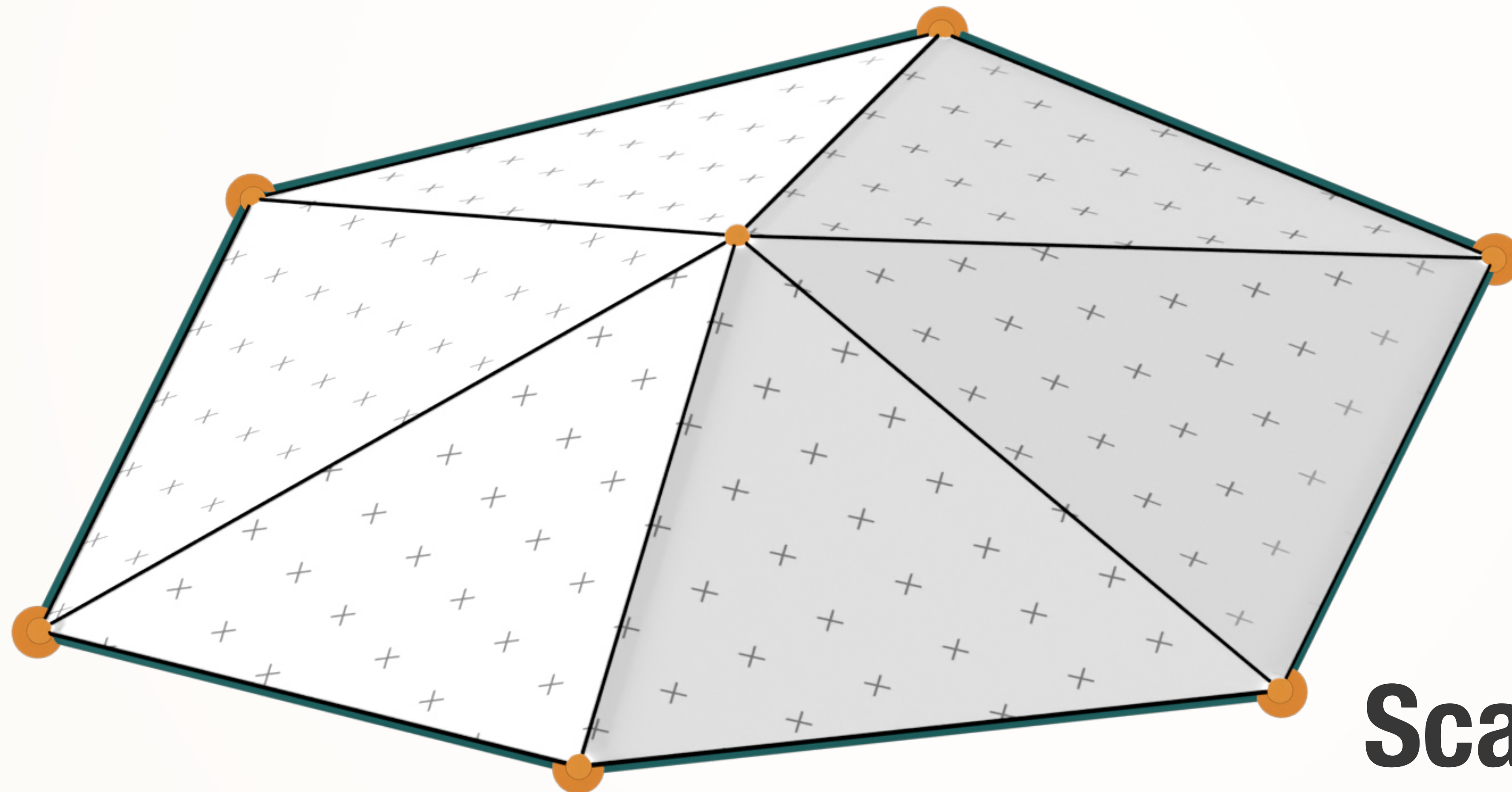


Scatter-Gather
cf. Yu [2022], Dodik [2026]

faces 0 faces 1 faces 2

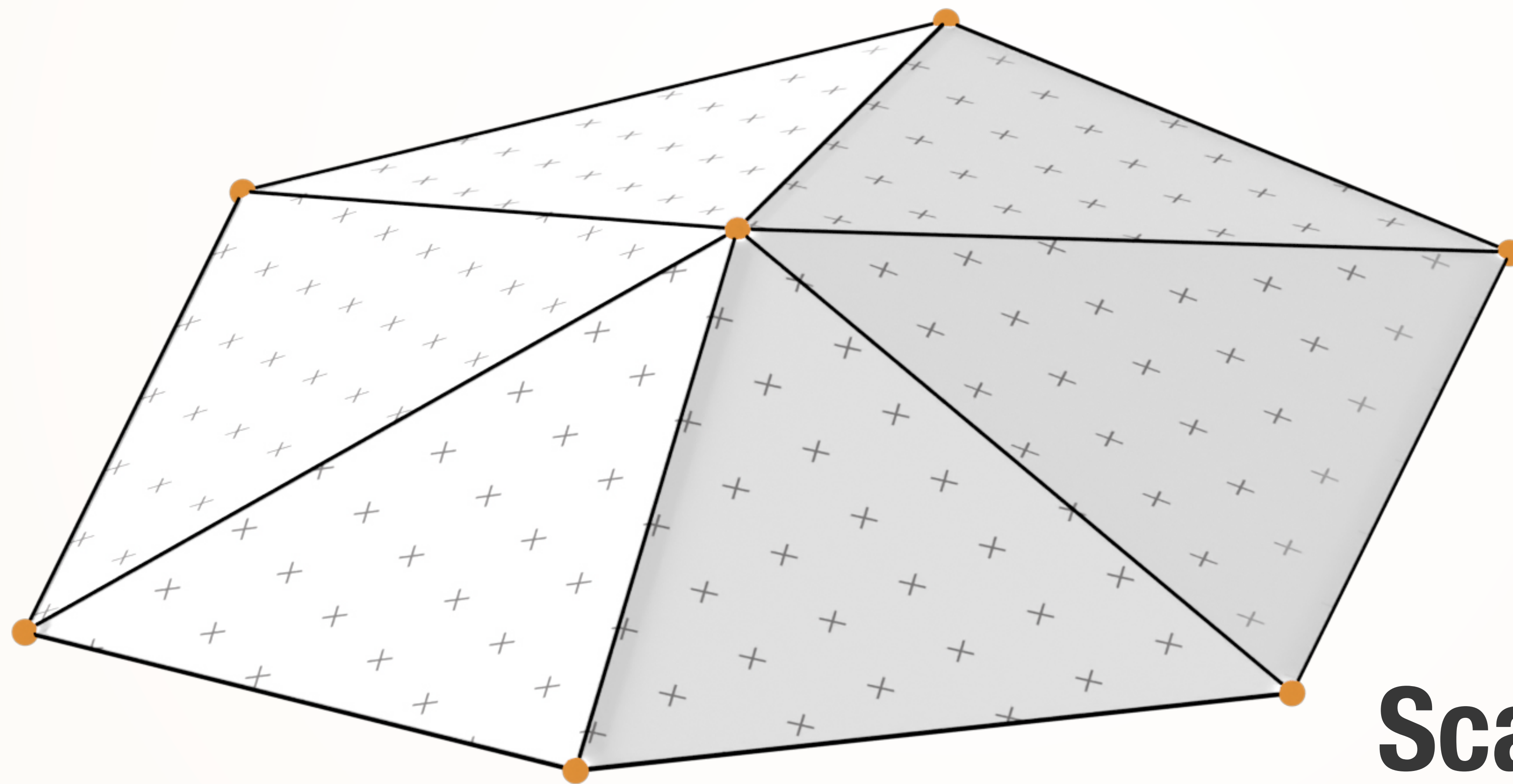


Scatter-Gather
cf. Yu [2022], Dodik [2026]



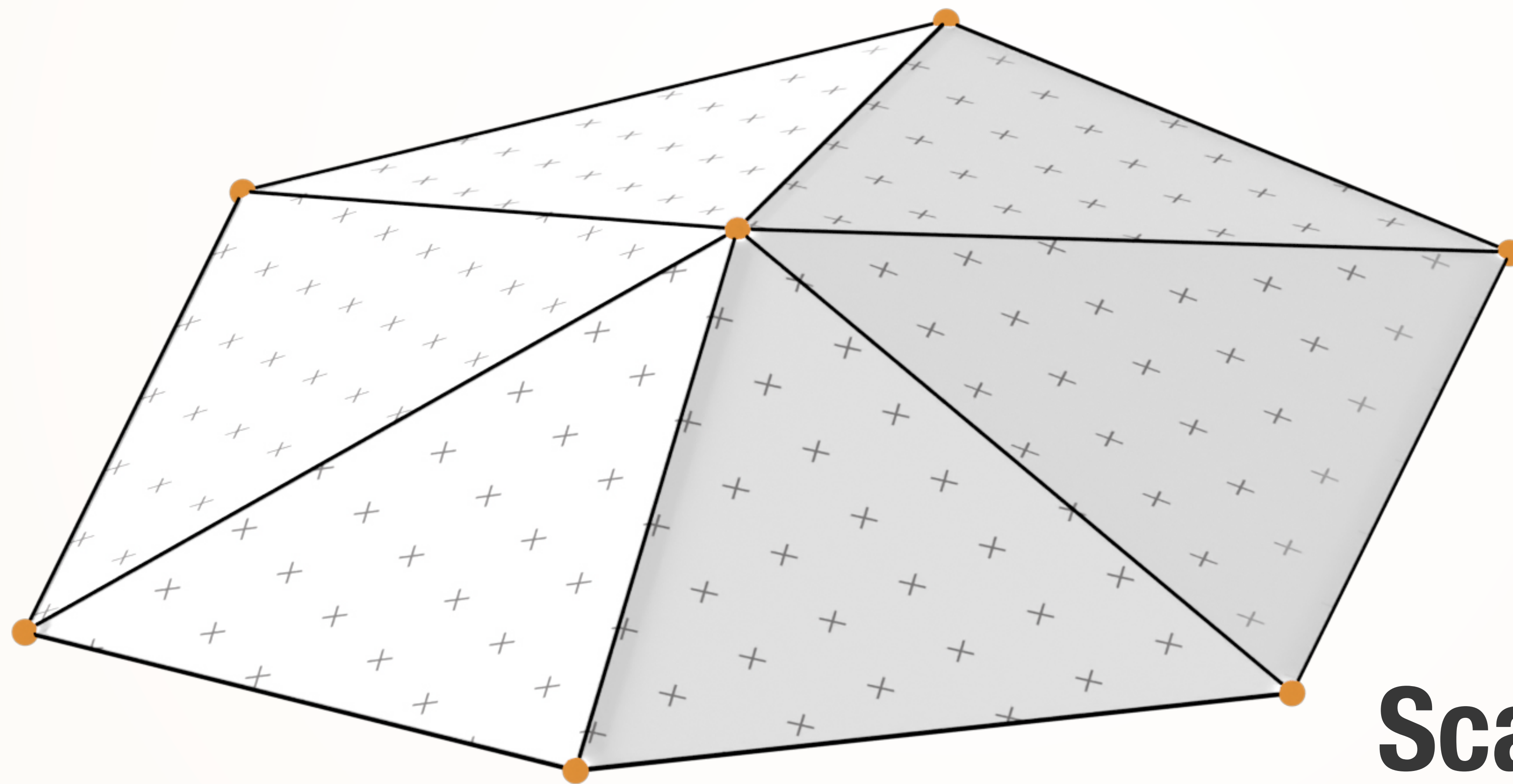
Scatter-Gather

cf. Yu [2022], Dodik [2026]



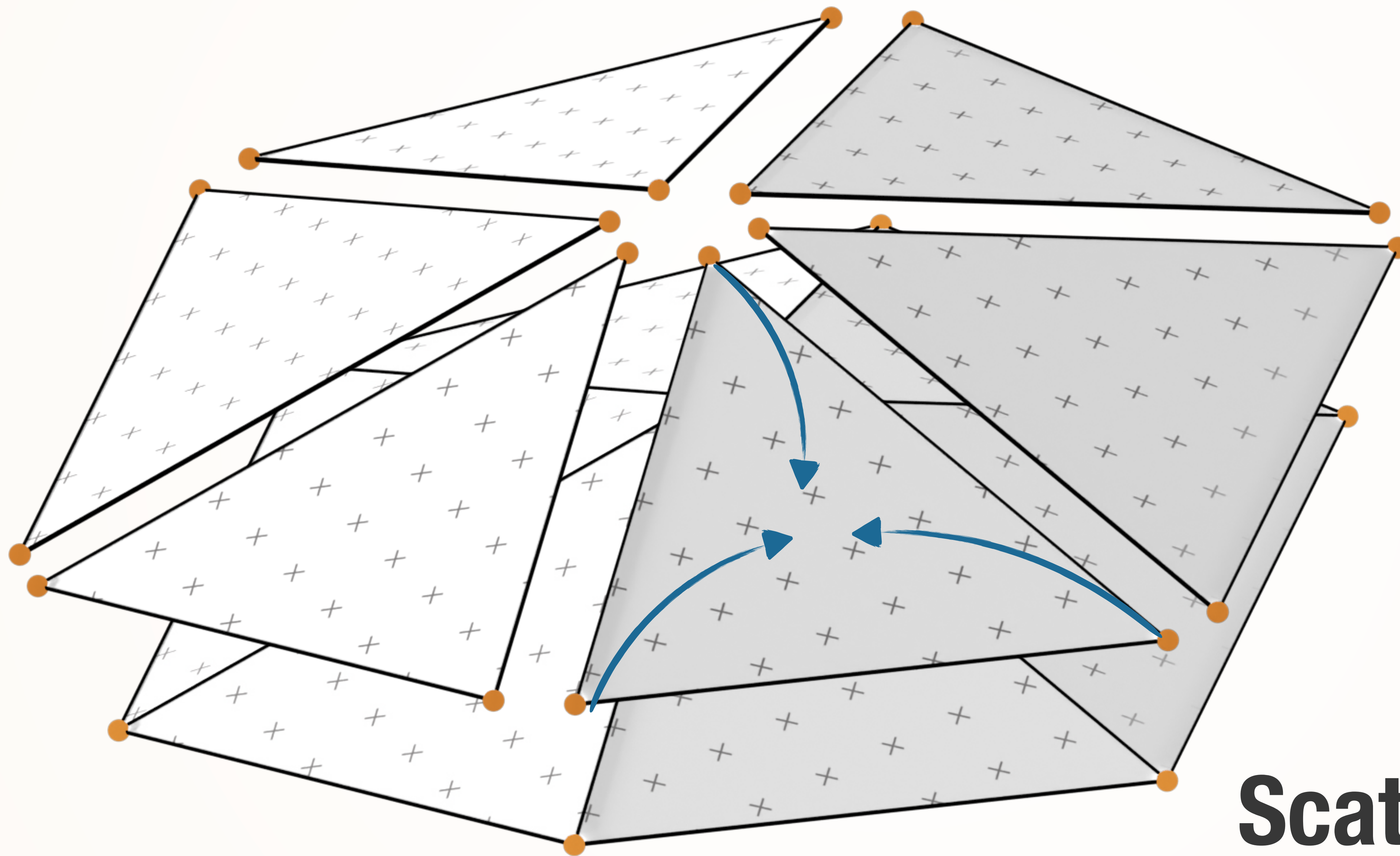
Scatter-Gather

cf. Yu [2022], Dodik [2026]



Scatter-Gather

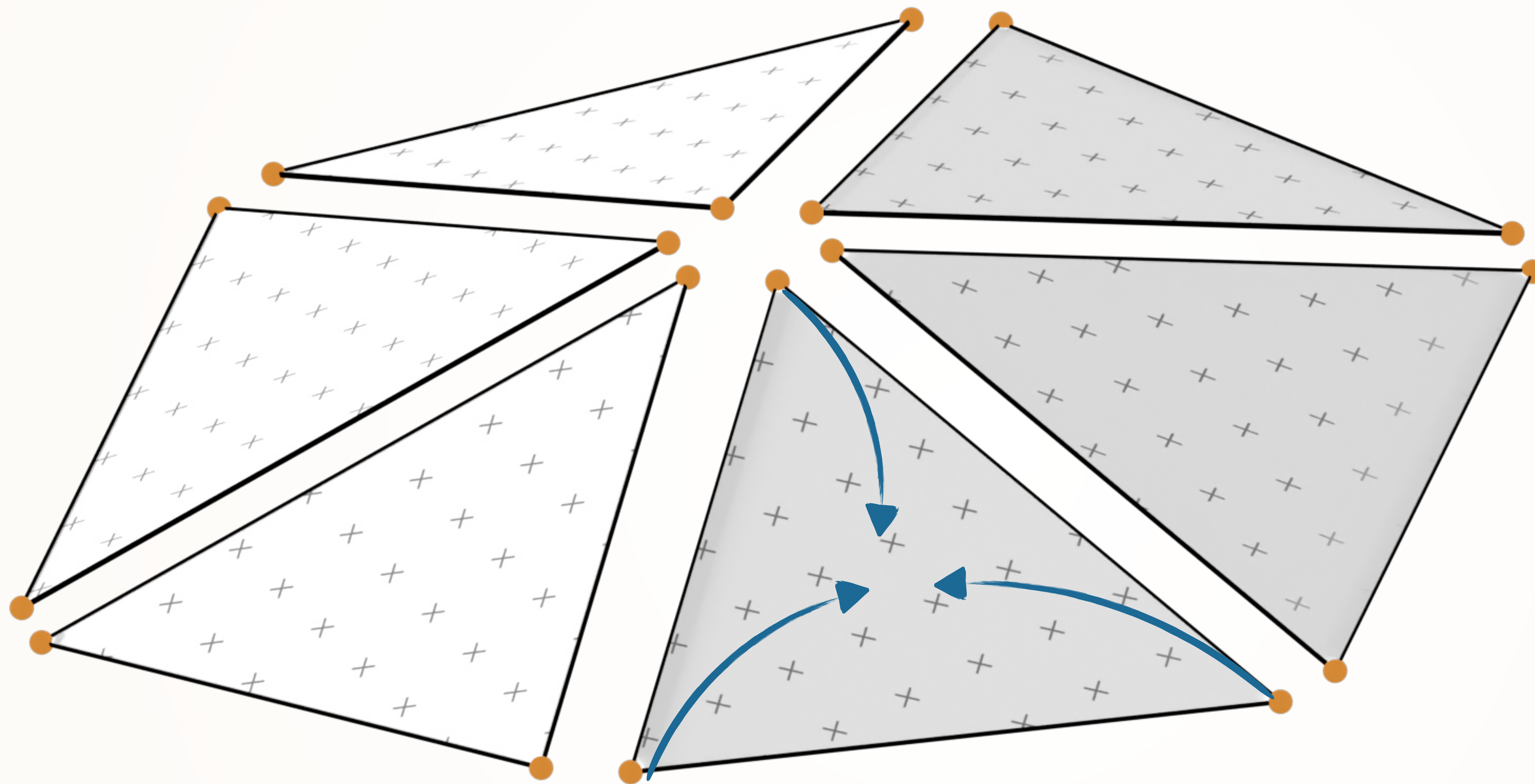
cf. Yu [2022], Dodik [2026]



gather
0 on 2

Scatter-Gather

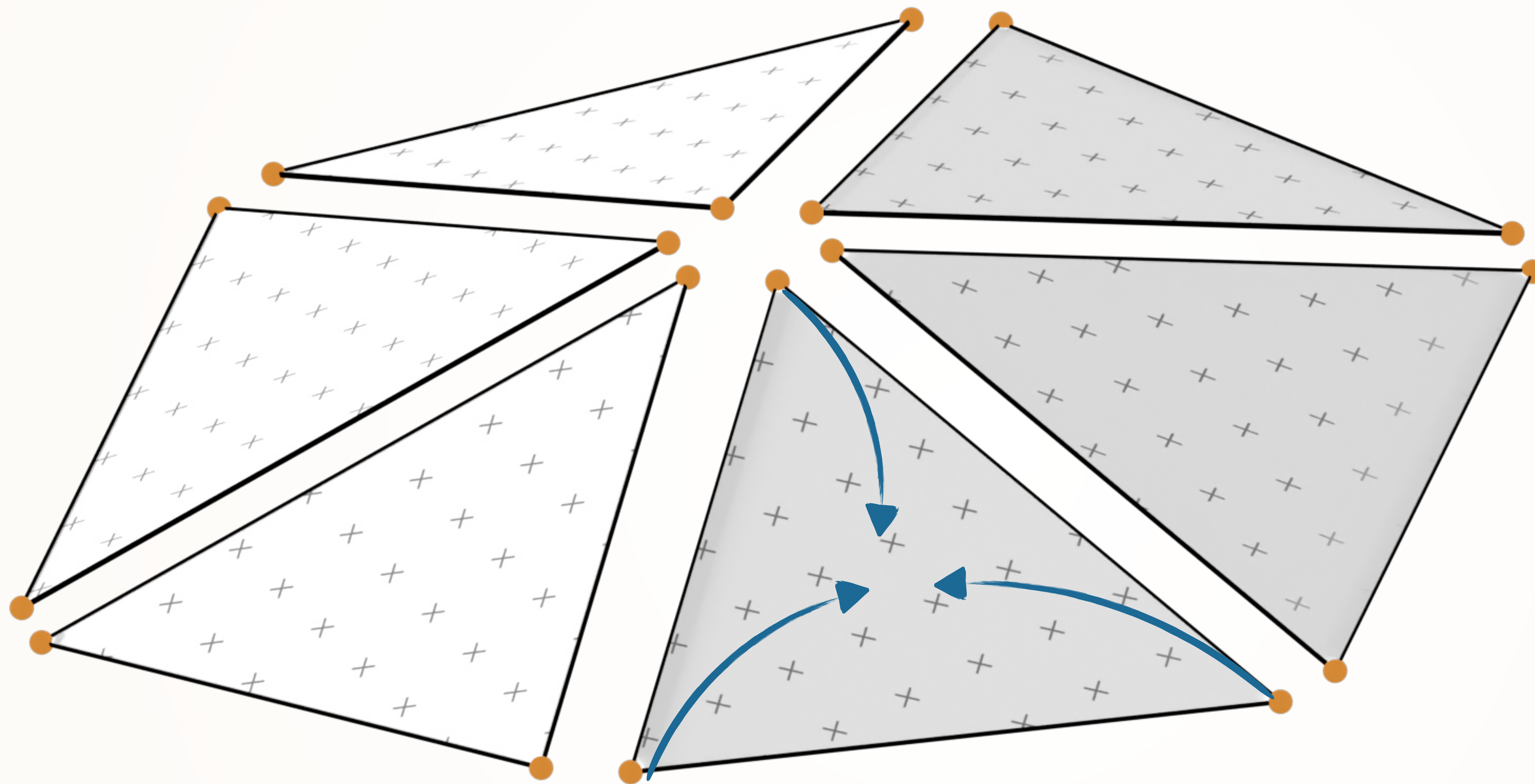
cf. Yu [2022], Dodik [2026]



gather
0 on 2

Scatter-Gather

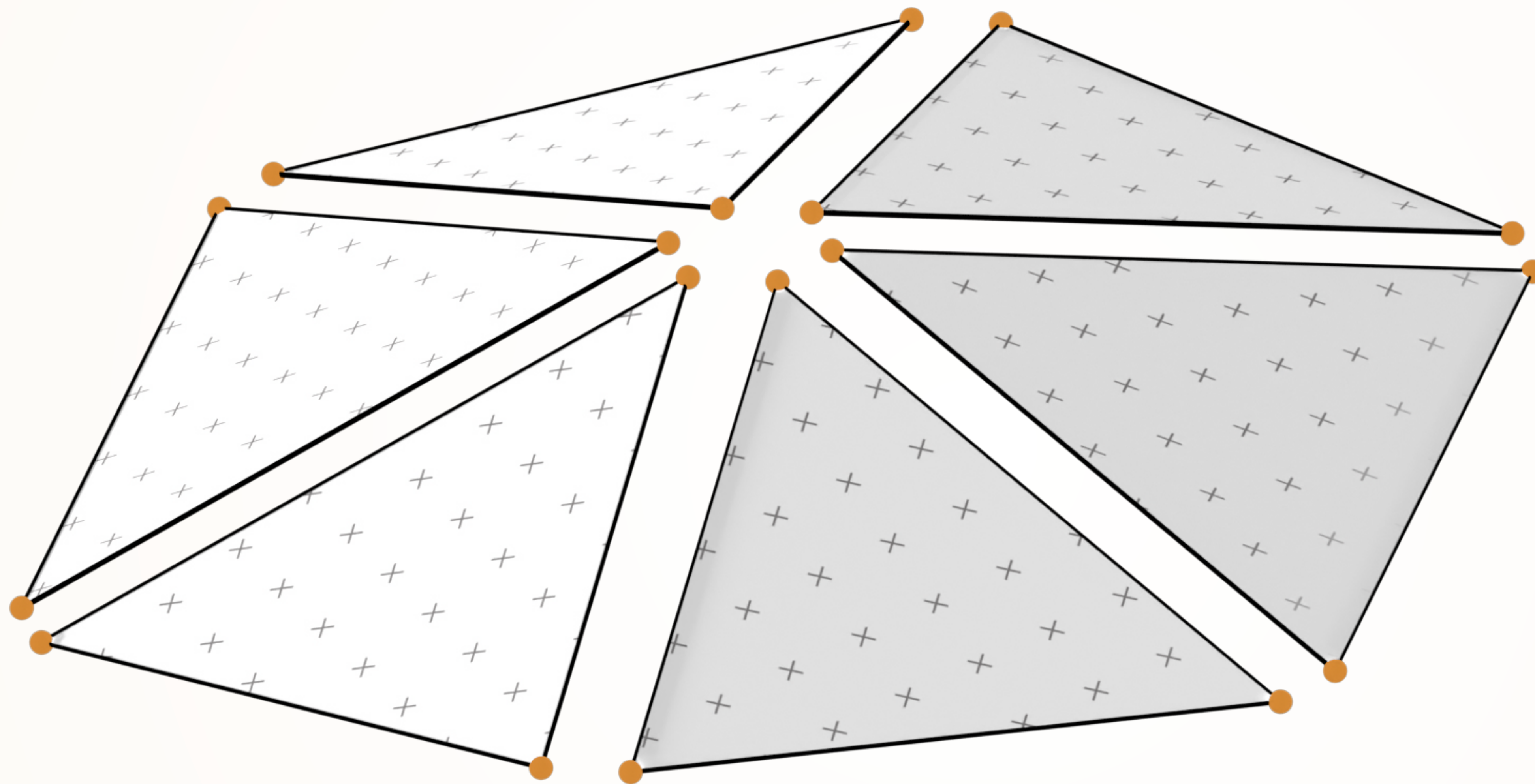
cf. Yu [2022], Dodik [2026]



gather
0 on 2

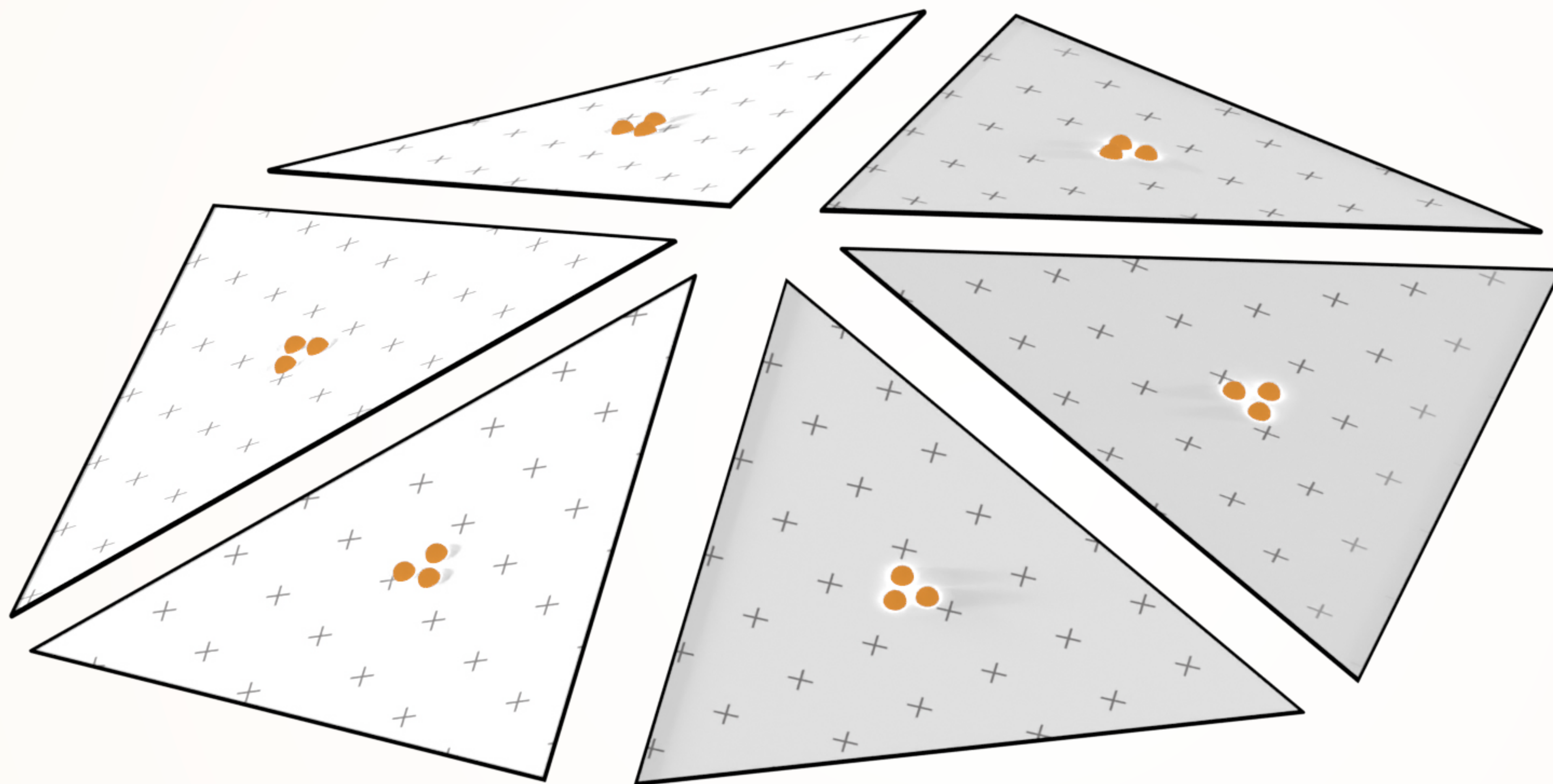
Scatter-Gather

cf. Yu [2022], Dodik [2026]



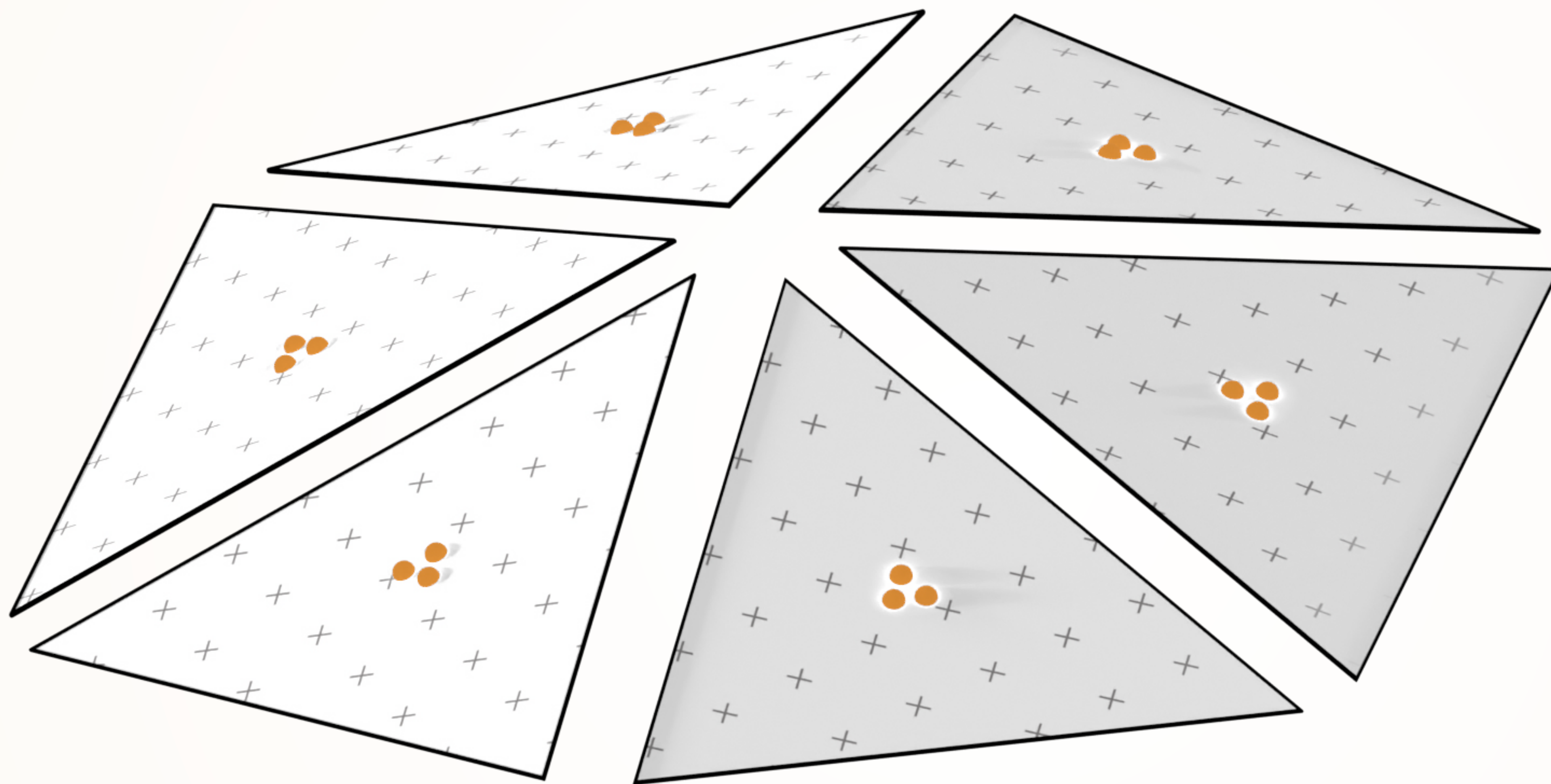
Scatter-Gather

cf. Yu [2022], Dodik [2026]



Scatter-Gather

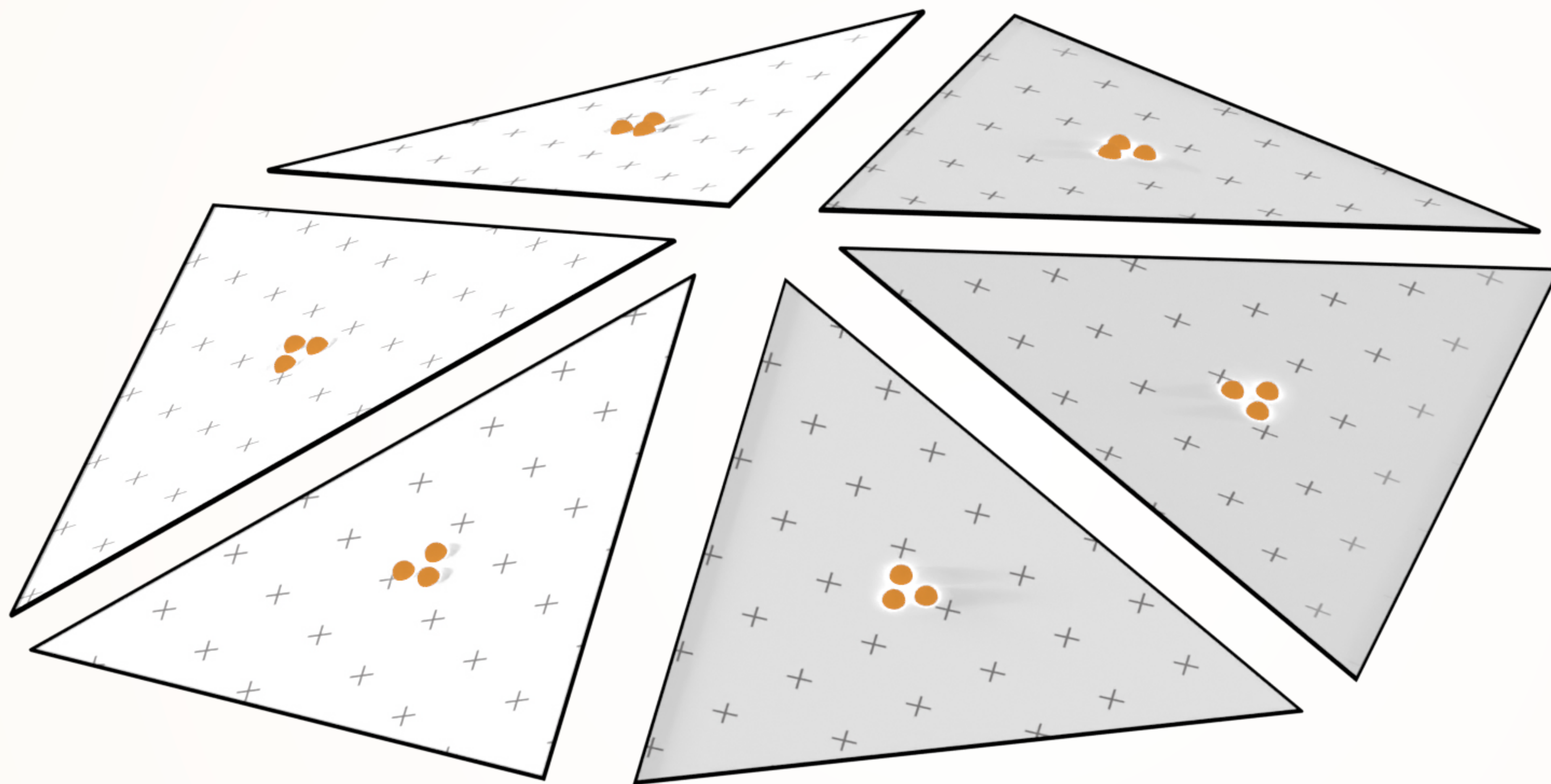
cf. Yu [2022], Dodik [2026]



compute
face
normals

Scatter-Gather

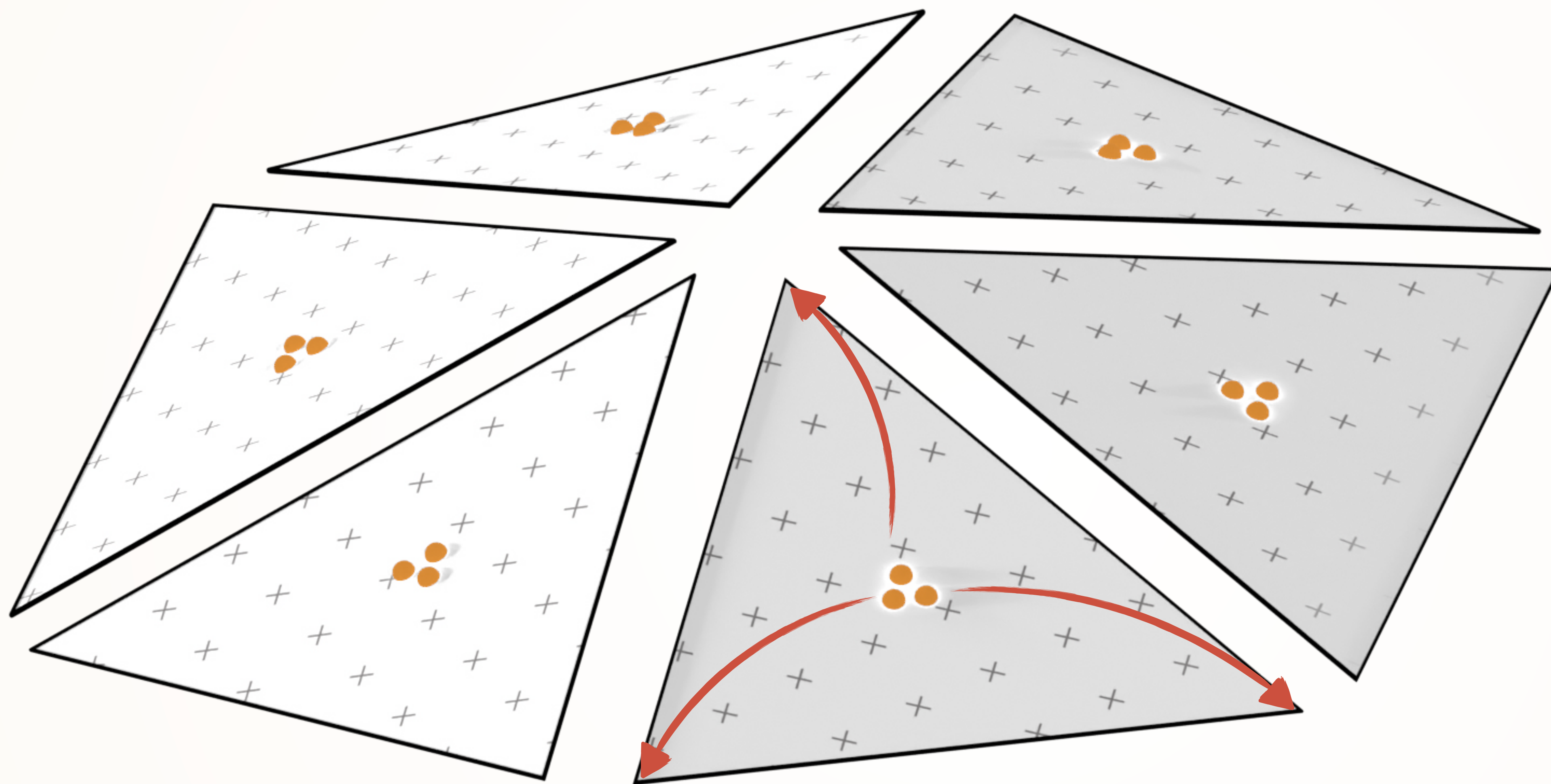
cf. Yu [2022], Dodik [2026]



compute
face
normals

Scatter-Gather

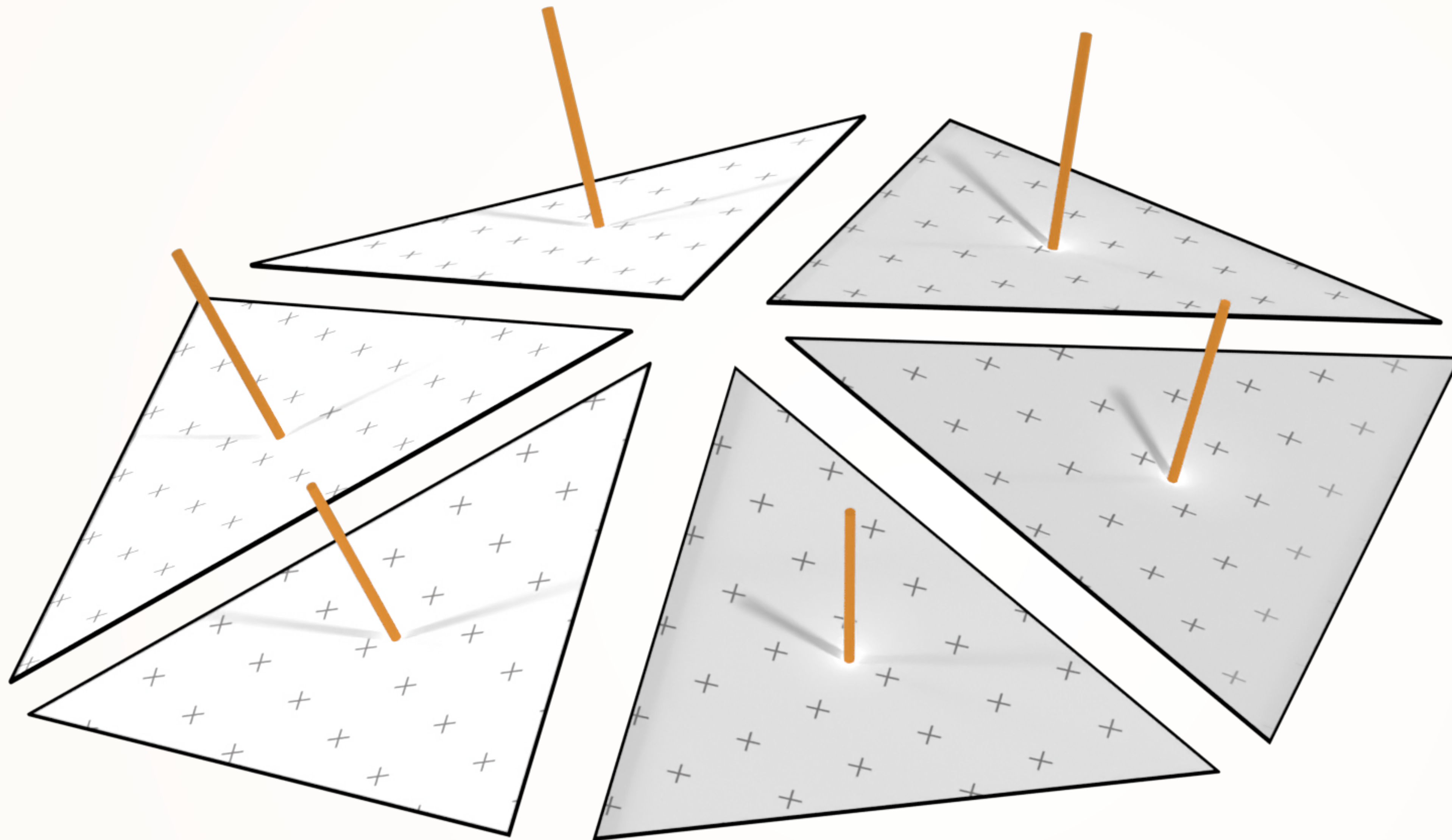
cf. Yu [2022], Dodik [2026]



scatter
2 to 0

Scatter-Gather

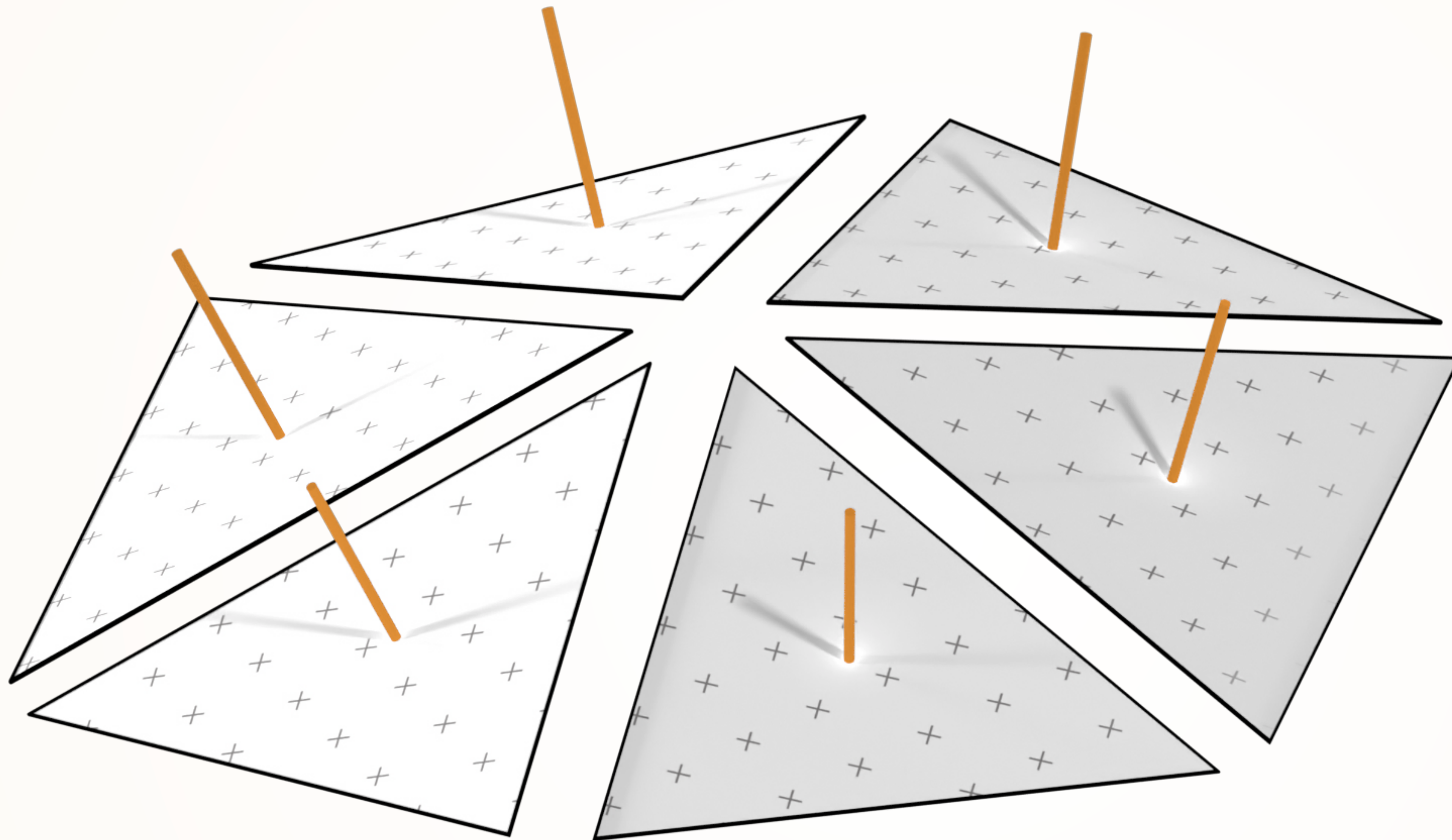
cf. Yu [2022], Dodik [2026]



scatter
2 to 0

Scatter-Gather

cf. Yu [2022], Dodik [2026]

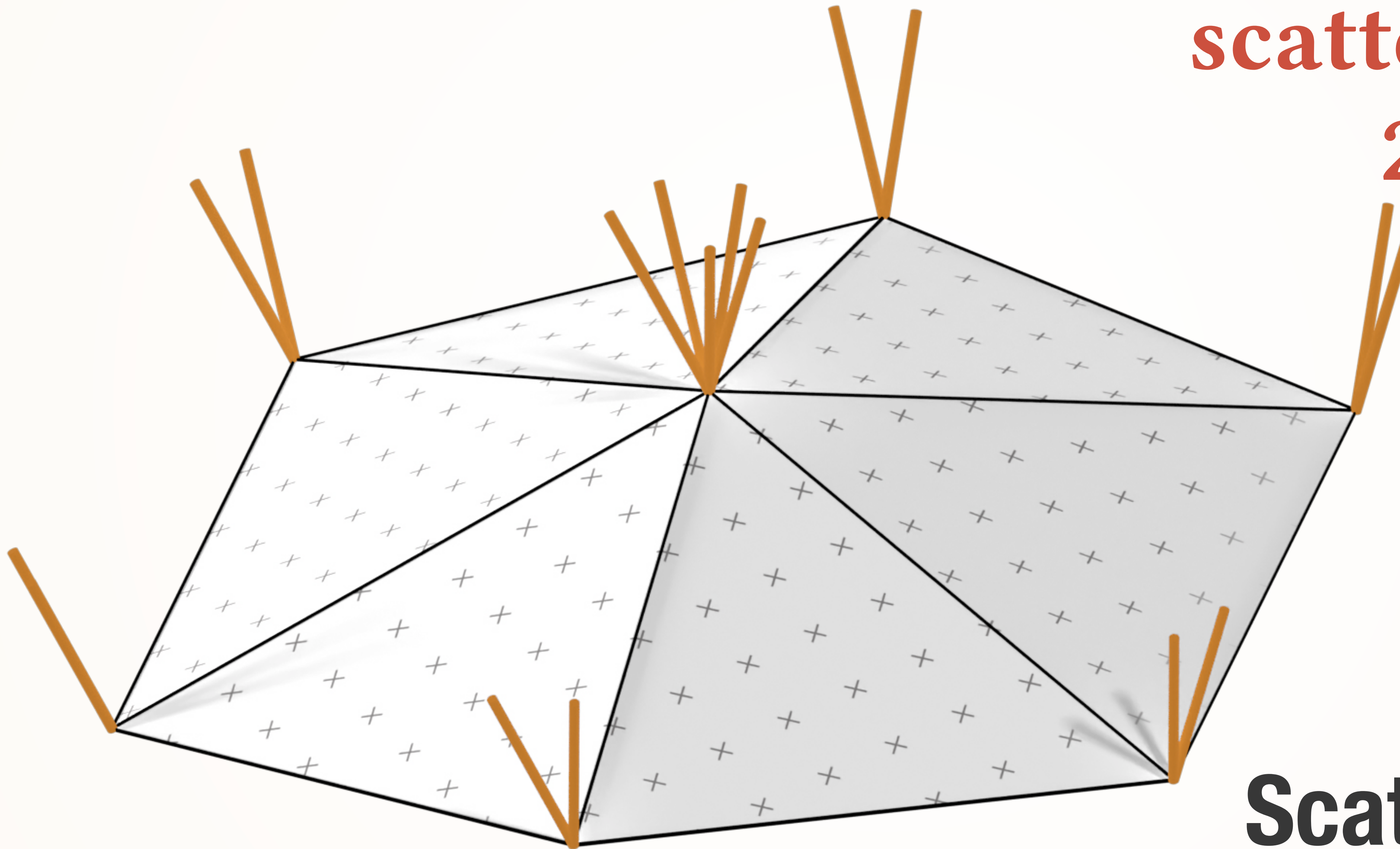


scatter
2 to 0

Scatter-Gather

cf. Yu [2022], Dodik [2026]

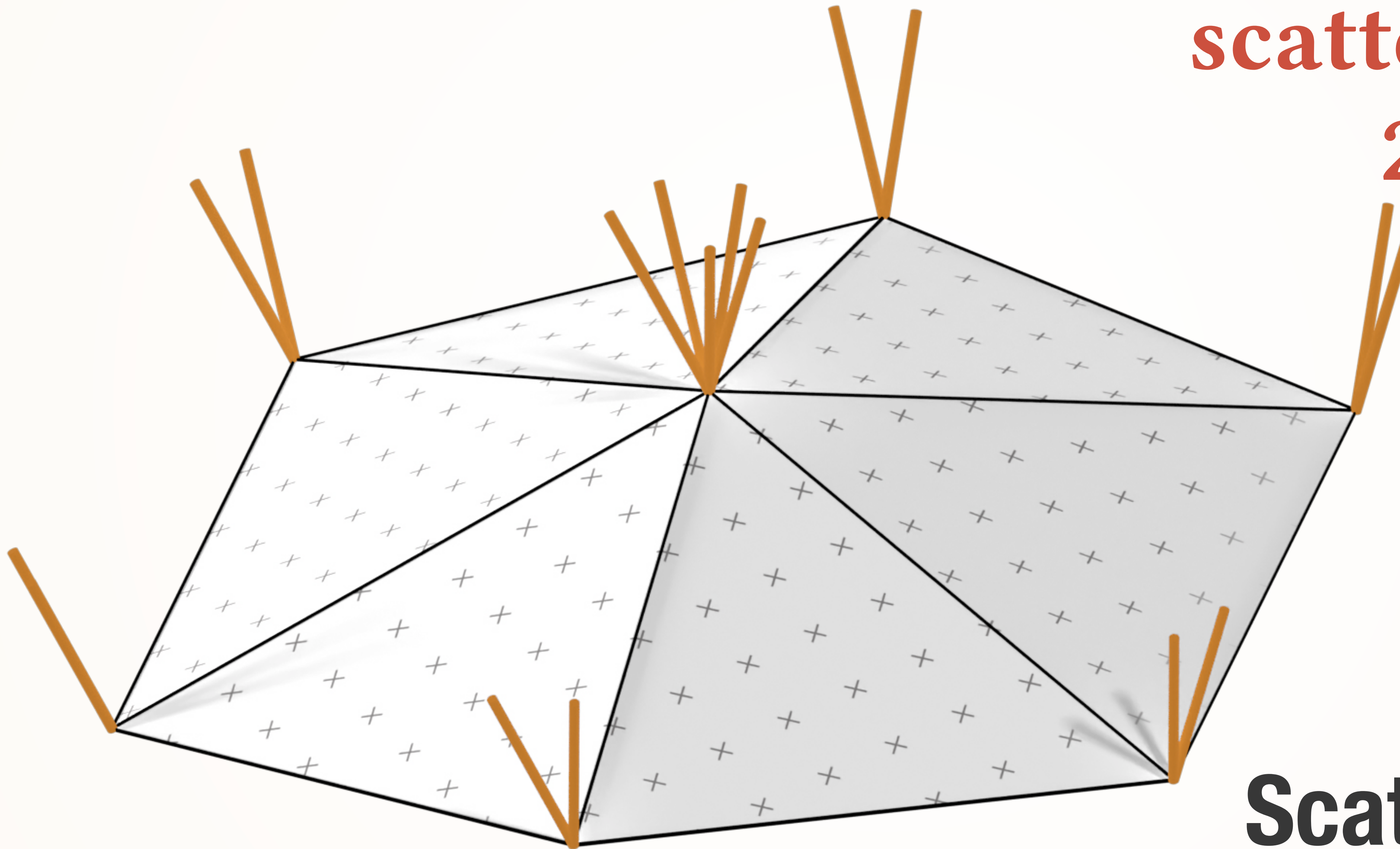
scatter-reduce
2 to 0



Scatter-Gather

cf. Yu [2022], Dodik [2026]

scatter-reduce
2 to 0



Scatter-Gather

cf. Yu [2022], Dodik [2026]

```
from torch.nn.functional import normalize

from iskra.geometry import triangle_normals
from iskra.topology import face_index, reduce_on_subface

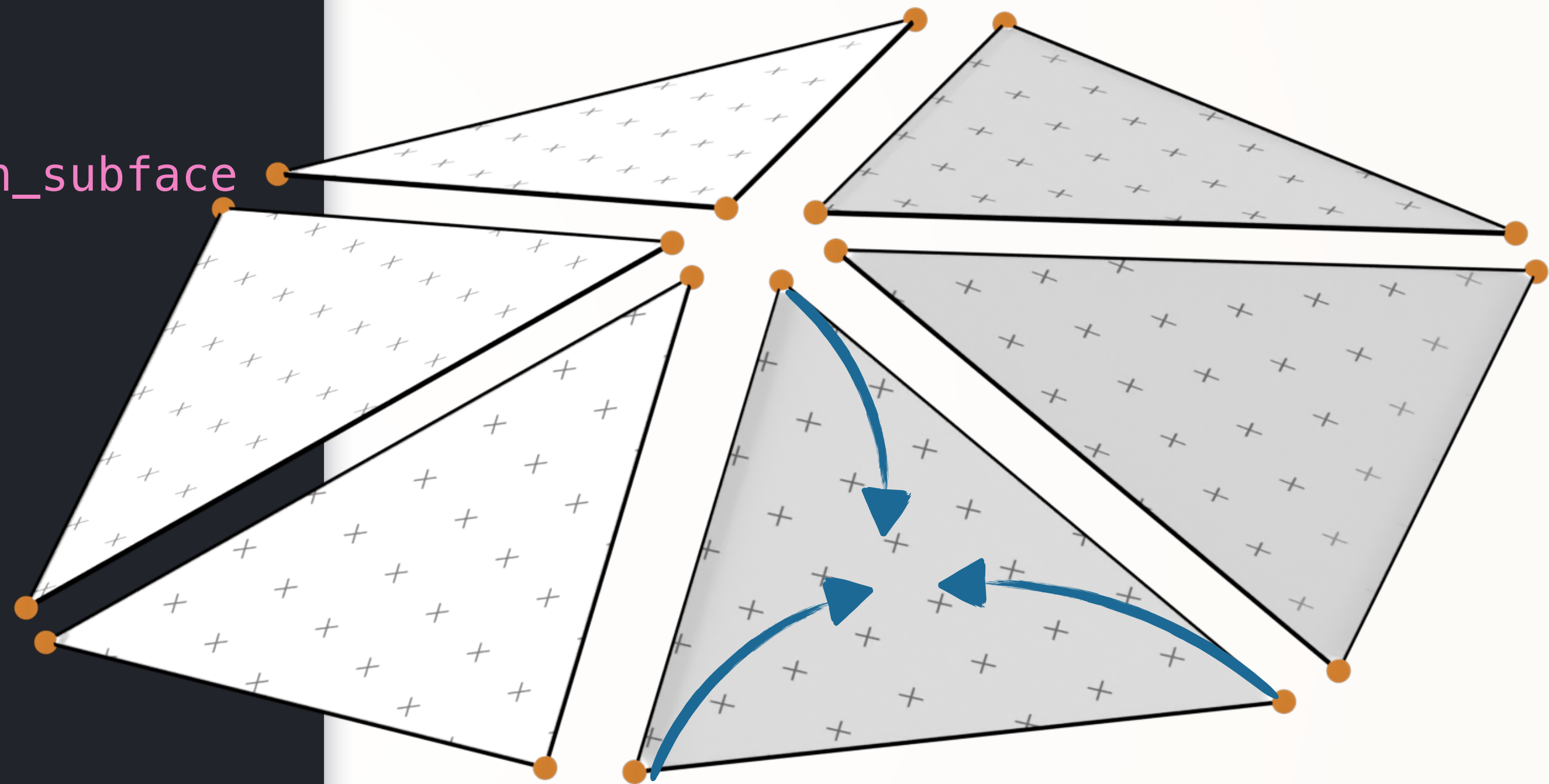
verts = mesh.vertices # [V, 3]
faces = mesh.faces # [F, 3]
```

```
from torch.nn.functional import normalize

from iskra.geometry import triangle_normals
from iskra.topology import face_index, reduce_on_subface

verts = mesh.vertices # [V, 3]
faces = mesh.faces # [F, 3]

tris = face_index(verts, faces) # [F, 3, 3]
```




```

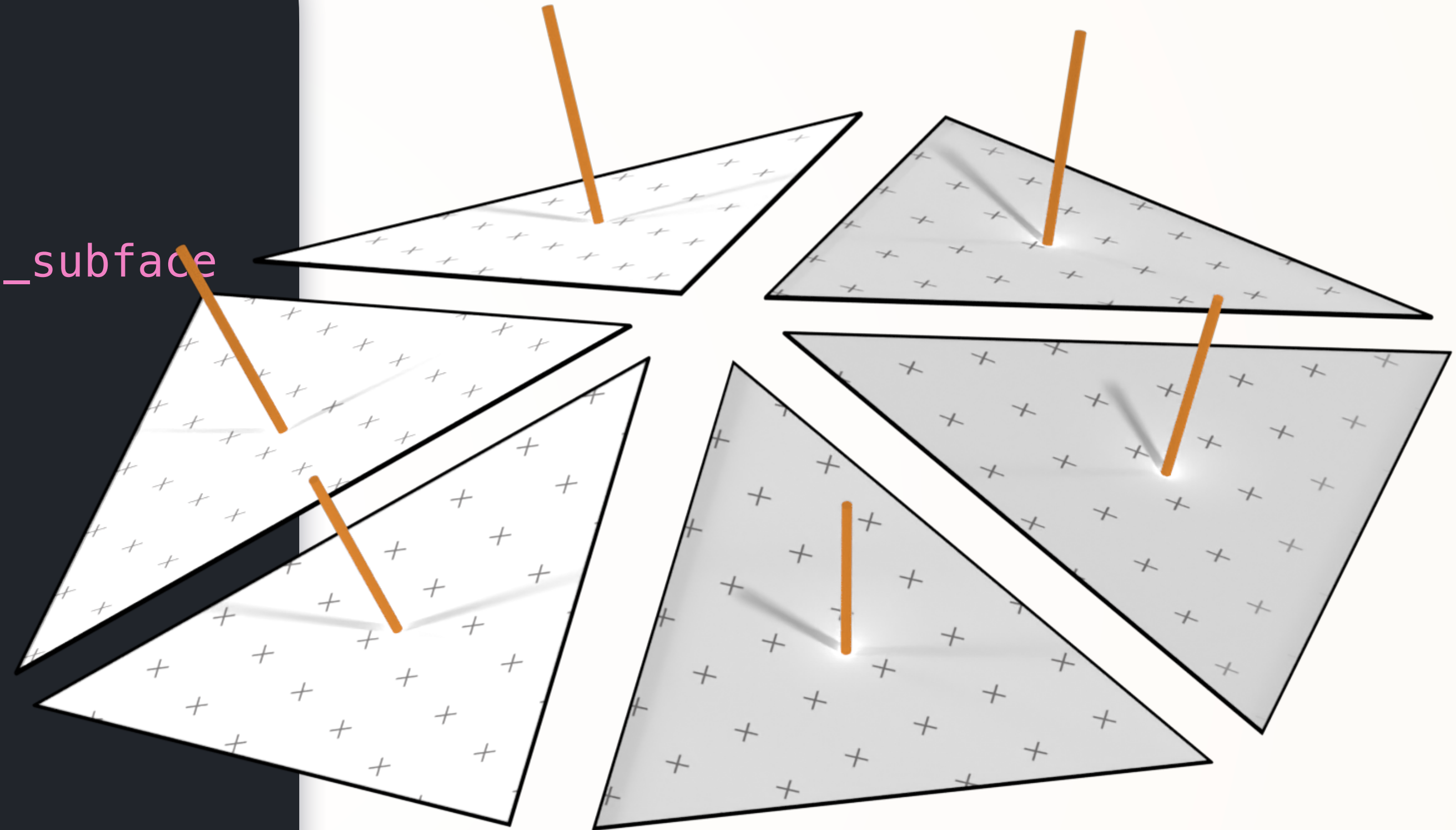
from torch.nn.functional import normalize

from iskra.geometry import triangle_normals
from iskra.topology import face_index, reduce_on_subface

verts = mesh.vertices # [V, 3]
faces = mesh.faces # [F, 3]

tris = face_index(verts, faces) # [F, 3, 3]
tri_normals = triangle_normals(tris) # [F, 3]

```



```

from torch.nn.functional import normalize

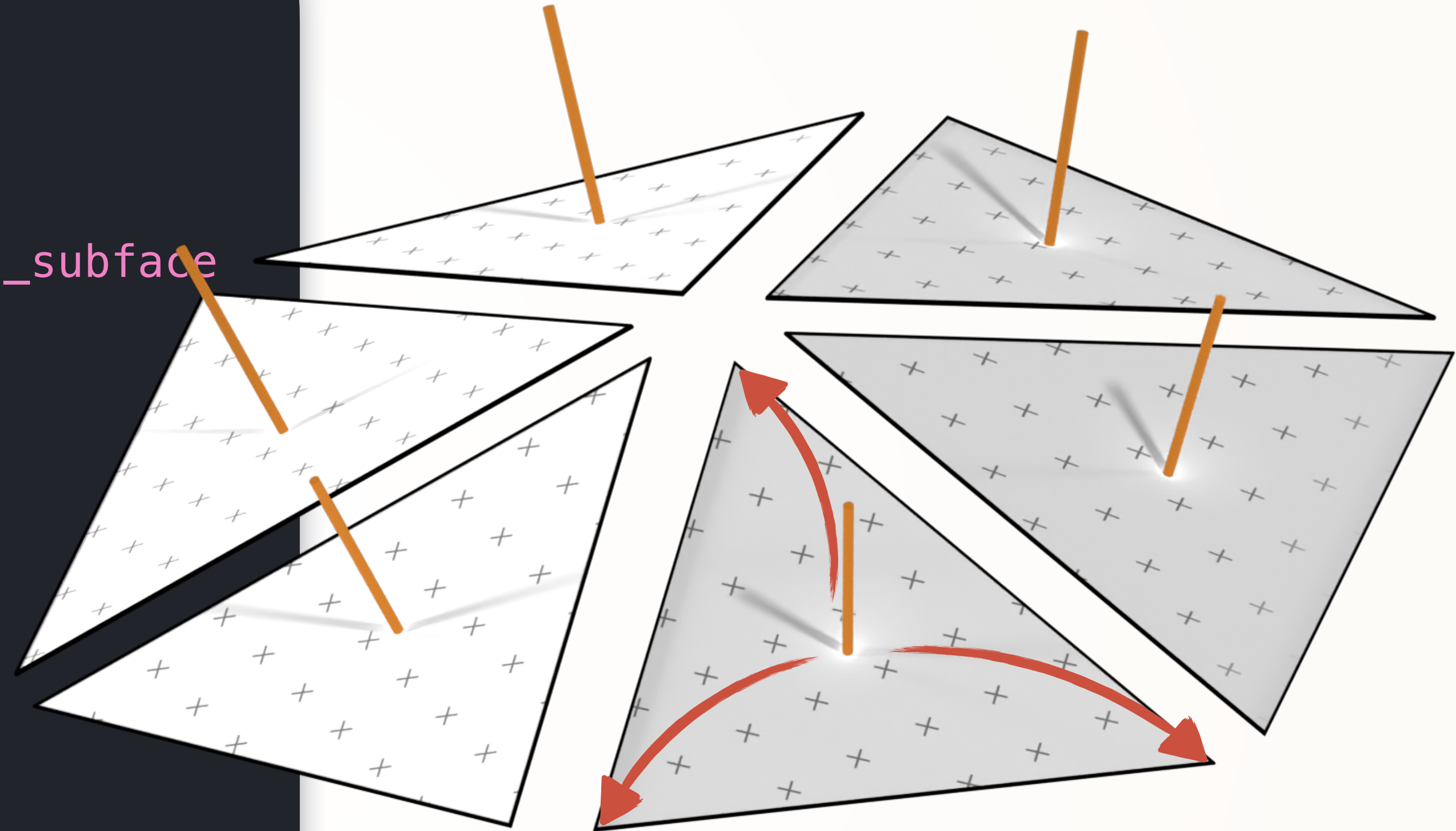
from iskra.geometry import triangle_normals
from iskra.topology import face_index, reduce_on_subface

verts = mesh.vertices # [V, 3]
faces = mesh.faces # [F, 3]

tris = face_index(verts, faces) # [F, 3, 3]
tri_normals = triangle_normals(tris) # [F, 3]

vert_normals = reduce_on_subface( # [V, 3]
    tri_normals, faces, verts.shape[0], "sum"
)
vert_normals = normalize(vert_normals, dim=-1)

```



```

from torch.nn.functional import normalize

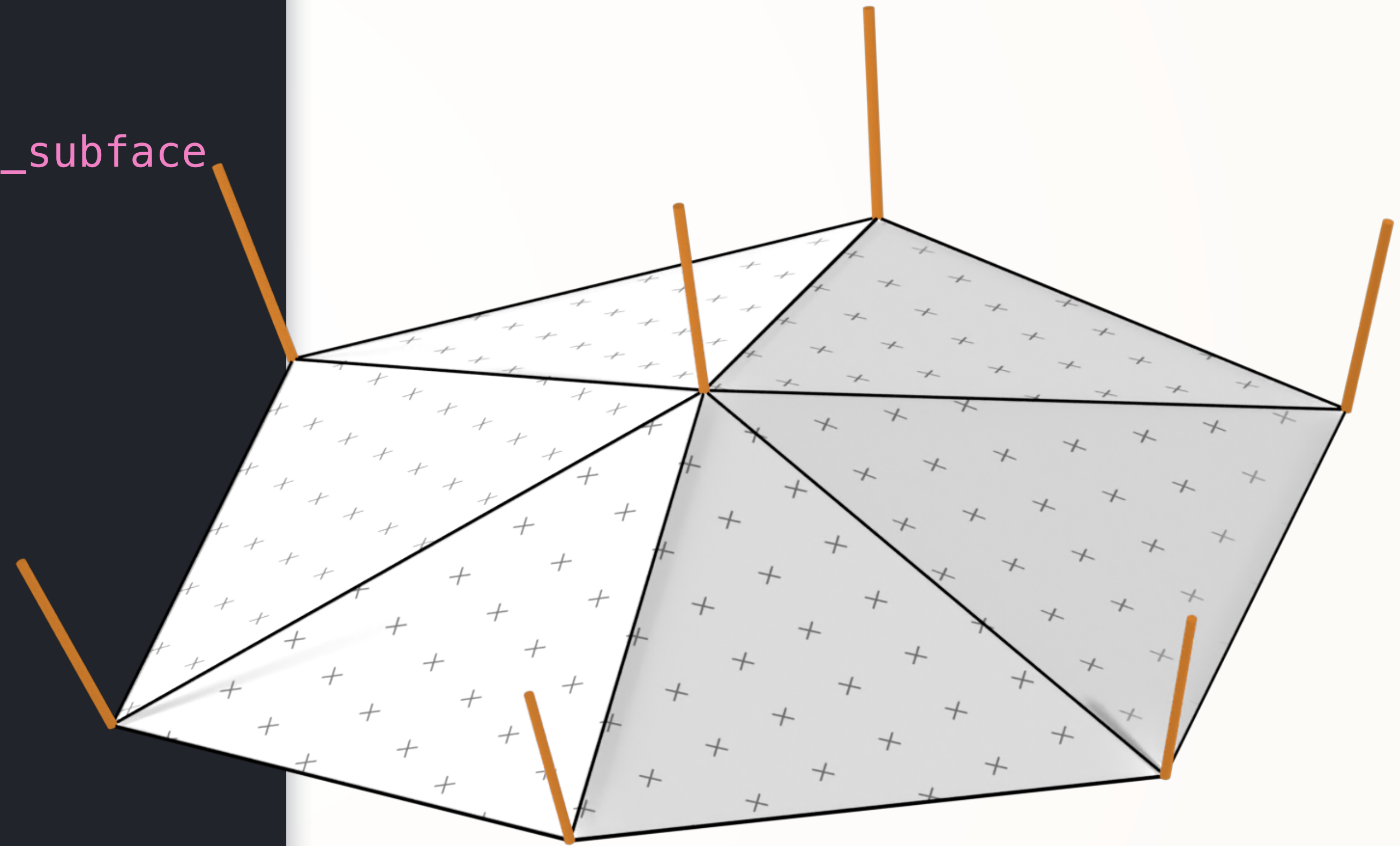
from iskra.geometry import triangle_normals
from iskra.topology import face_index, reduce_on_subface

verts = mesh.vertices # [V, 3]
faces = mesh.faces # [F, 3]

tris = face_index(verts, faces) # [F, 3, 3]
tri_normals = triangle_normals(tris) # [F, 3]

vert_normals = reduce_on_subface( # [V, 3]
    tri_normals, faces, verts.shape[0], "sum"
)
vert_normals = normalize(vert_normals, dim=-1)

```




```

from torch.nn.functional import normalize

from iskra.geometry import triangle_normals
from iskra.topology import face_index, reduce_on_subface

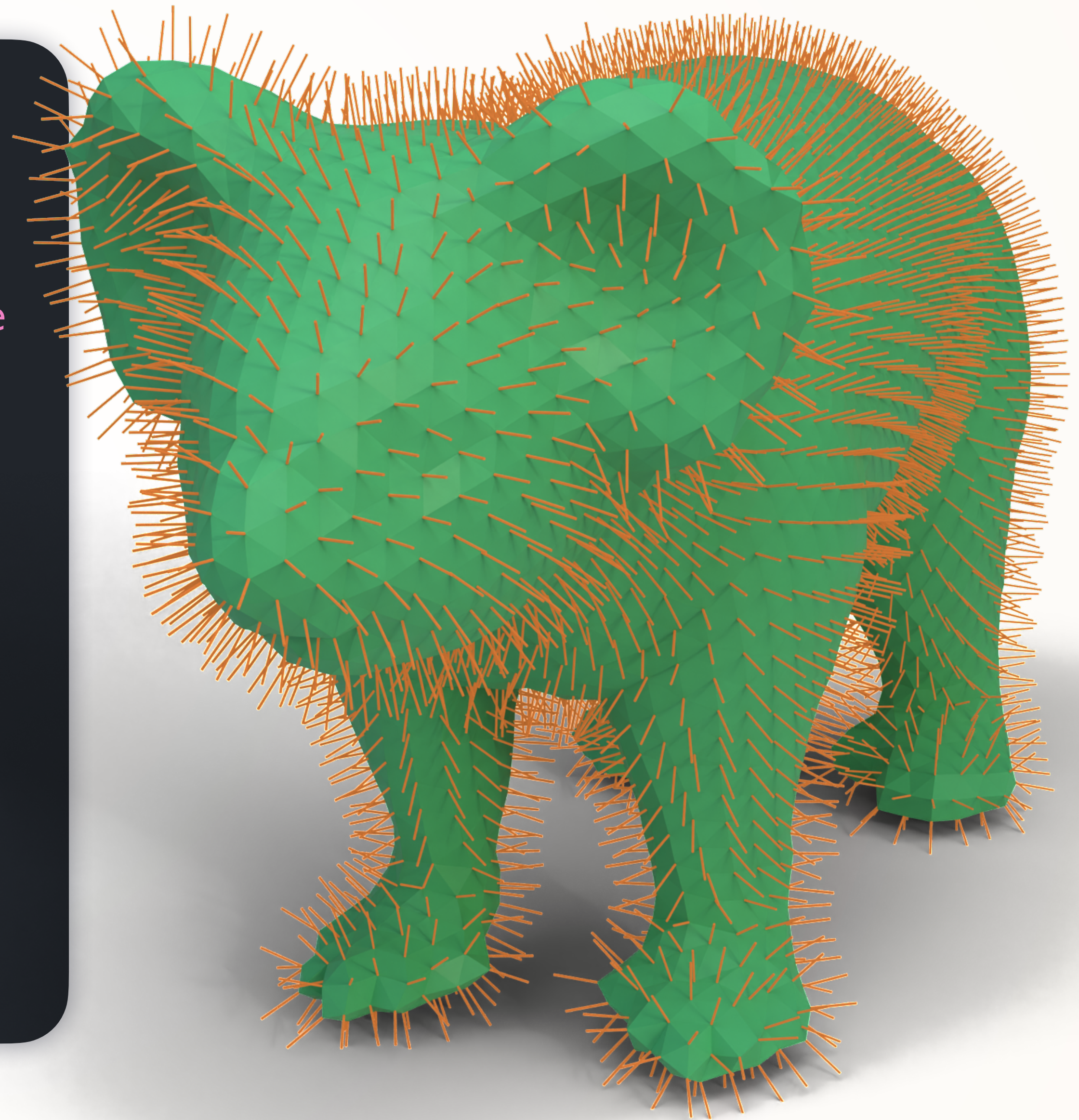
verts = mesh.vertices # [V, 3]
faces = mesh.faces # [F, 3]

tris = face_index(verts, faces) # [F, 3, 3]

tri_normals = triangle_normals(tris) # [F, 3]

vert_normals = reduce_on_subface( # [V, 3]
    tri_normals, faces, verts.shape[0], "sum"
)
vert_normals = normalize(vert_normals, dim=-1)

```

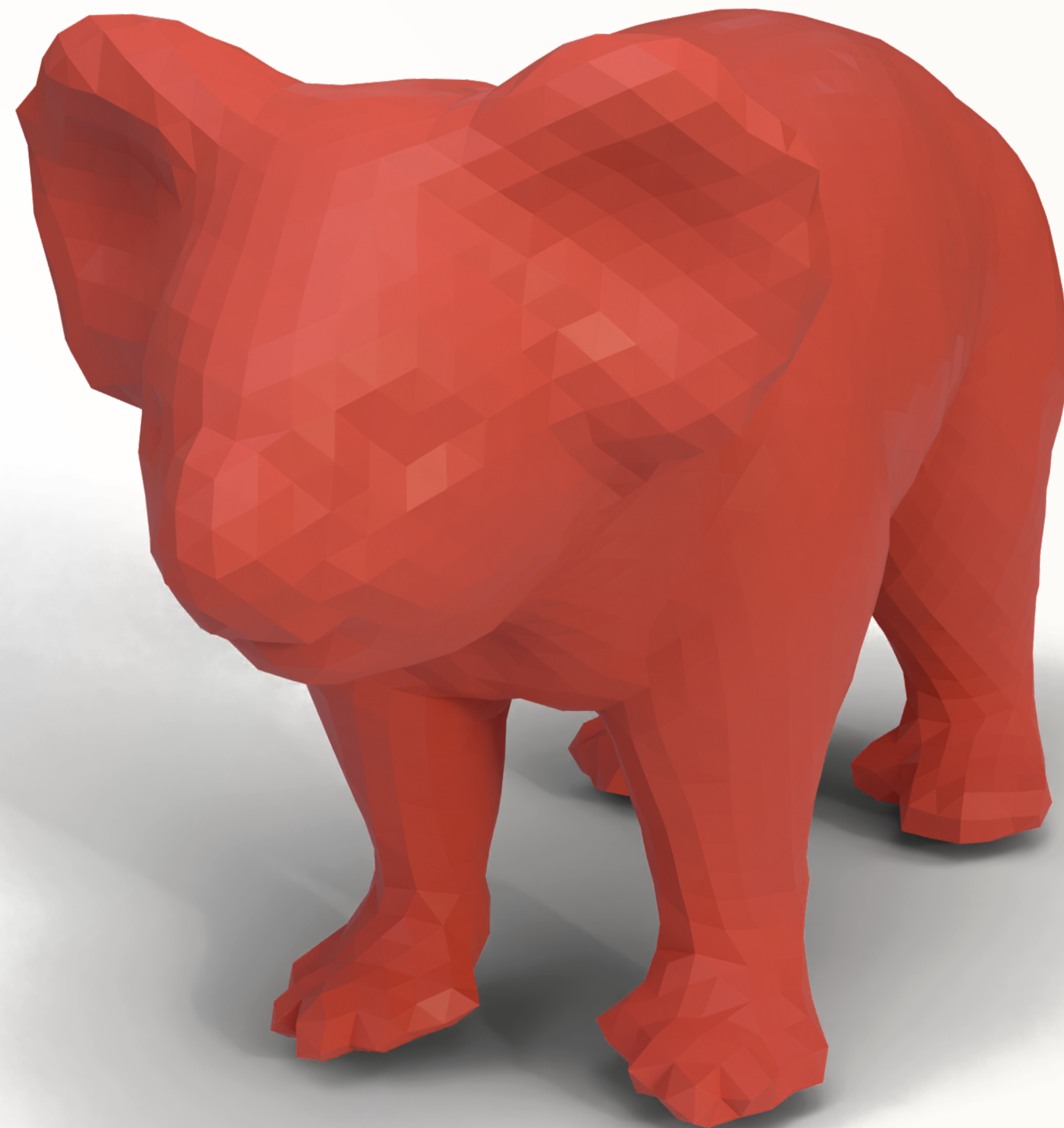


$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (L\mathbf{v})_i$$

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega)$$

mean curvature flow

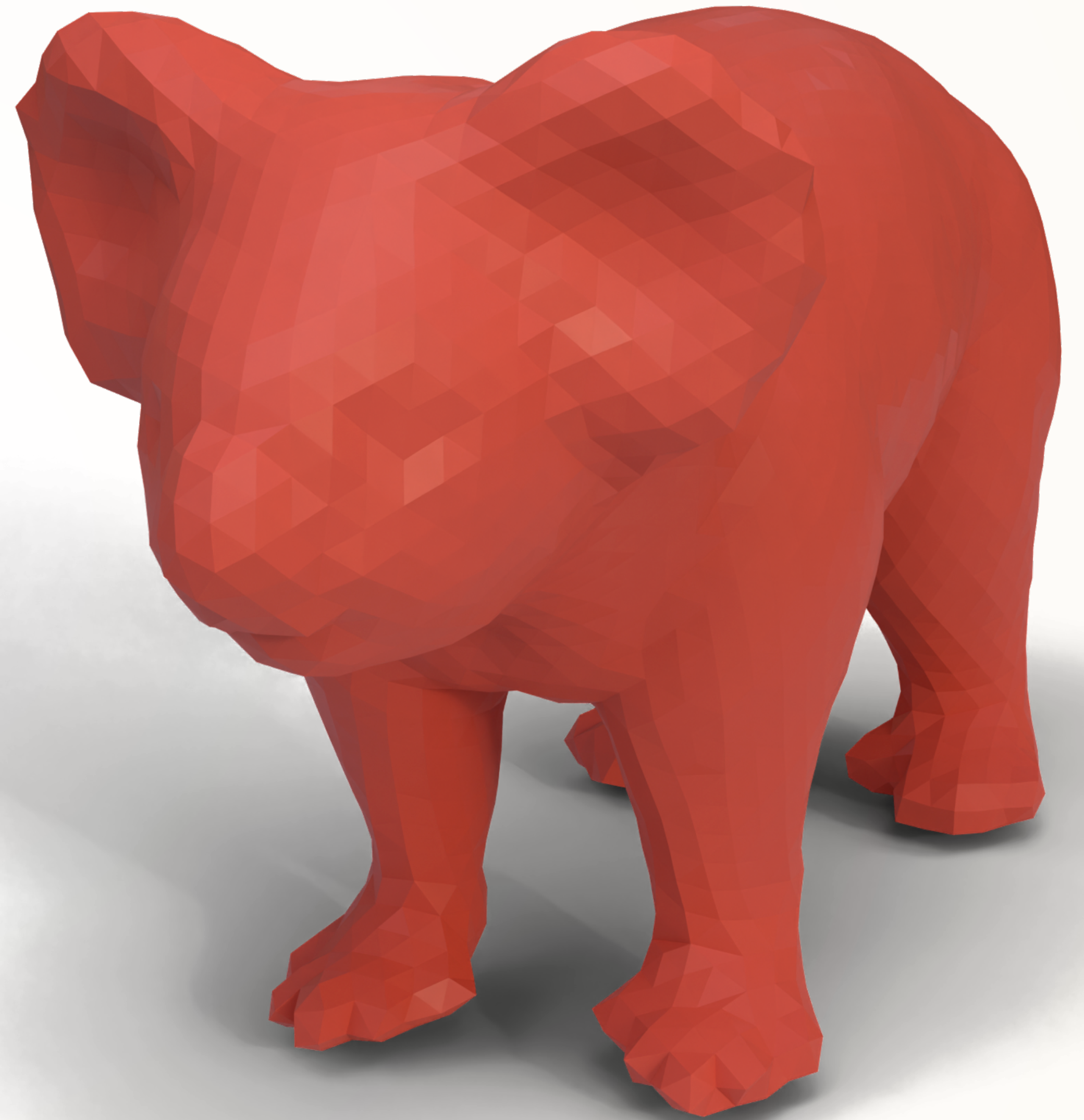


$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (L\mathbf{v})_i$$

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega)$$

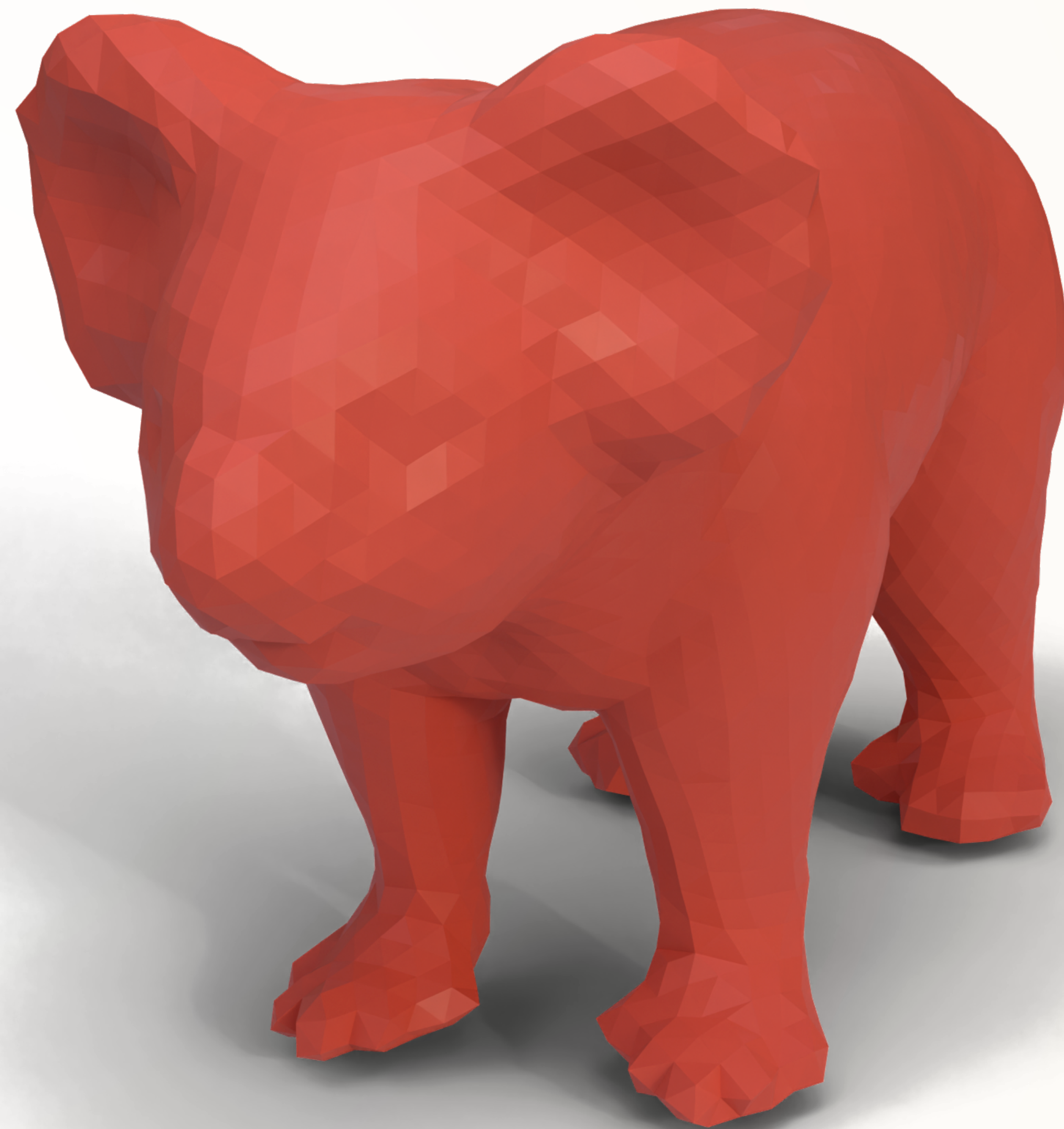
mean curvature flow



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{\boldsymbol{v}_i} \mathcal{E}(\Omega) = 2H_i \boldsymbol{n}_i = (L\boldsymbol{v})_i$$

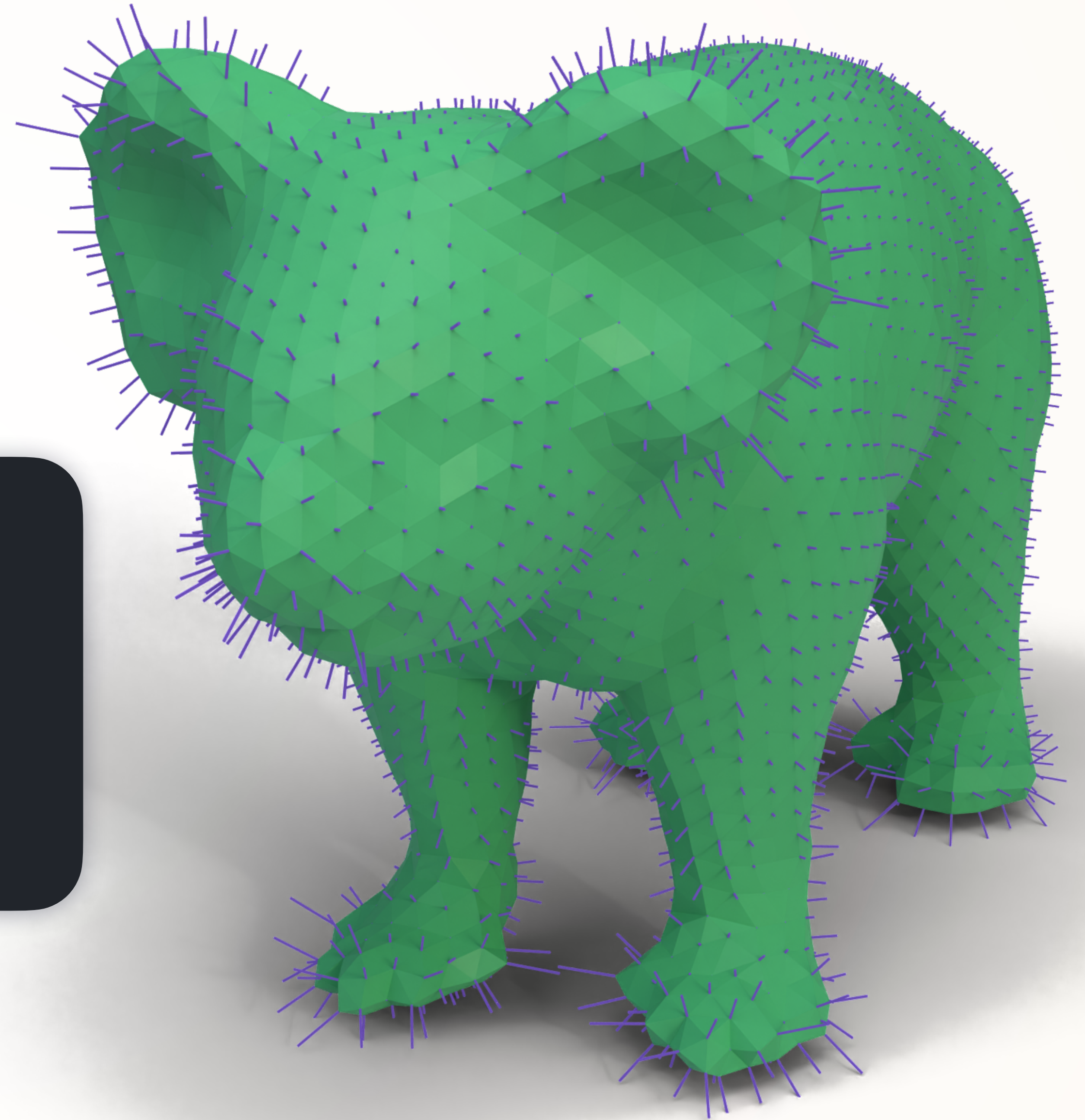
mean curvature flow



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

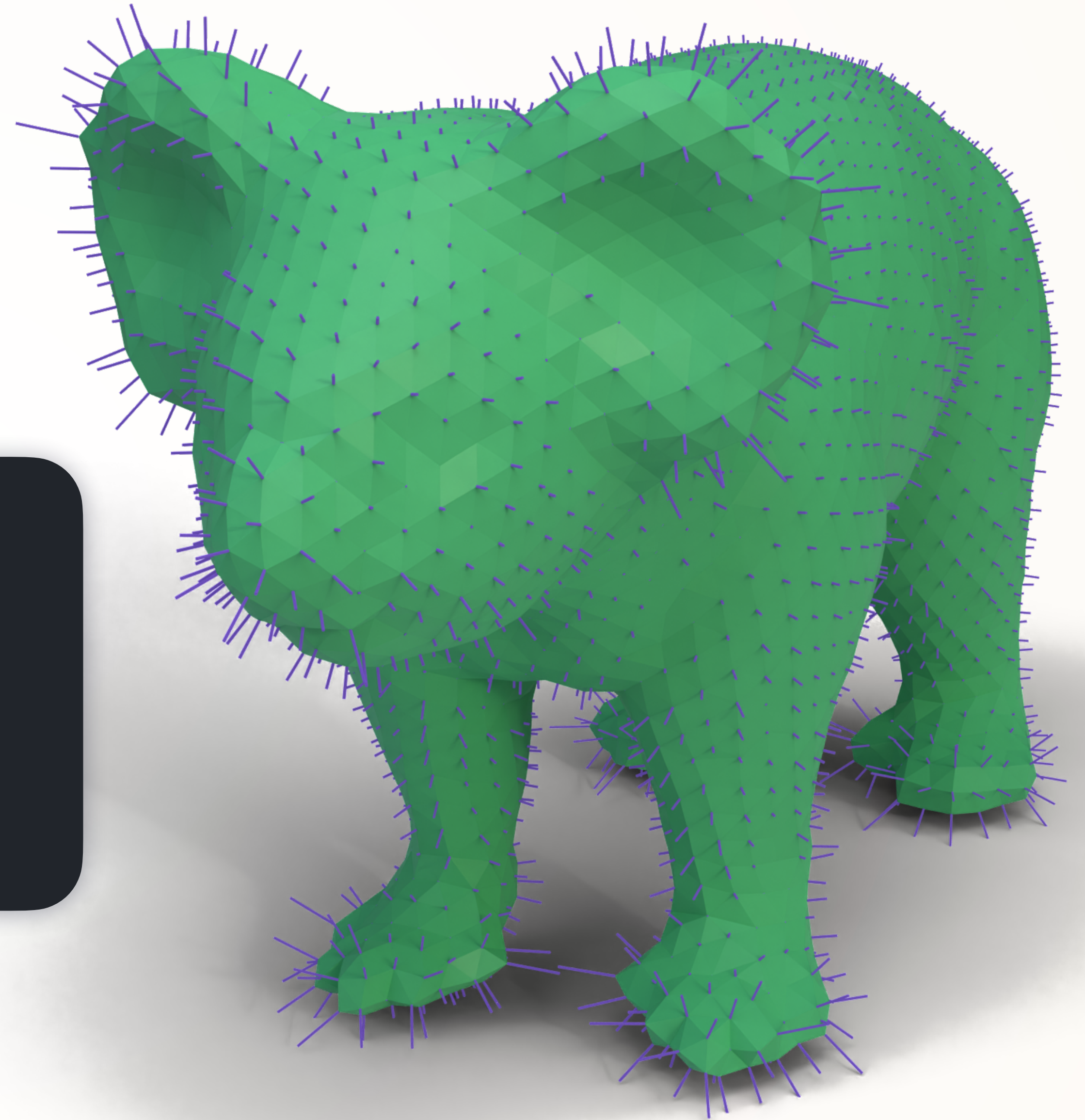
```
from iskra.geometry.volume import triangle_areas
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

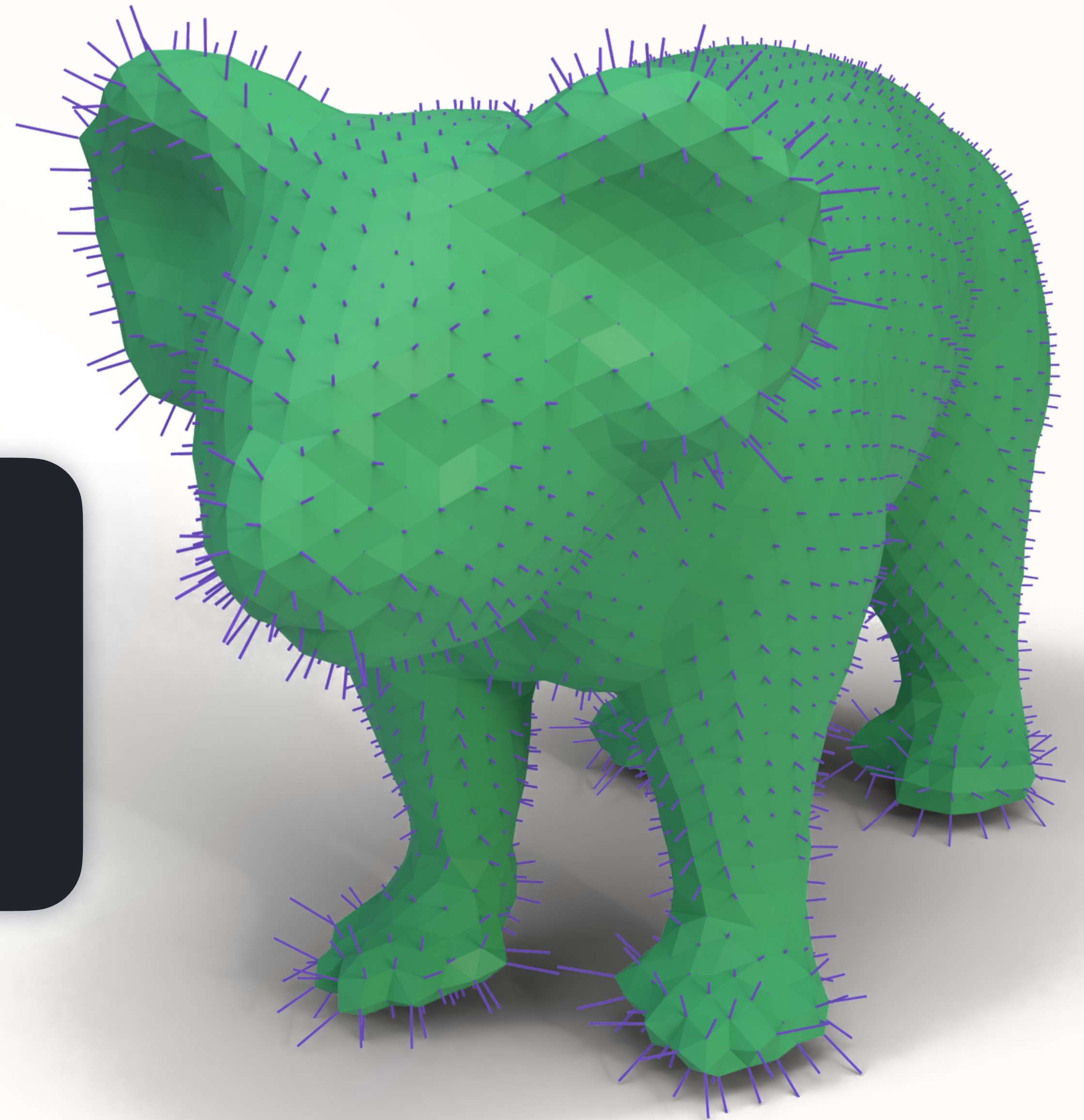
```
from iskra.geometry.volume import triangle_areas
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

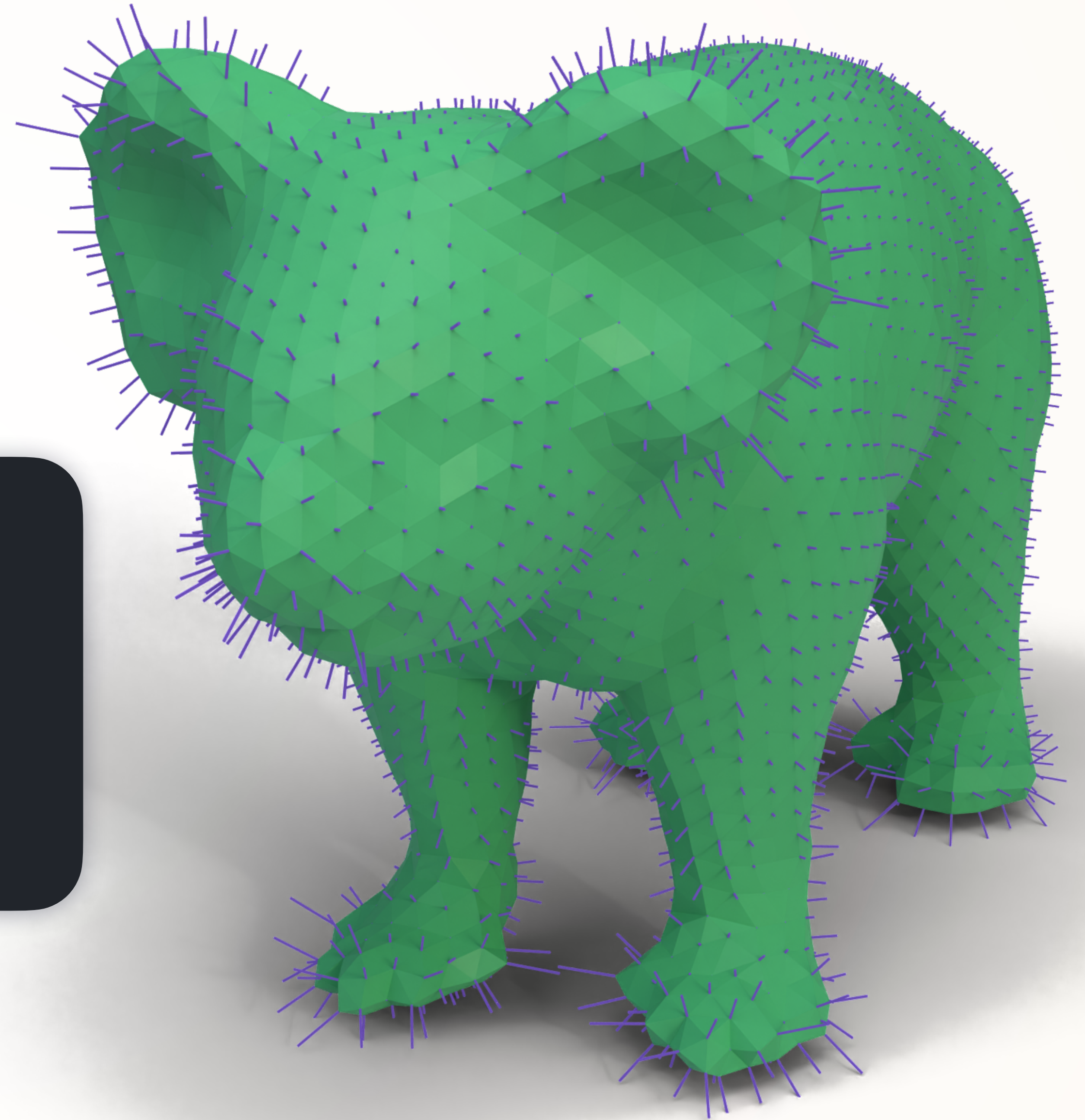
```
from iskra.geometry.volume import triangle_areas
verts_opt = verts.clone().requires_grad_(True)
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

```
from iskra.geometry.volume import triangle_areas
verts_opt = verts.clone().requires_grad_(True)
triangles = face_index(verts_opt, faces)
```

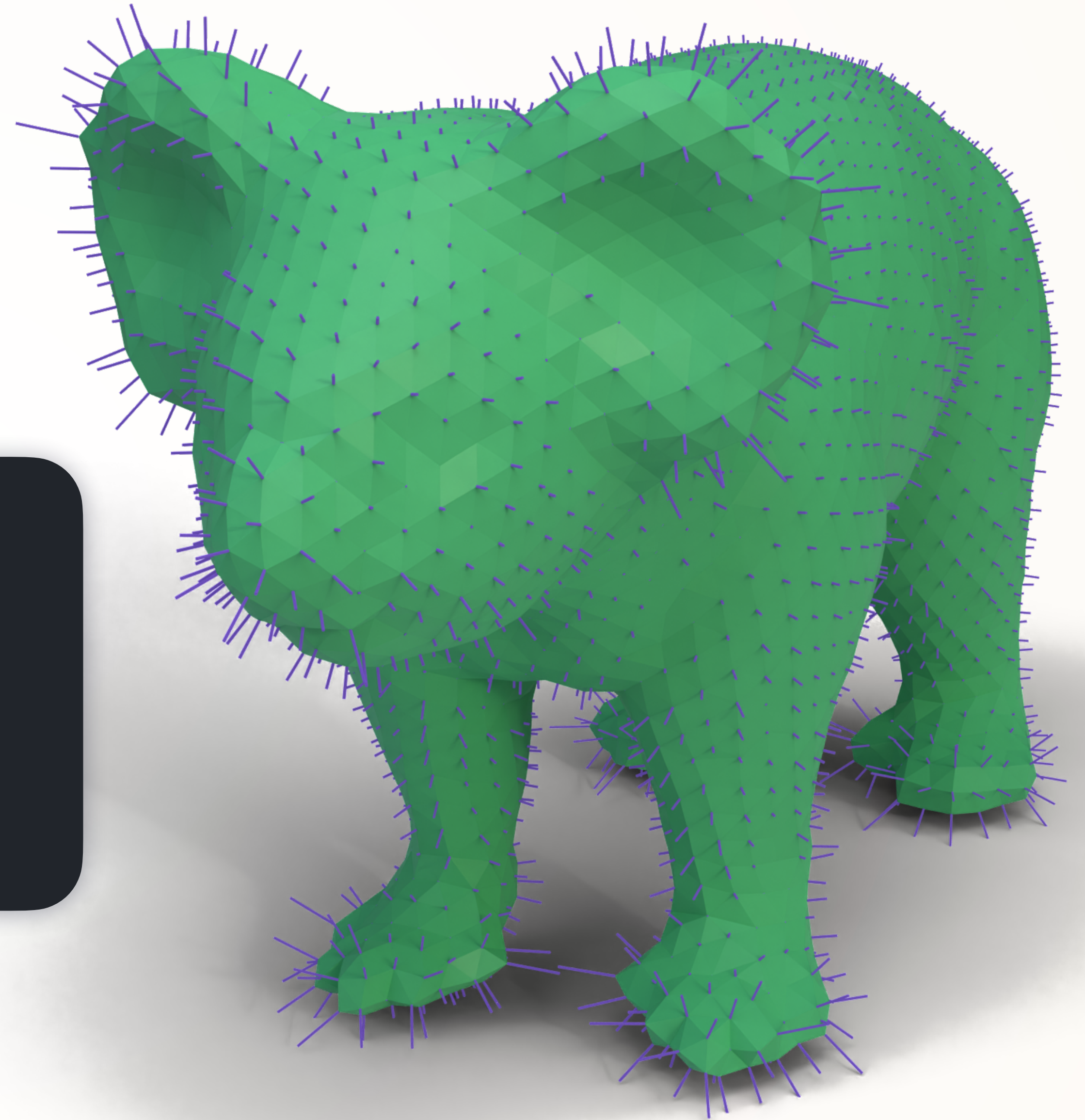


$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

```
from iskra.geometry.volume import triangle_areas

verts_opt = verts.clone().requires_grad_(True)
triangles = face_index(verts_opt, faces)
total_area = triangle_areas(triangles).sum()
```

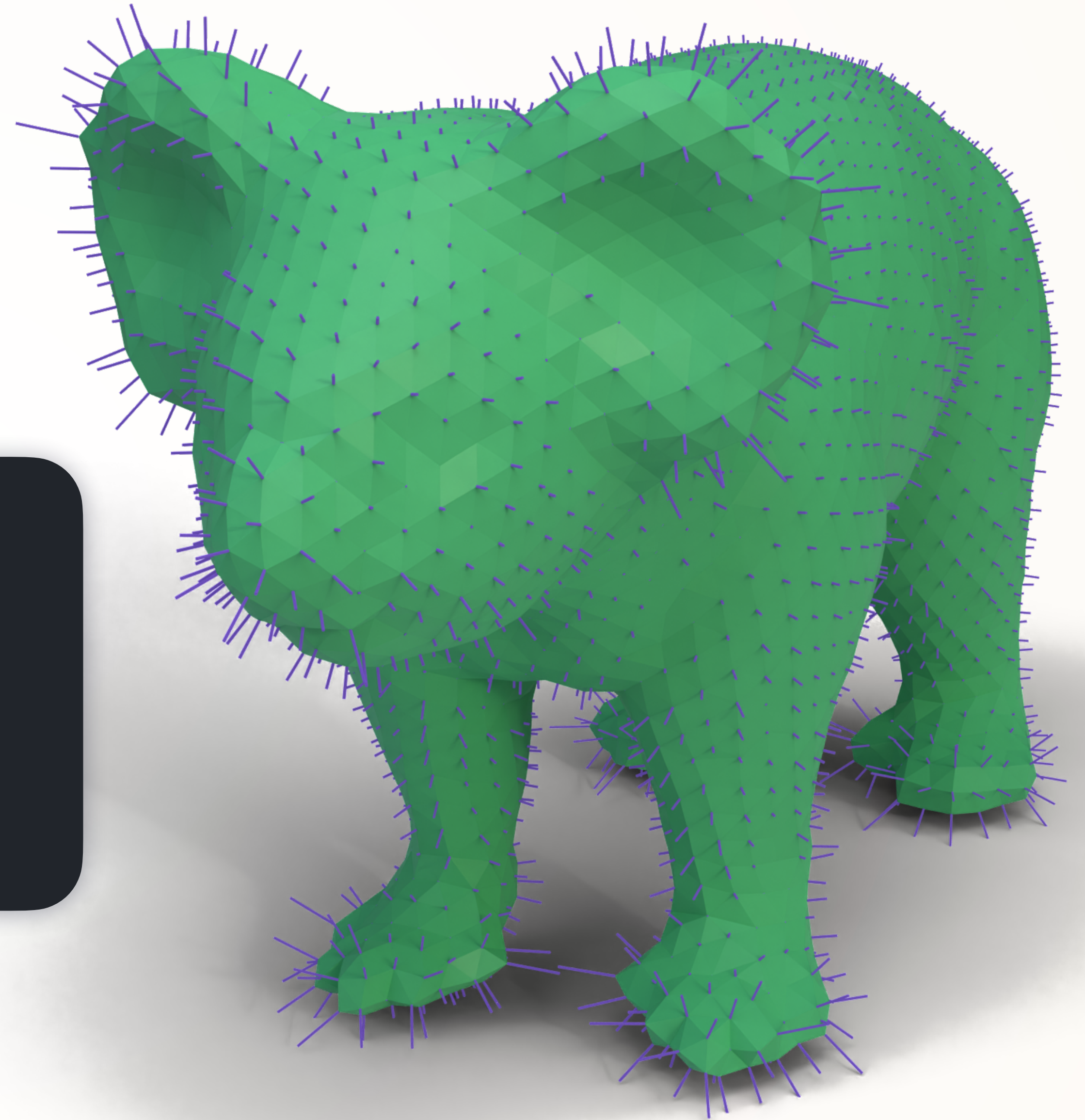


$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

```
from iskra.geometry.volume import triangle_areas

verts_opt = verts.clone().requires_grad_(True)
triangles = face_index(verts_opt, faces)
total_area = triangle_areas(triangles).sum()
total_area.backward()
```

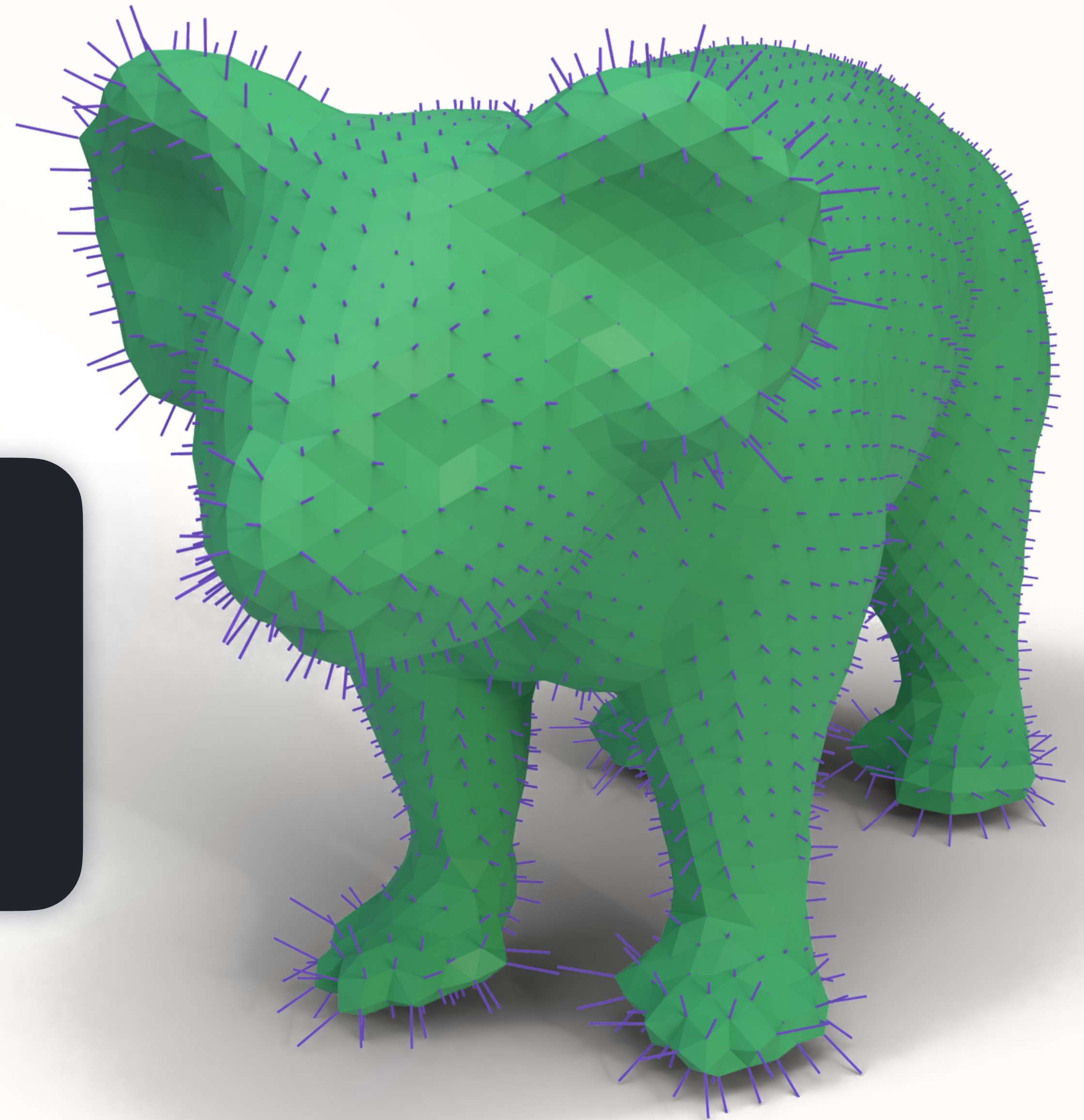


$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

```
from iskra.geometry.volume import triangle_areas

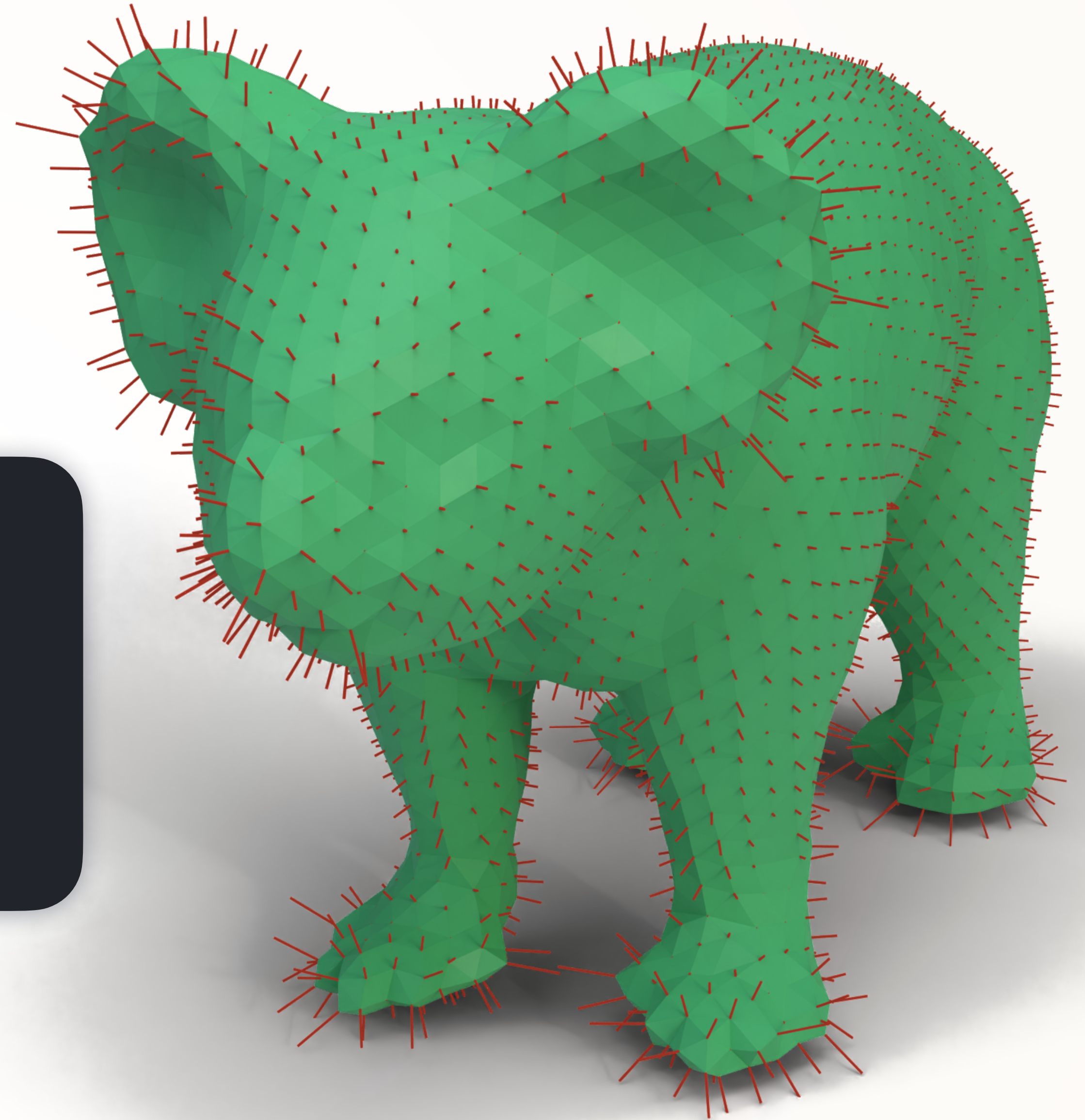
verts_opt = verts.clone().requires_grad_(True)
triangles = face_index(verts_opt, faces)
total_area = triangle_areas(triangles).sum()
total_area.backward()
mcf_normals_var = verts_opt.grad
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

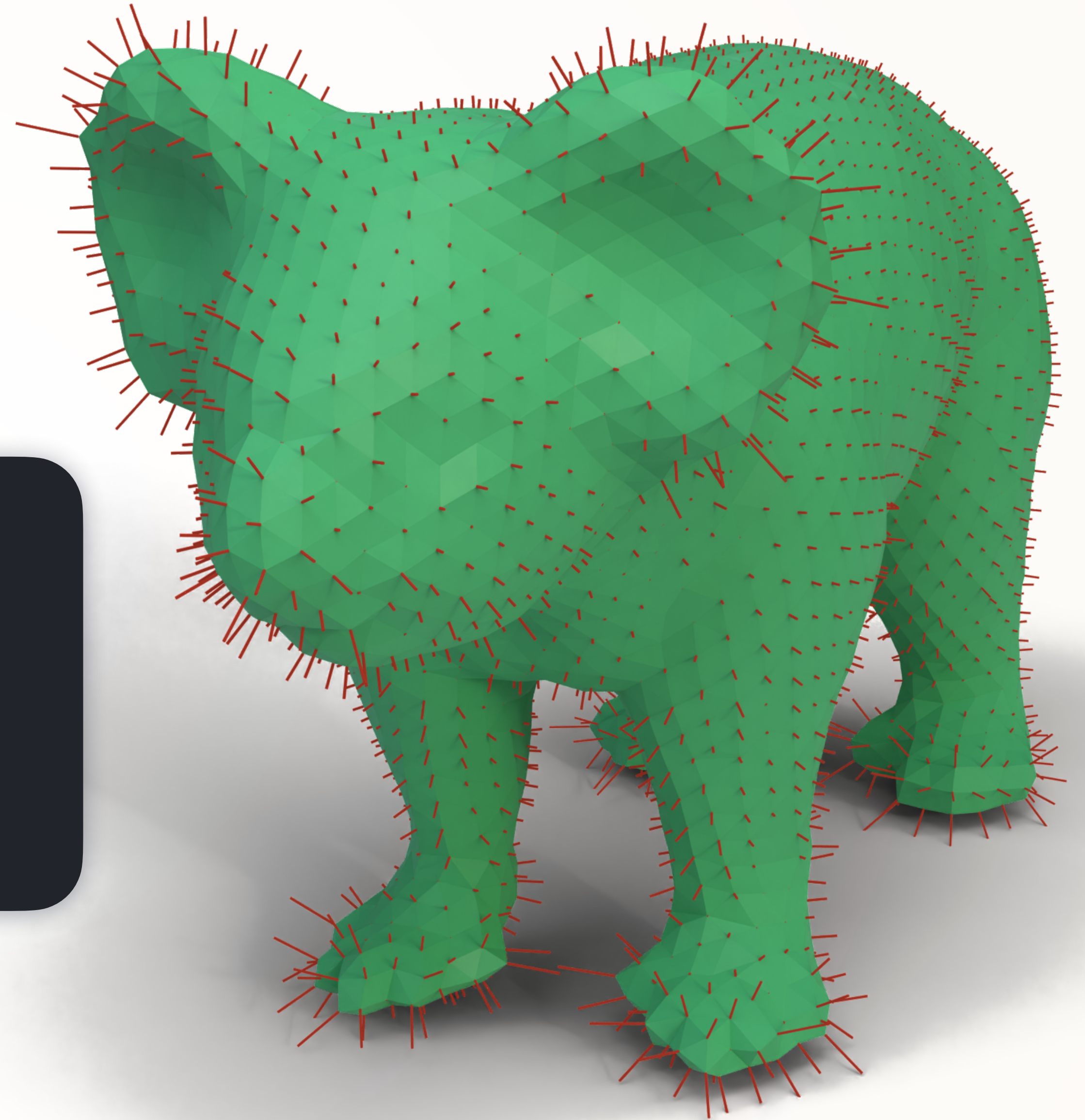
```
from iskra.dec import laplacian
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

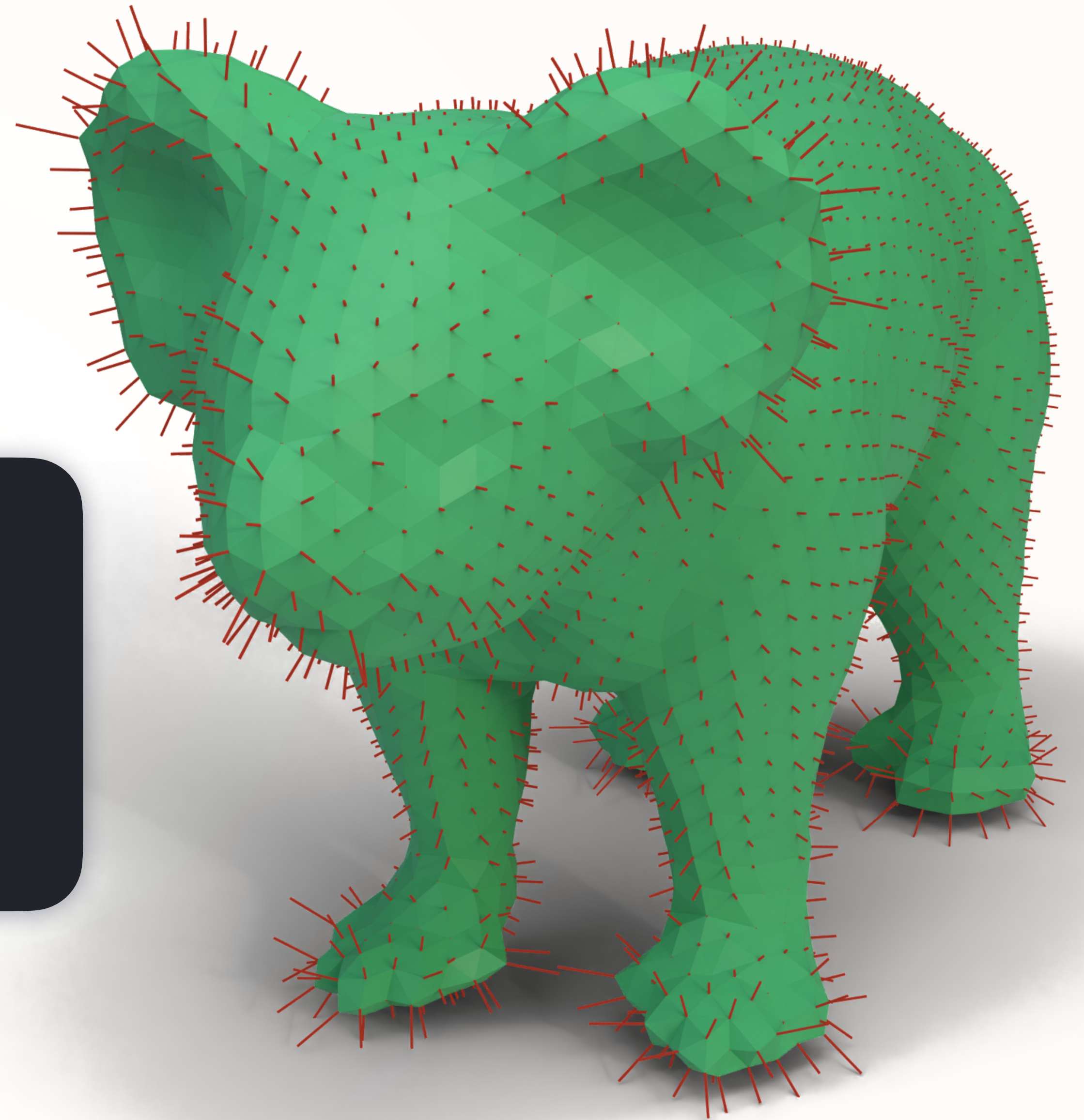
```
from iskra.dec import laplacian
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

```
from iskra.dec import laplacian  
lap, mass = laplacian(verts, faces)
```



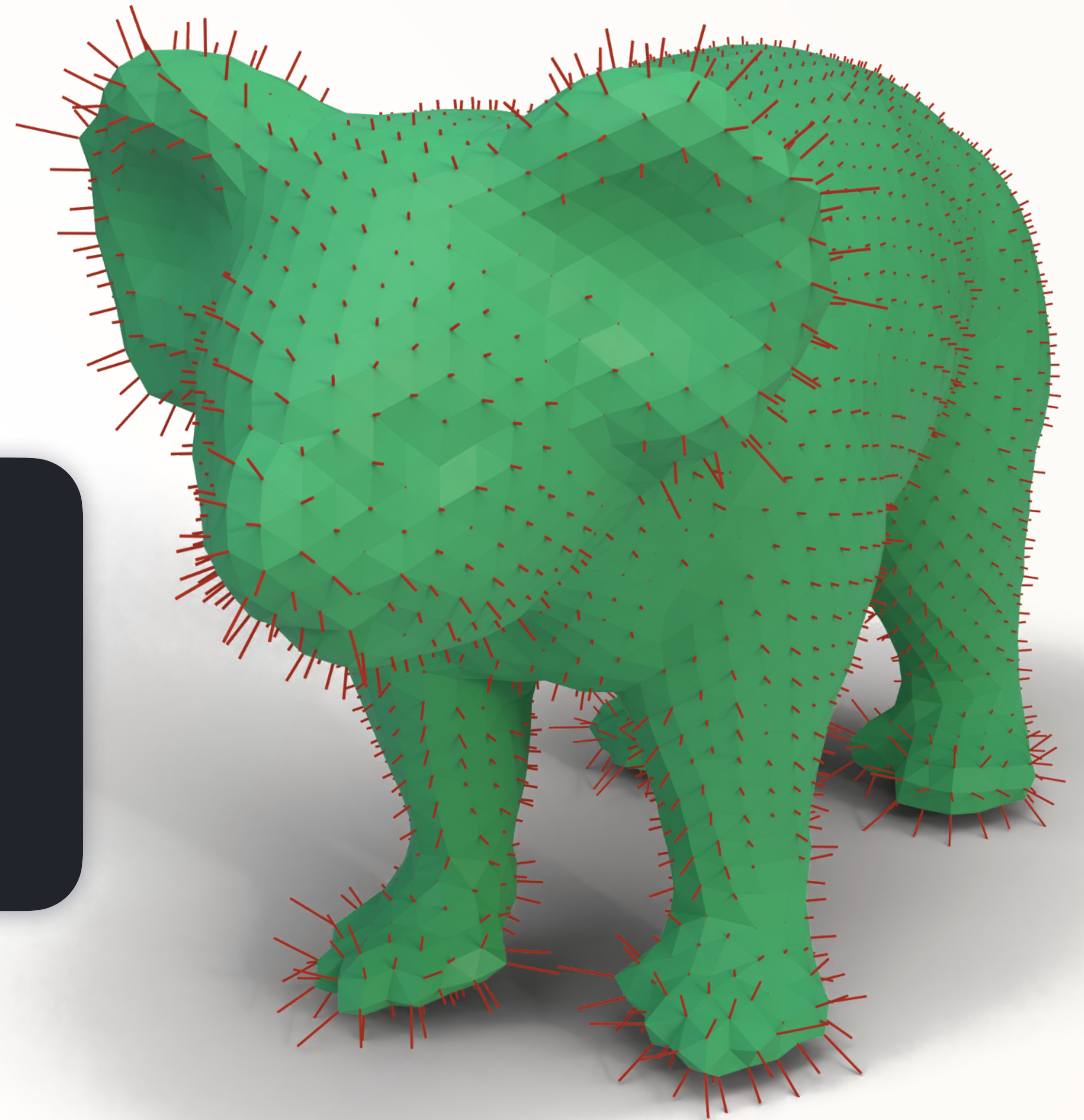
$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

```
from iskra.dec import laplacian
```

```
lap, mass = laplacian(verts, faces)
```

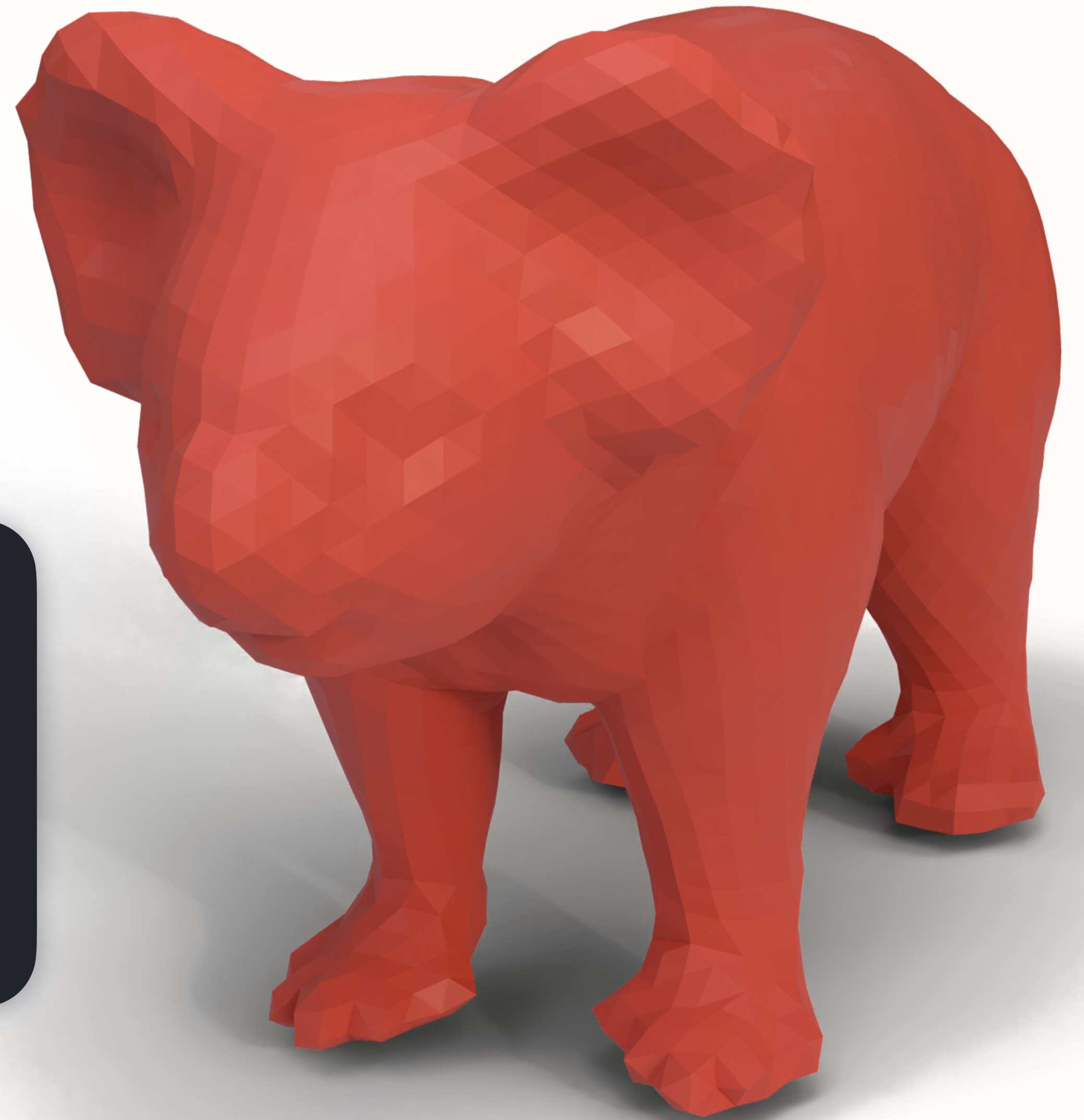
```
mcf_normals_lap = lap @ verts
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{v_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (Lv)_i$$

```
from iskra.dec import laplacian  
from iskra.sparse_linalg import linear_solve
```



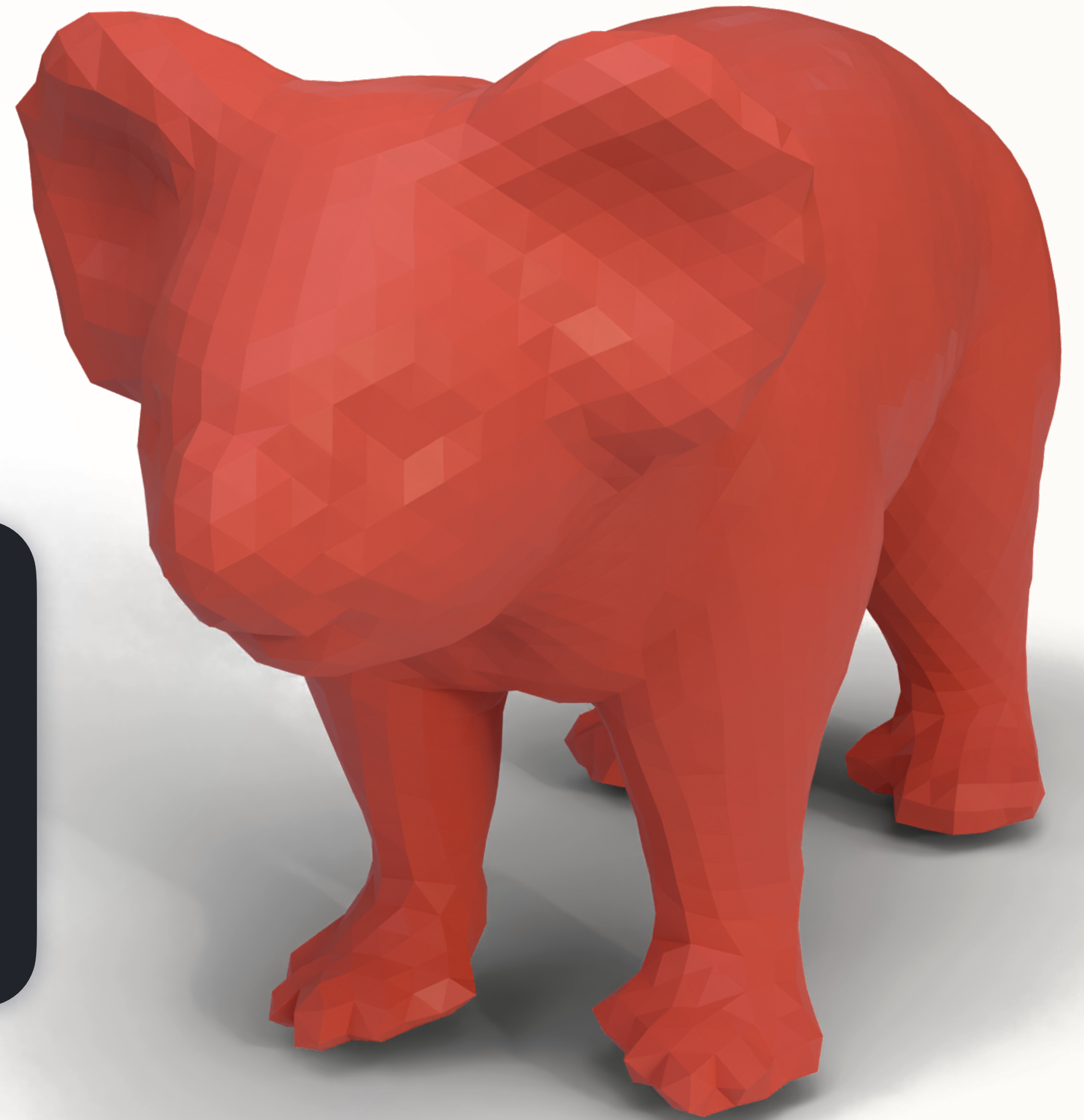
$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (L\mathbf{v})_i$$

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega)$$

$$(M + \tau L)\mathbf{v}' = M\mathbf{v}$$

```
from iskra.dec import laplacian
from iskra.sparse_linalg import linear_solve
```



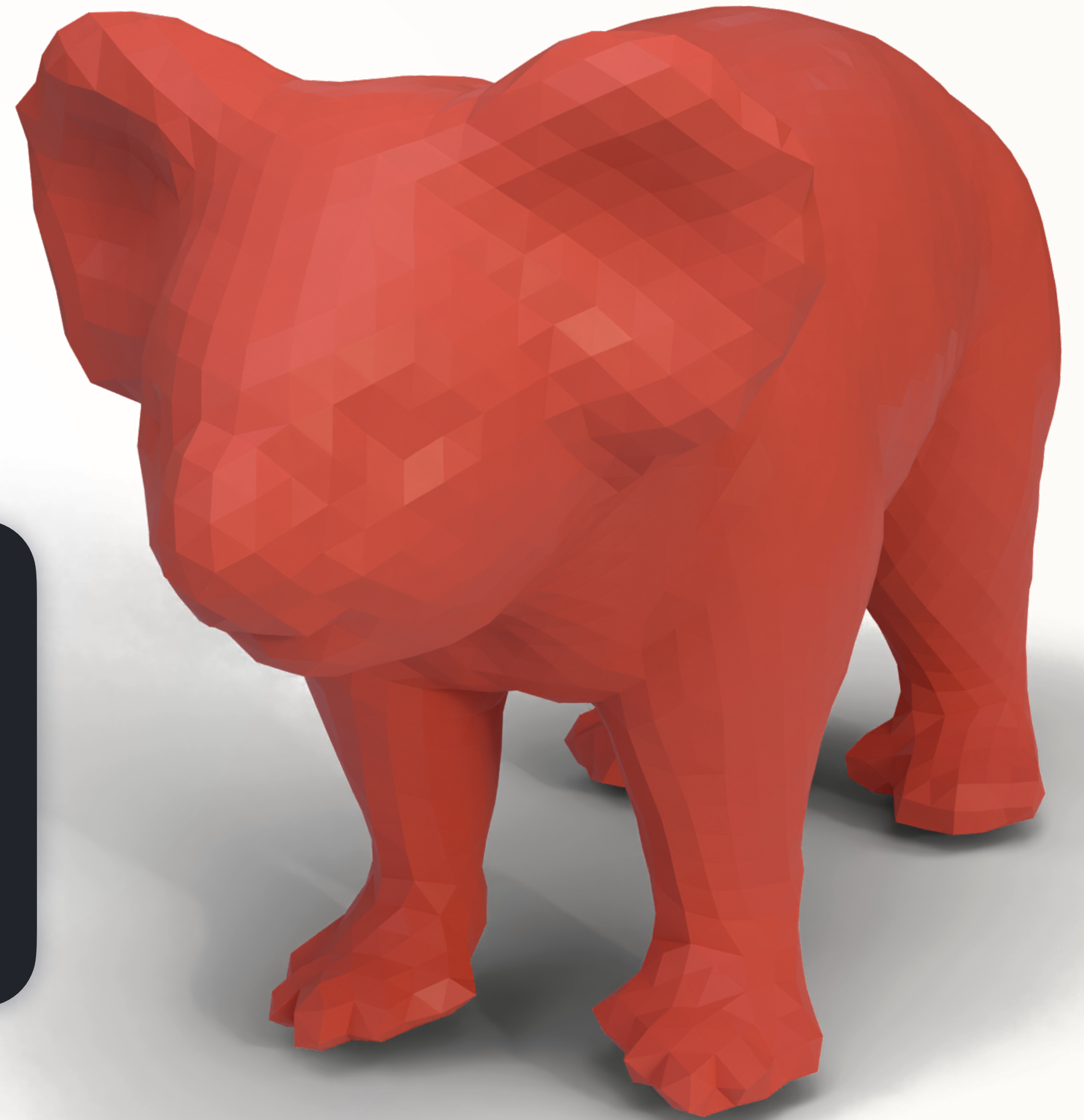
$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (L\mathbf{v})_i$$

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega)$$

$$(M + \tau L)\mathbf{v}' = M\mathbf{v}$$

```
from iskra.dec import laplacian
from iskra.sparse_linalg import linear_solve
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

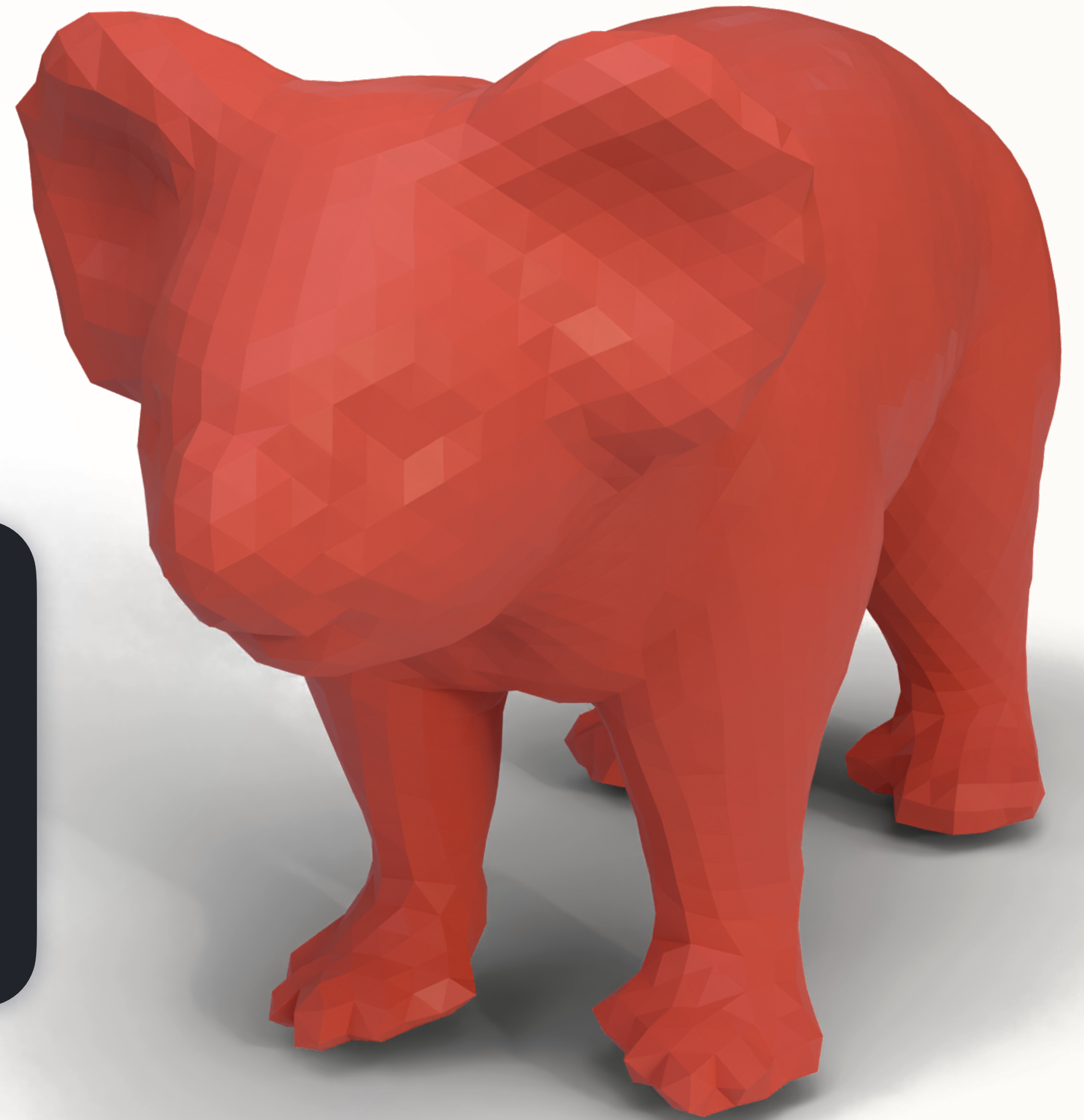
$$\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (L\mathbf{v})_i$$

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega)$$

$$(M + \tau L)\mathbf{v}' = M\mathbf{v}$$

```
from iskra.dec import laplacian
from iskra.sparse_linalg import linear_solve

lap, mass = laplacian(verts, faces)
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

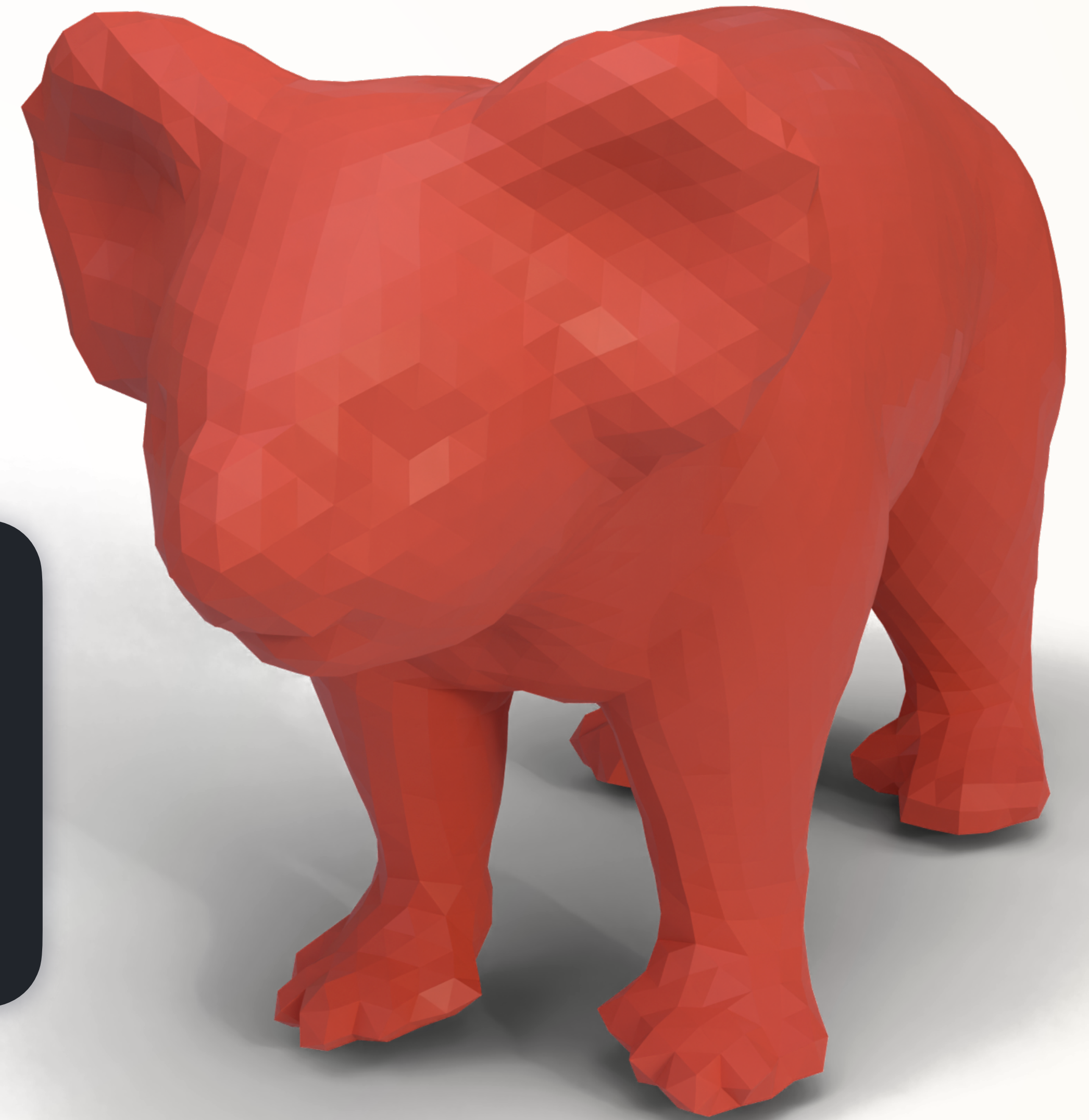
$$\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (L\mathbf{v})_i$$

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega)$$

$$(M + \tau L)\mathbf{v}' = M\mathbf{v}$$

```
from iskra.dec import laplacian
from iskra.sparse_linalg import linear_solve

lap, mass = laplacian(verts, faces)
_, smooth = linear_solve(mass + t * lap, mass @ verts)
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

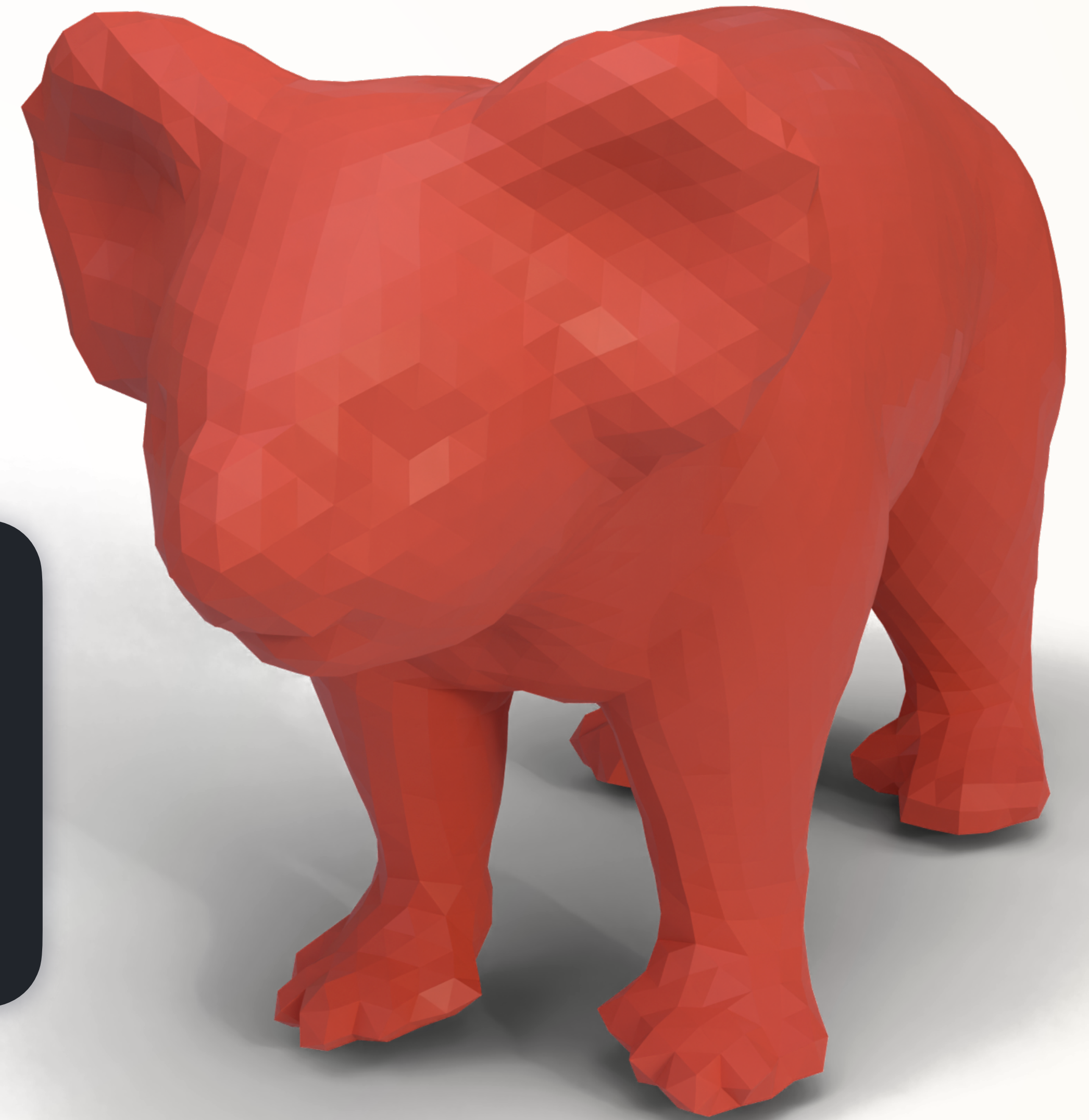
$$\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (L\mathbf{v})_i$$

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega)$$

$$(M + \tau L)\mathbf{v}' = M\mathbf{v}$$

```
from iskra.dec import laplacian
from iskra.sparse_linalg import linear_solve

lap, mass = laplacian(verts, faces)
_, smooth = linear_solve(mass + t * lap, mass @ verts)
```



$$\mathcal{E}(\Omega) = \sum_{t \in \mathbb{T}} A_t$$

$$\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega) = 2H_i \mathbf{n}_i = (L\mathbf{v})_i$$

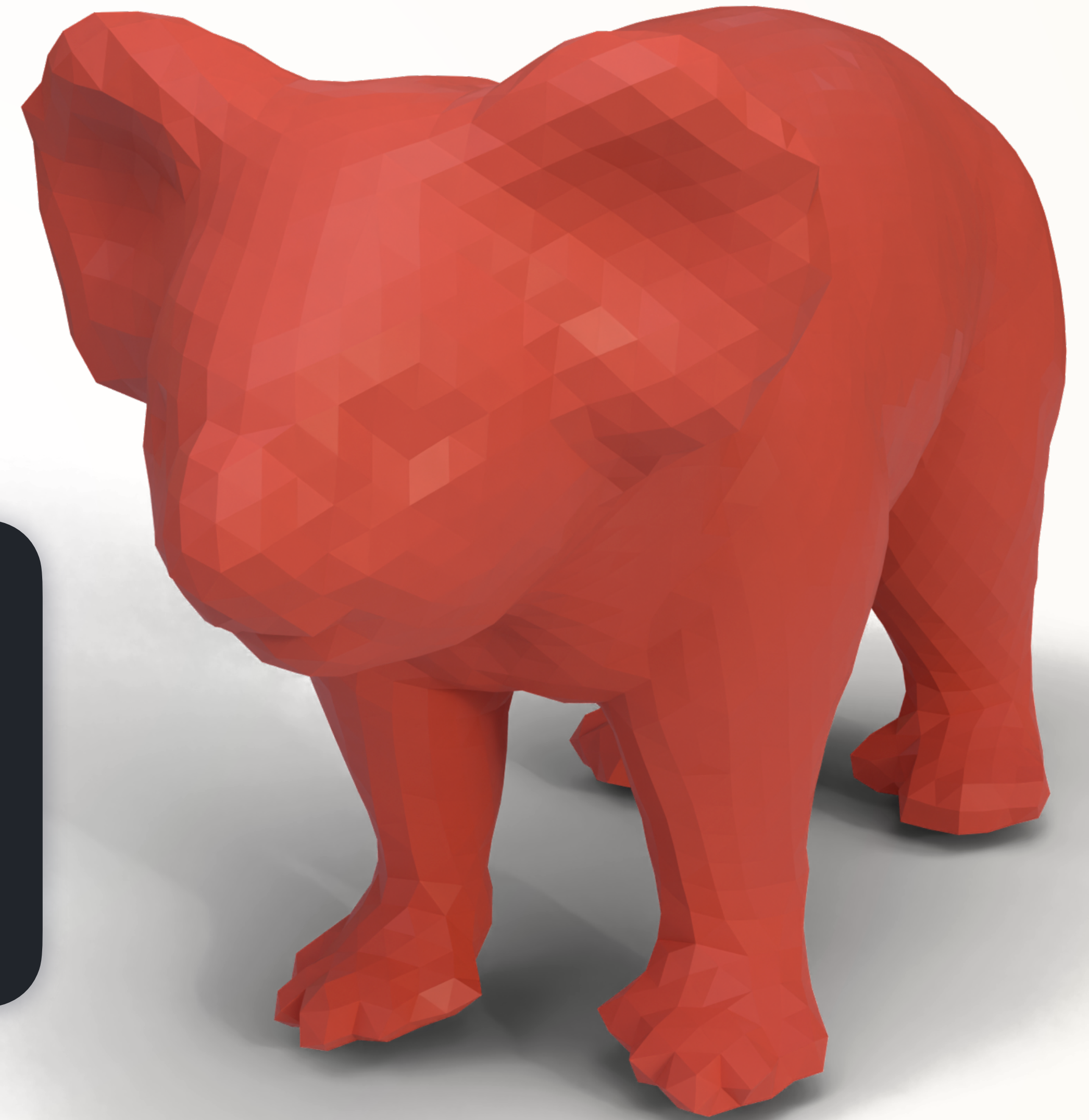
$$\frac{\partial \mathbf{v}_i}{\partial t} = -\nabla_{\mathbf{v}_i} \mathcal{E}(\Omega)$$

$$(M + \tau L)\mathbf{v}' = M\mathbf{v}$$

```
from iskra.dec import laplacian
from iskra.sparse_linalg import linear_solve

lap, mass = laplacian(verts, faces)
_, smooth = linear_solve(mass + t * lap, mass @ verts)
```

How do we differentiate
through a linear solver???



5 MINUTE BREAK

- **Chapter 1:** Introduction to Automatic Differentiation
- **Chapter 2:** Differentiable Geometry Processing in Python
- **Chapter 3:** Introduction to The Adjoint Method
- **Chapter 4:** Inverse Geometry Processing in Python

Introduction to The Adjoint Method

Chapter 3

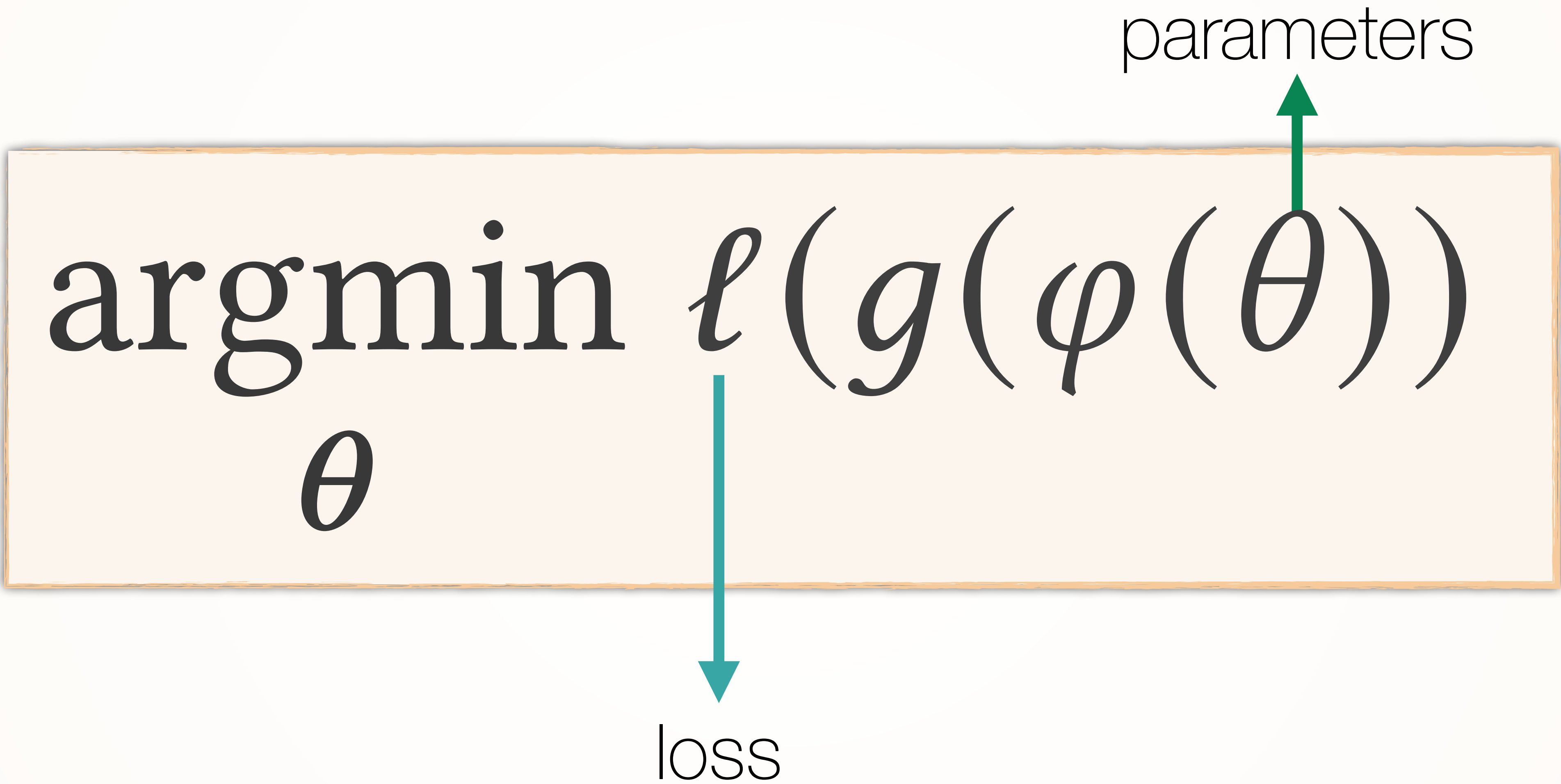
$$\operatorname{argmin}_{\theta} \ell(g(\varphi(\theta)))$$

Optimization Problem

$$\operatorname{argmin}_{\theta} \ell(g(\varphi(\theta)))$$

↓
loss

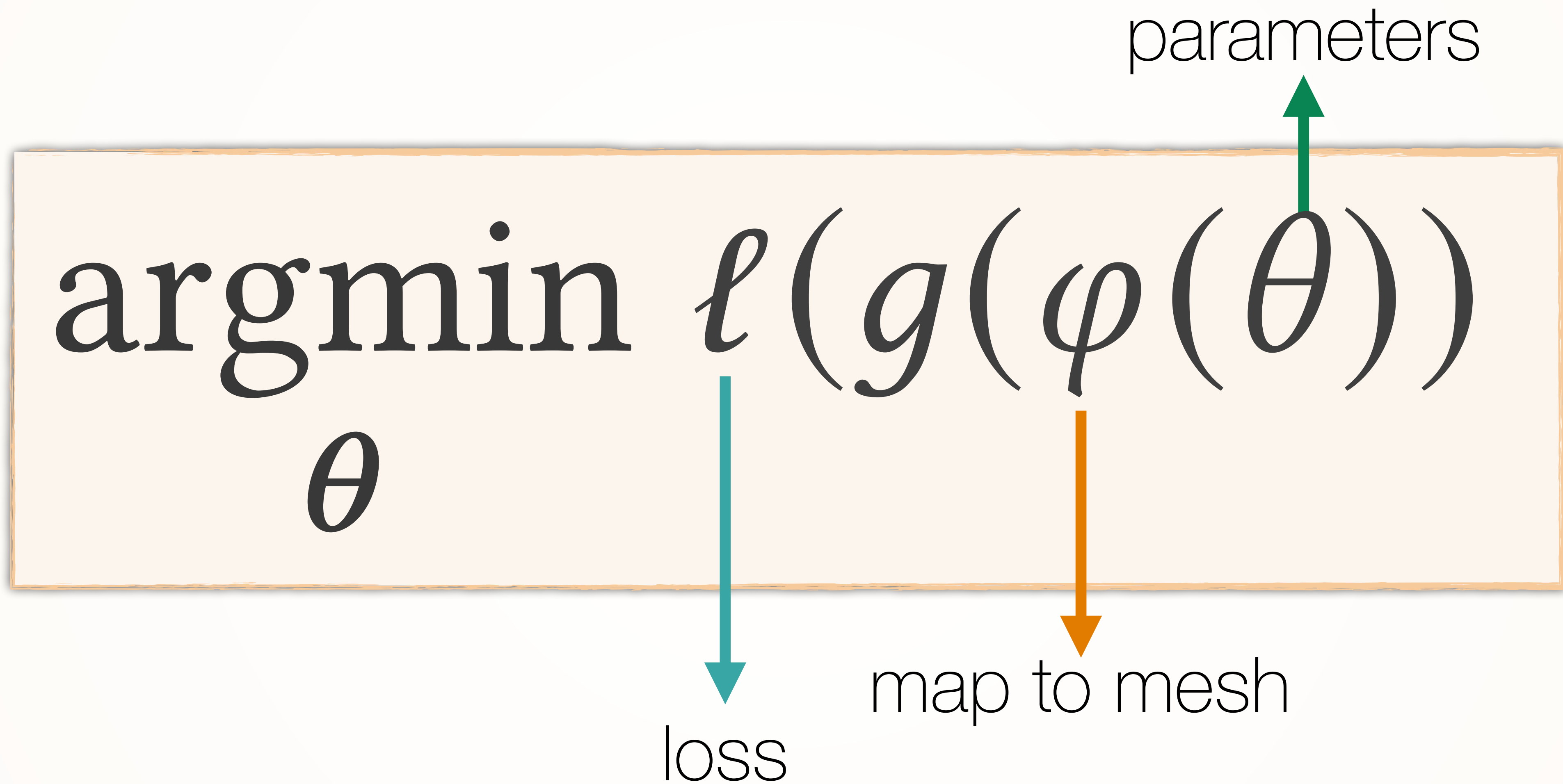
Optimization Problem

$$\operatorname{argmin}_{\theta} \ell(g(\varphi(\theta)))$$


loss

parameters

Optimization Problem



Optimization Problem

geometry processing
optimization problem

parameters

$$\operatorname{argmin}_{\theta} \ell(g(\varphi(\theta)))$$

loss

map to mesh

Optimization Problem

geometry processing
optimization problem

parameters

$$\operatorname{argmin}_{\theta} \ell(g(\varphi(\theta)))$$

loss

map to mesh

Optimization Problem

geometry processing
optimization problem

parameters

$$\operatorname{argmin}_{\theta} \ell(g(\varphi(\theta)))$$

The diagram illustrates the optimization problem $\operatorname{argmin}_{\theta} \ell(g(\varphi(\theta)))$. It features a light orange rectangular box containing the equation. Four arrows point from different parts of the equation to descriptive labels: a teal arrow points from the loss function ℓ to the label 'loss'; an orange arrow points from the map function φ to the label 'map to mesh'; a red arrow points from the geometry processing function g to the label 'geometry processing optimization problem'; and a green arrow points from the parameters θ to the label 'parameters'.

loss

map to mesh

Optimization Problem

geometry processing
optimization problem

parameters

$$\operatorname{argmin}_{\theta} \ell(\underbrace{g(\varphi(\theta))}_y)$$

The diagram shows the equation $\operatorname{argmin}_{\theta} \ell(g(\varphi(\theta)))$ inside a light orange rectangular box. A red arrow points from the $g(\varphi(\theta))$ term to the text 'geometry processing optimization problem' above the box. A green arrow points from the θ term to the text 'parameters' above the box. A teal arrow points from the ℓ term to the text 'loss' below the box. An orange arrow points from the y label (which is under a brace covering $g(\varphi(\theta))$) to the text 'map to mesh' below the box.

loss

map to mesh

Optimization Problem

$$\operatorname{argmin}_{\theta} \ell(\underbrace{g(\varphi(\theta))}_y)$$

Optimization Problem

$$\operatorname{argmin}_{\theta} \ell(\underbrace{g(\varphi(\theta))}_y)$$

need

$$\frac{\partial \ell}{\partial \theta} = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial \varphi} \frac{\partial \varphi}{\partial \theta}$$

Optimization Problem

Have: $y \leftarrow g(x)$

Want: $\frac{\partial y}{\partial x}$

Know: $f(x, y) = 0$

Implicit Differentiation

$$f(x, y) = 0$$

$$f(x, y(x)) = 0$$

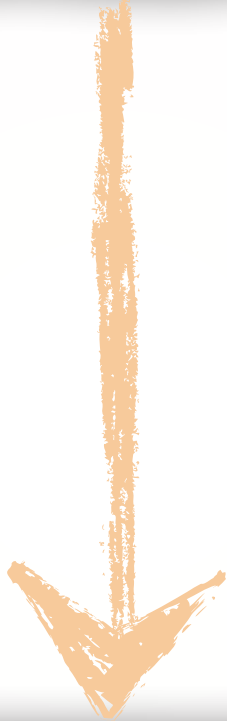
$$f(x, y(x)) = 0$$

$$f(x, y(x)) = 0$$

$$\frac{d}{dx}$$



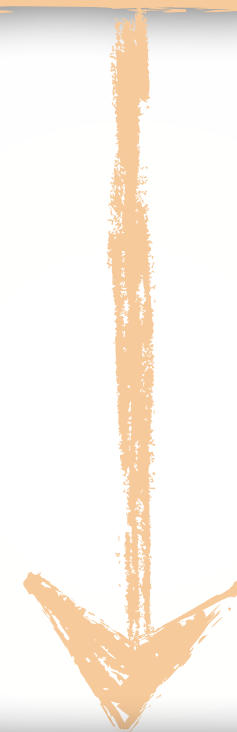
$$f(x, y(x)) = 0$$

$$\frac{d}{dx}$$


$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial x} = 0$$

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial x} = 0$$

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial x} = 0$$



$$\frac{\partial y}{\partial x} = - \left[\frac{\partial f}{\partial y} \right]^{-1} \frac{\partial f}{\partial x}$$

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial x} = 0$$

You specify: $f(x, y)$

You obtain: $\frac{\partial y}{\partial x}$

$$\frac{\partial f}{\partial x} + \left[\frac{\partial f}{\partial y} \right] \frac{\partial y}{\partial x} = 0$$

Have: $y \leftarrow g(x)$

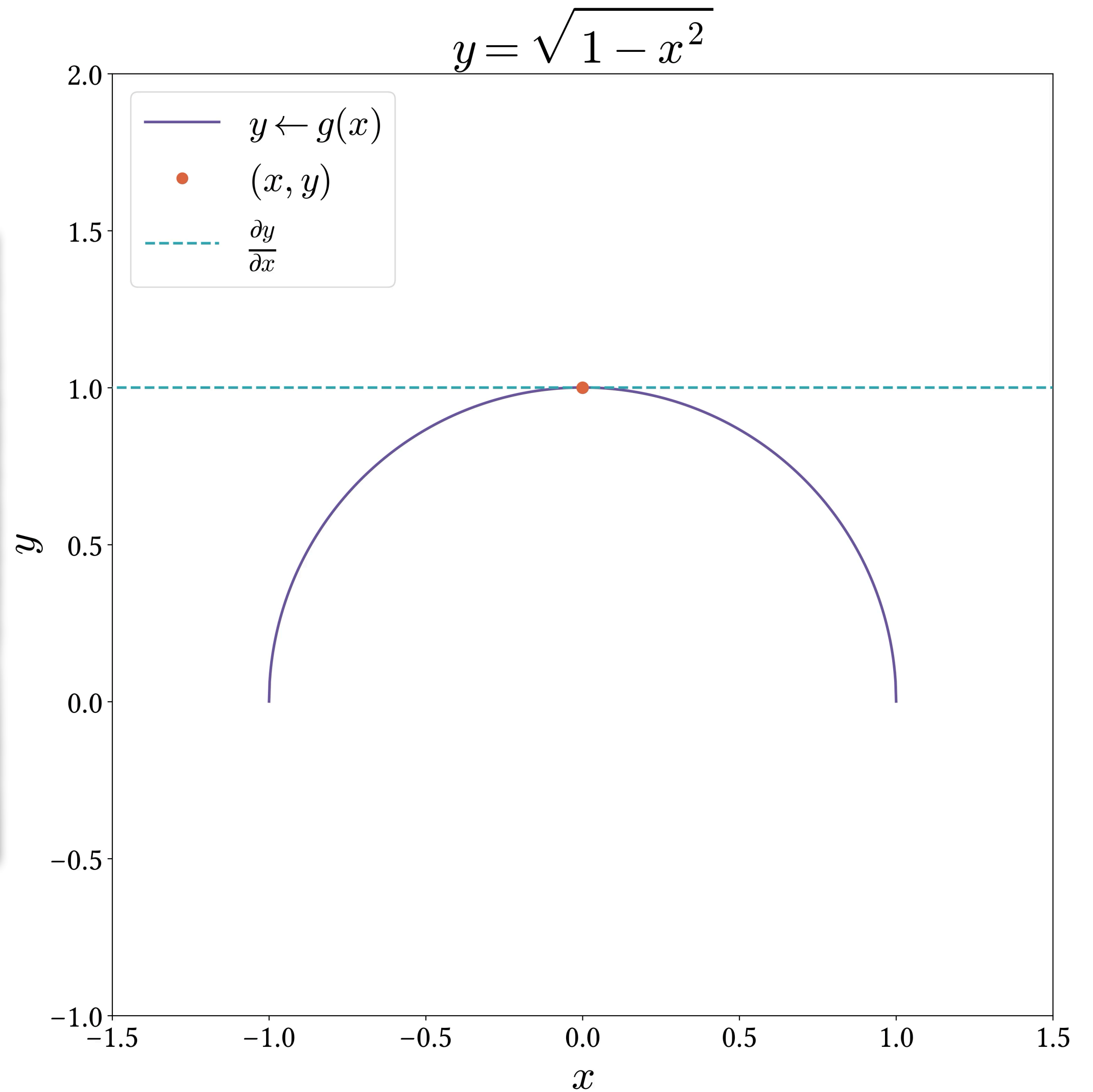
Know: $f(x, y) = 0$

Want: $\frac{\partial y}{\partial x}$

Have: $y \leftarrow g(x)$

Know: $f(x, y) = 0$

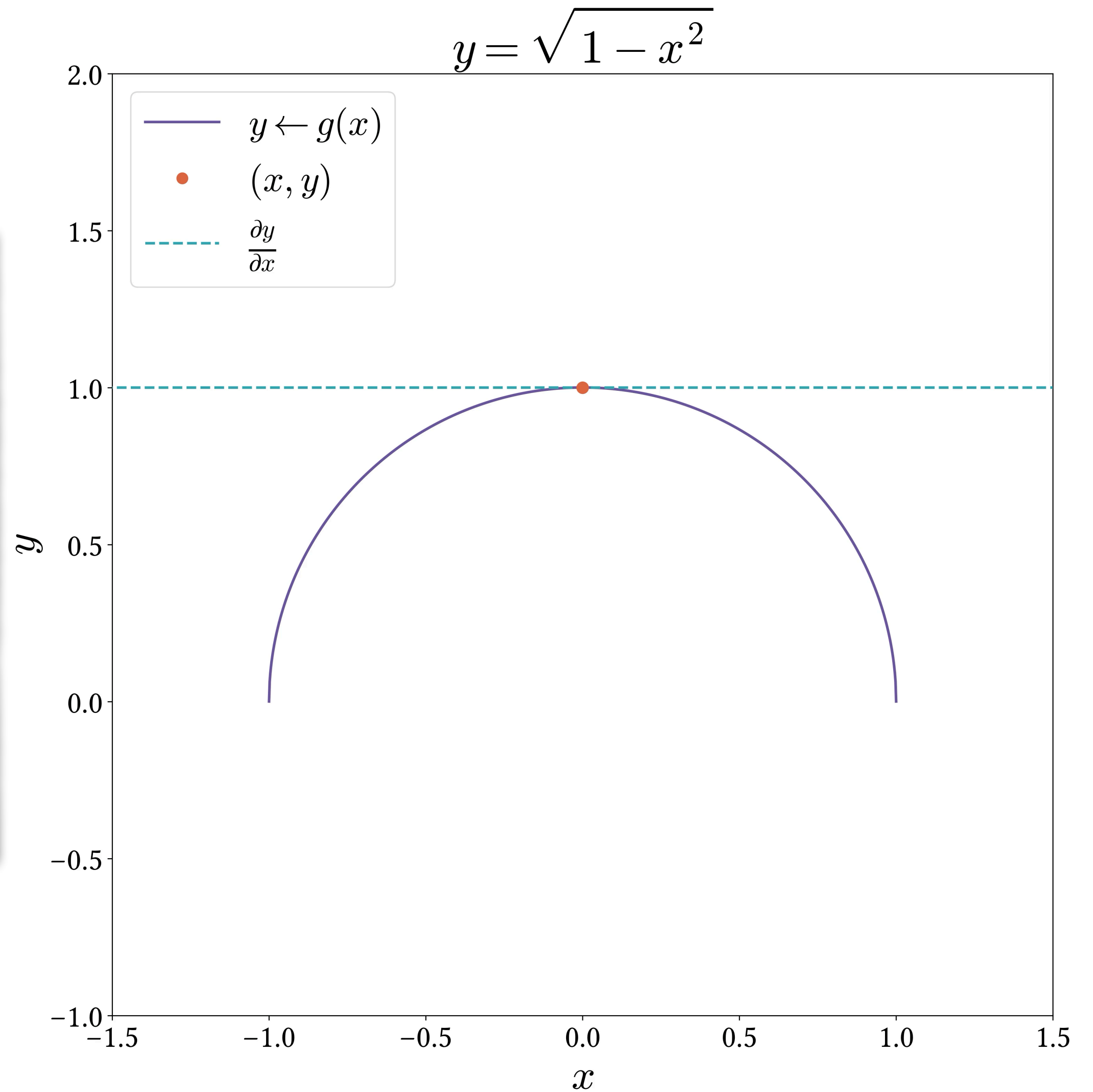
Want: $\frac{\partial y}{\partial x}$



Have: $y \leftarrow g(x) := \sqrt{1 - x^2}$

Know: $f(x, y) = 0$

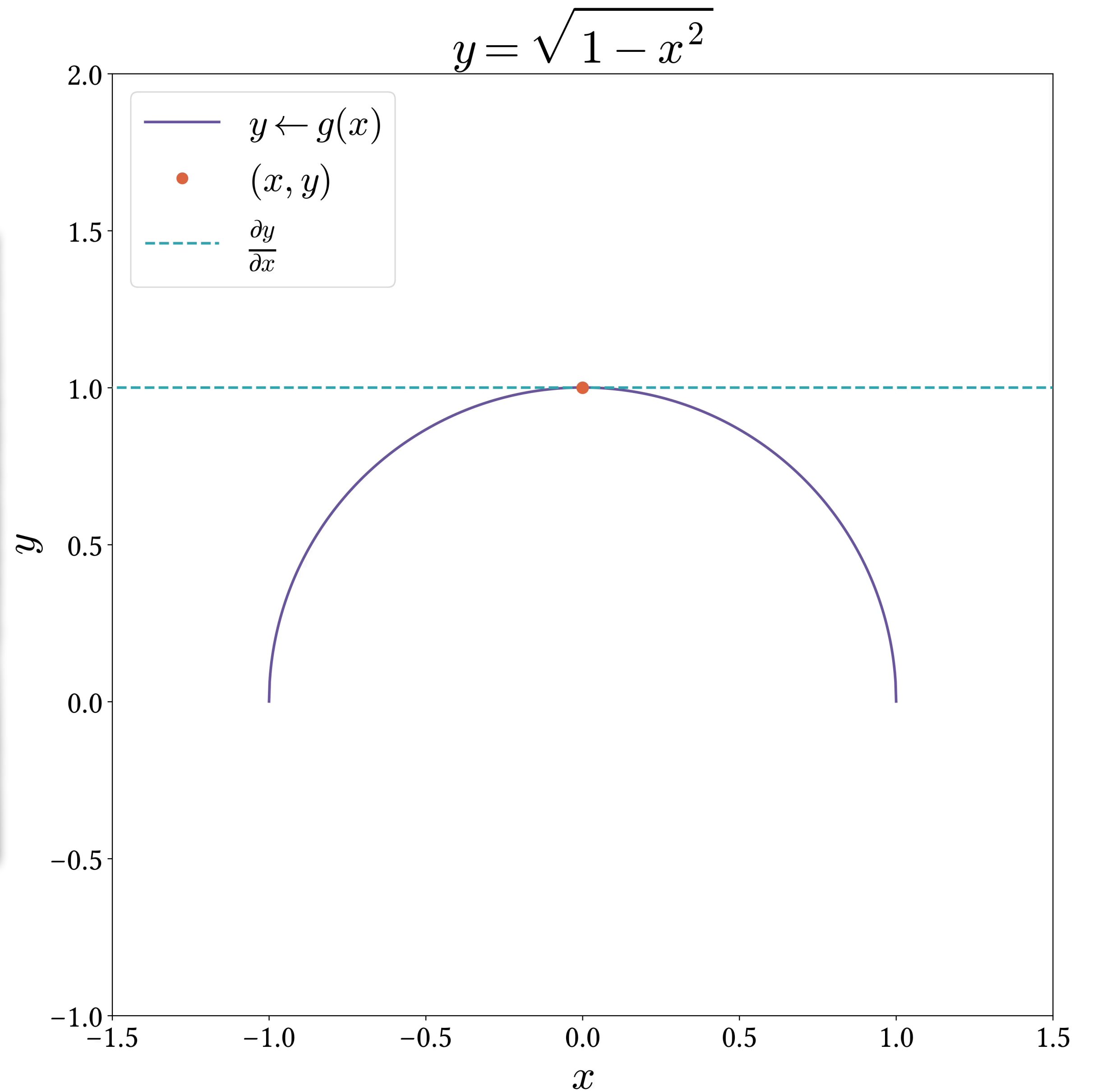
Want: $\frac{\partial y}{\partial x}$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y) = 0$

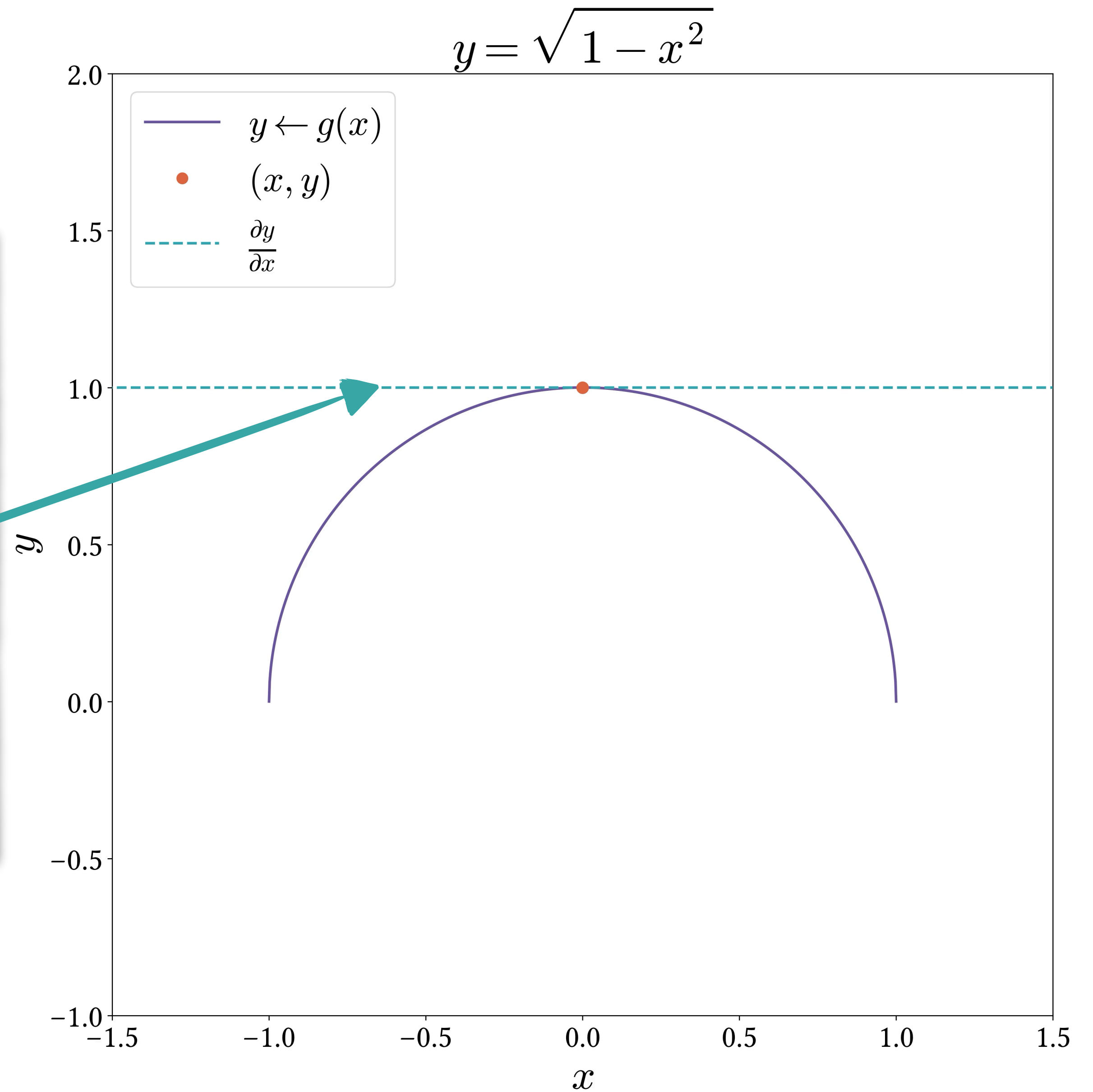
Want: $\frac{\partial y}{\partial x}$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y) = 0$

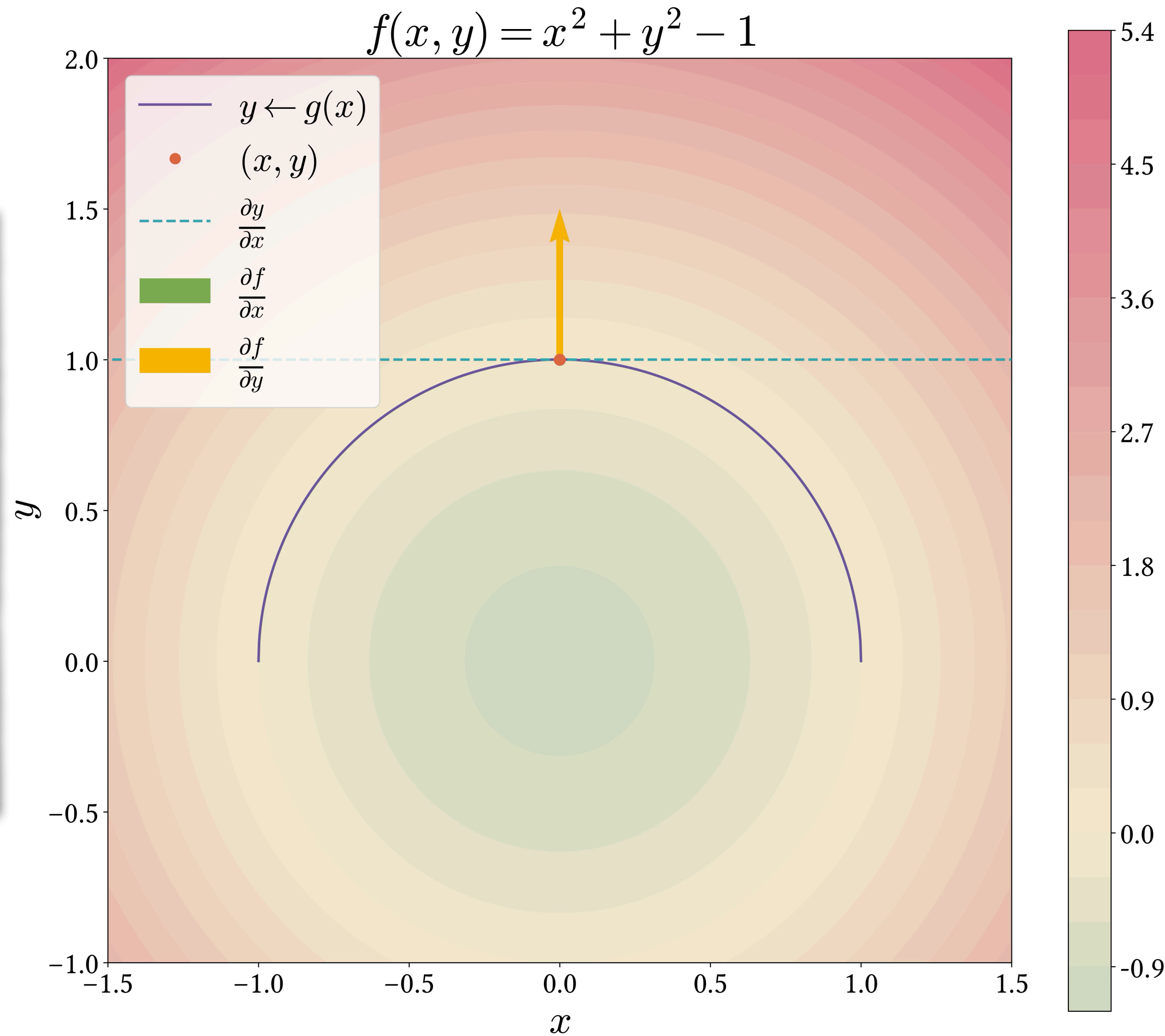
Want: $\frac{\partial y}{\partial x}$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y) = 0$

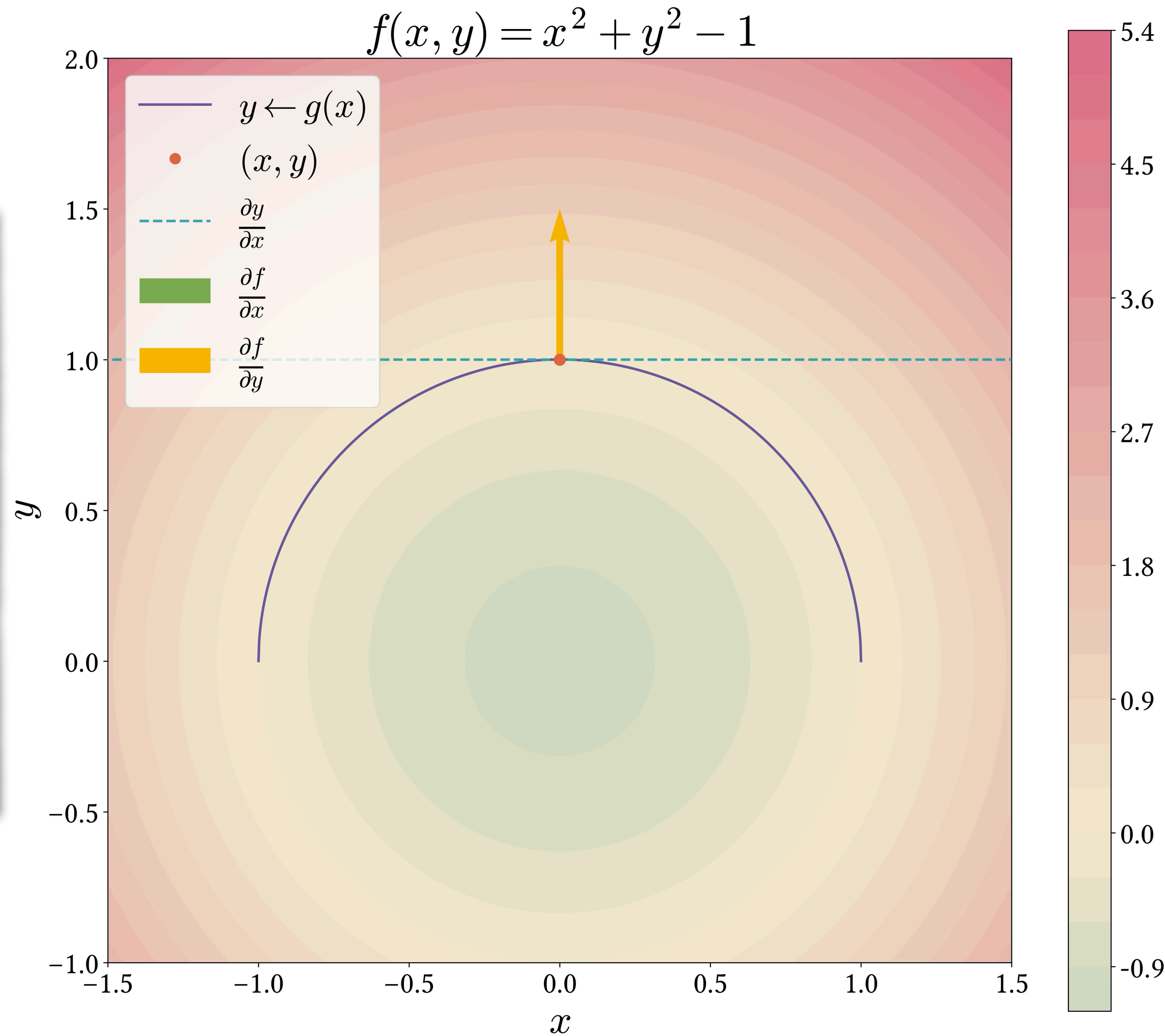
Want: $\frac{\partial y}{\partial x}$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y) = x^2 + y^2 - 1 = 0$

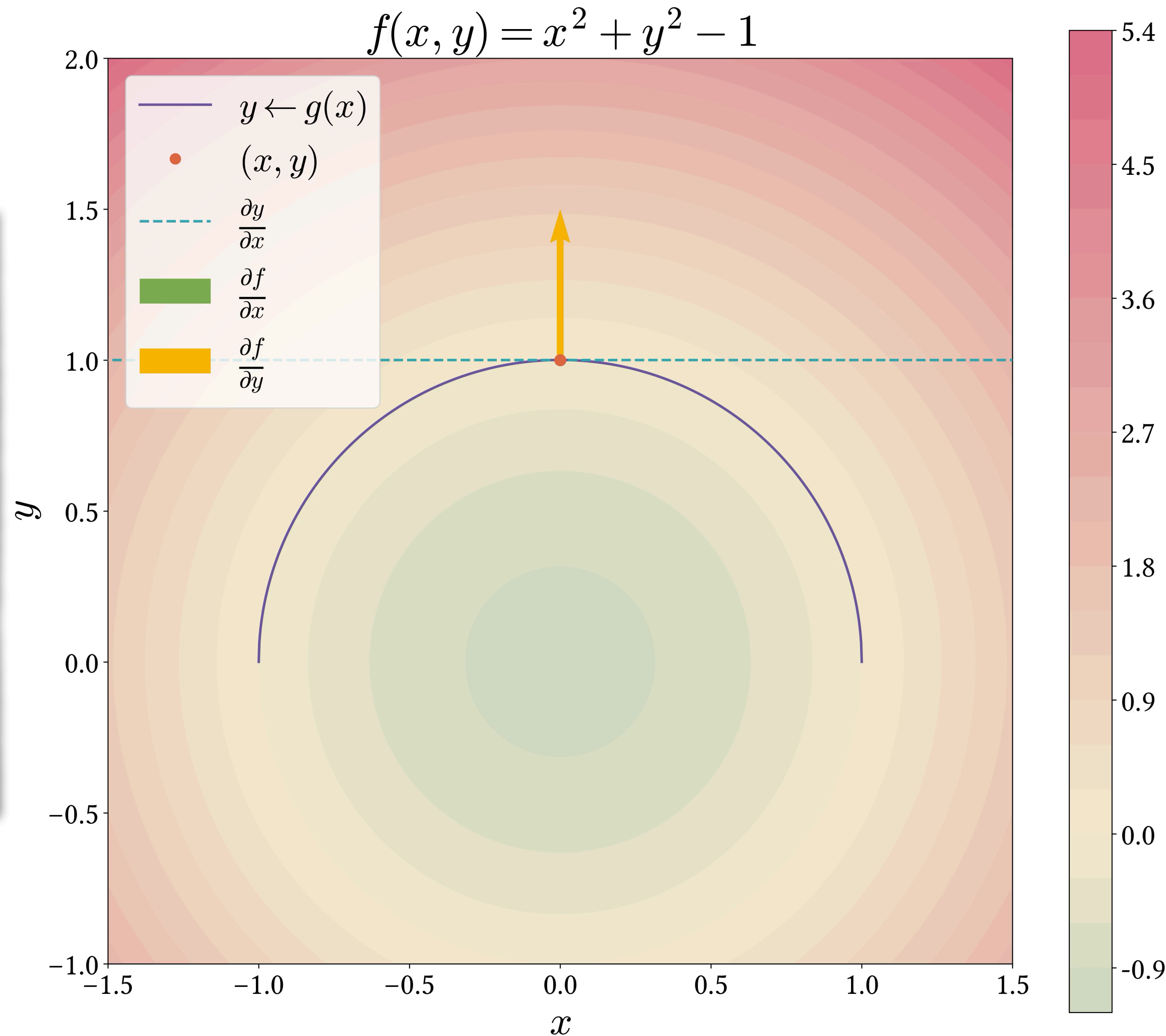
Want: $\frac{\partial y}{\partial x}$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y(x)) = x^2 + y^2 - 1 = 0$

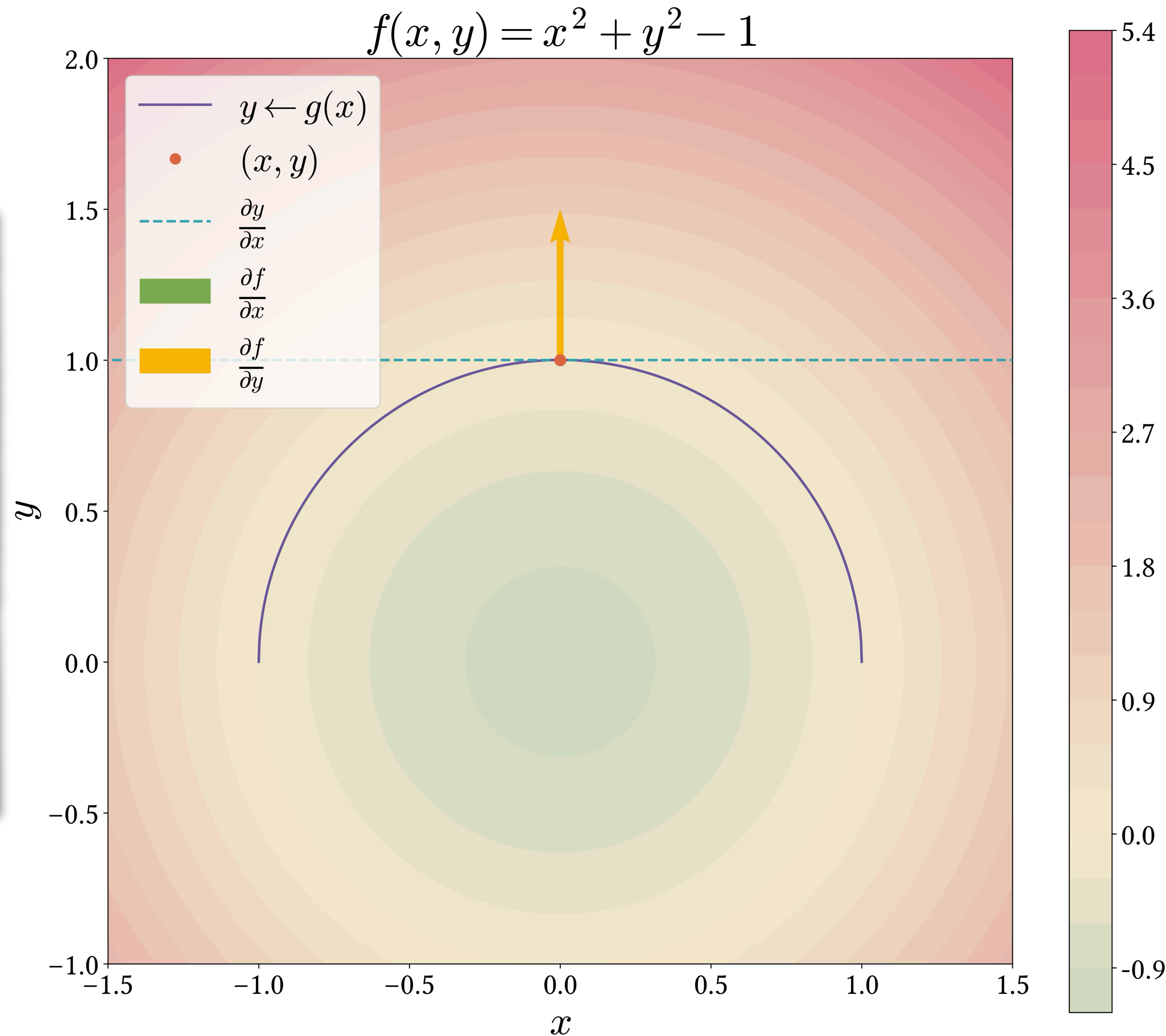
Want: $\frac{\partial y}{\partial x}$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y(x)) = x^2 + y^2 - 1 = 0$

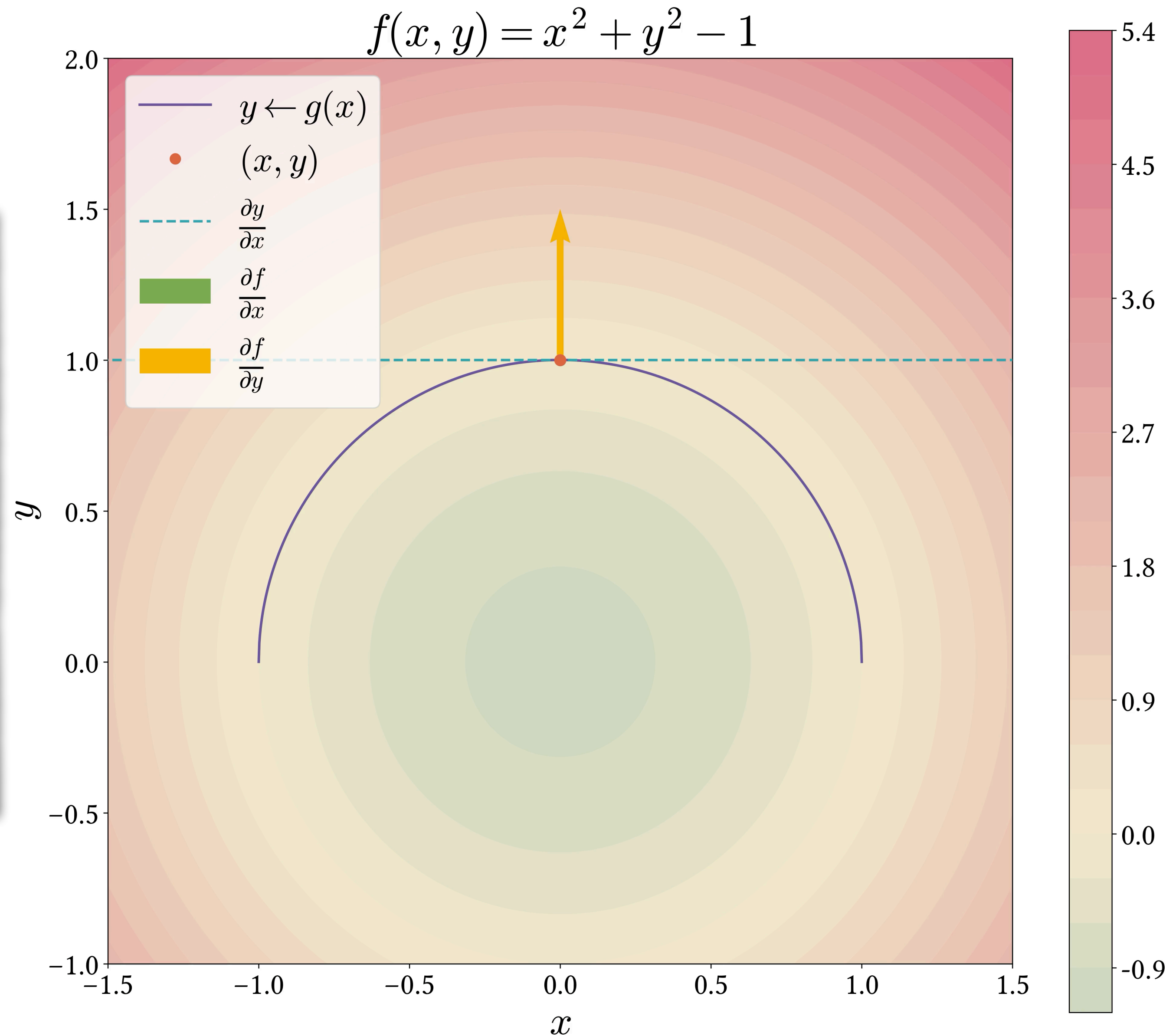
Want: $\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial x} = 0$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y(x)) = x^2 + y^2 - 1 = 0$

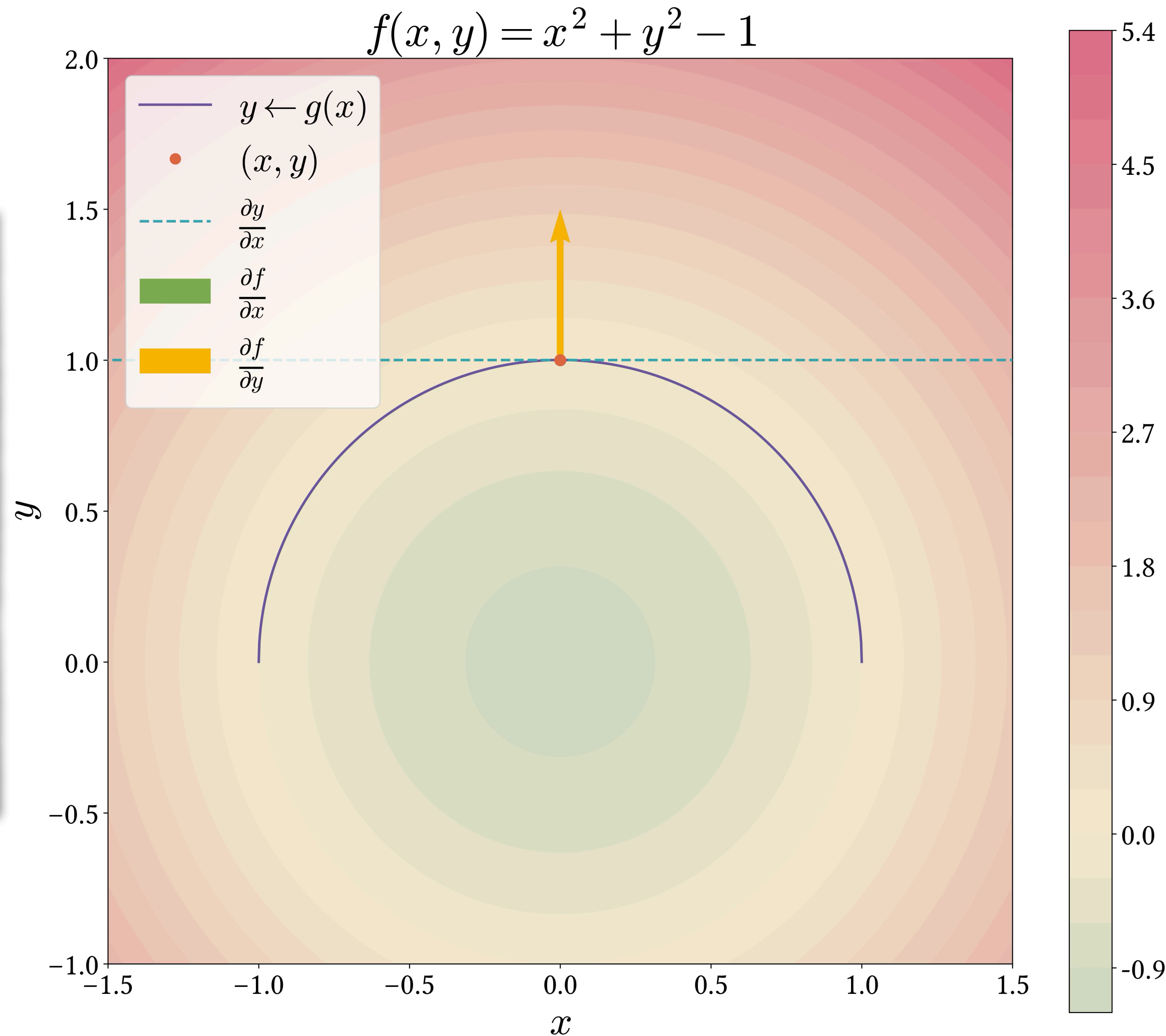
Want: $2x + 2y \frac{\partial y}{\partial x} = 0$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y(x)) = x^2 + y^2 - 1 = 0$

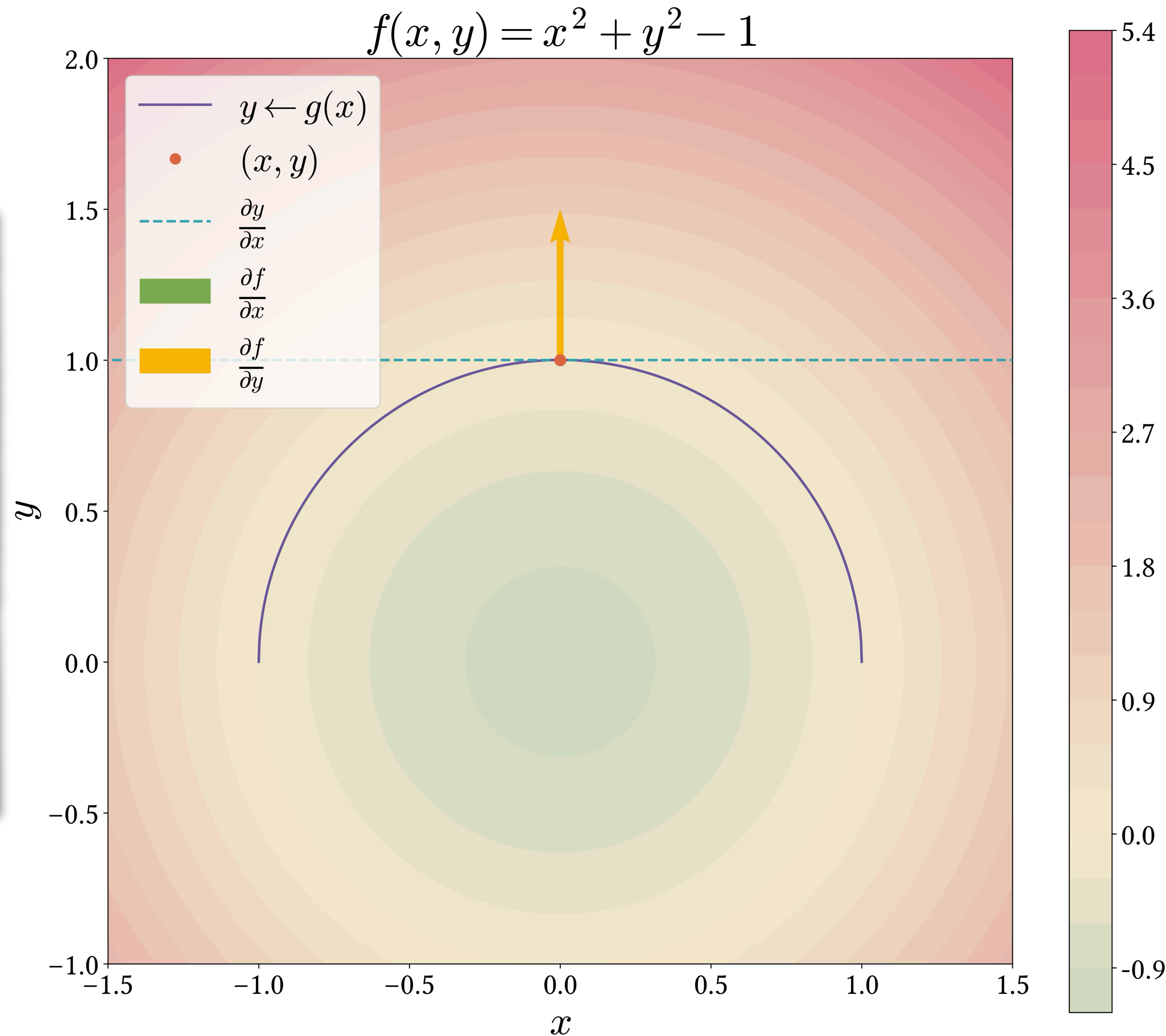
Want: $\frac{\partial y}{\partial x} = -y^{-1}x$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y(x)) = x^2 + y^2 - 1 = 0$

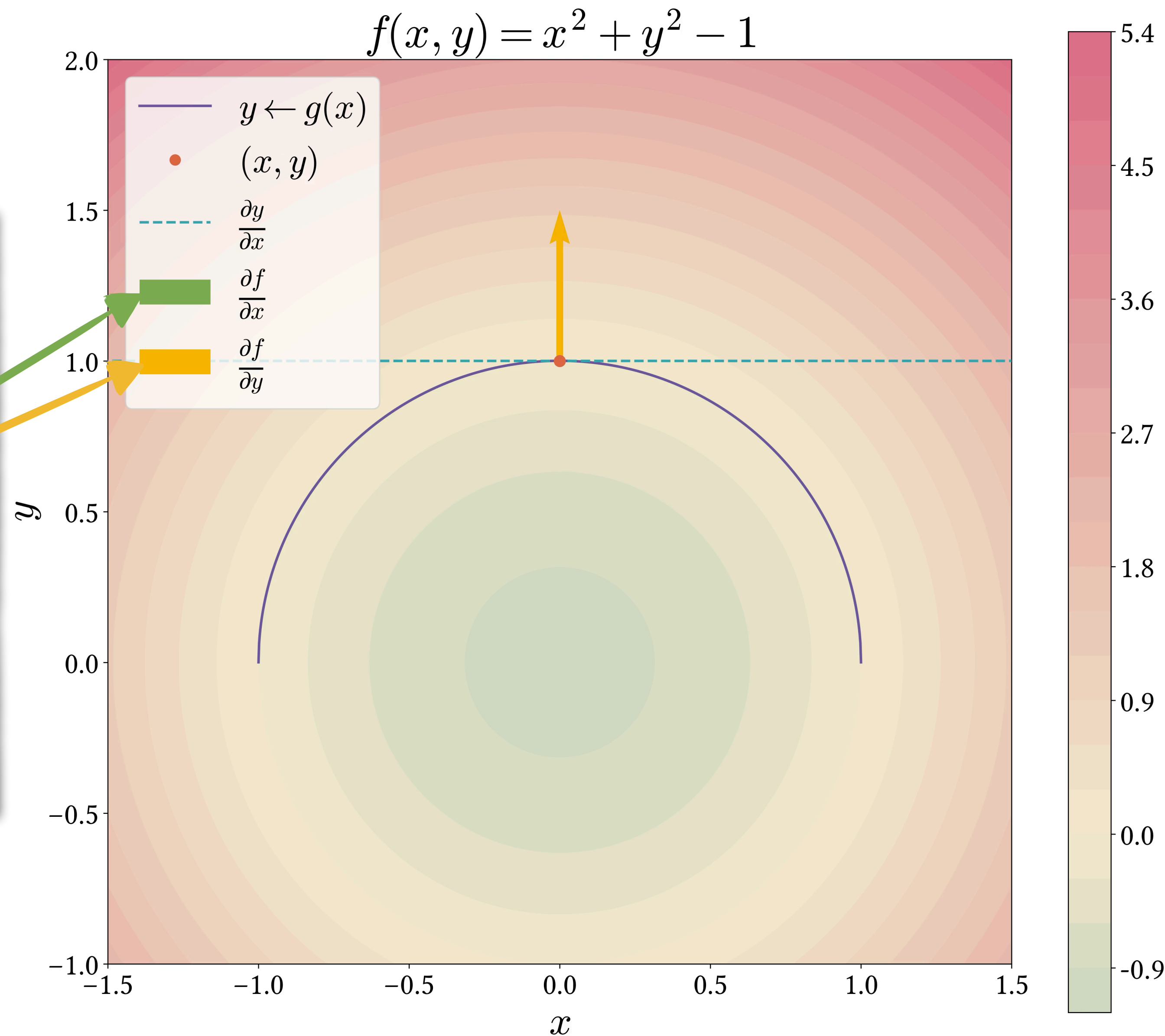
Want: $\frac{\partial y}{\partial x} = -y^{-1}x$



Have: $y \leftarrow g(x) := \text{SAFE_SQRT}(1 - x^2)$

Know: $f(x, y(x)) = x^2 + y^2 - 1 = 0$

Want: $\frac{\partial y}{\partial x} = -y^{-1}x$



Have:

$$y \leftarrow g(x) := \text{SAFE_CODE}(1 - x^2)$$

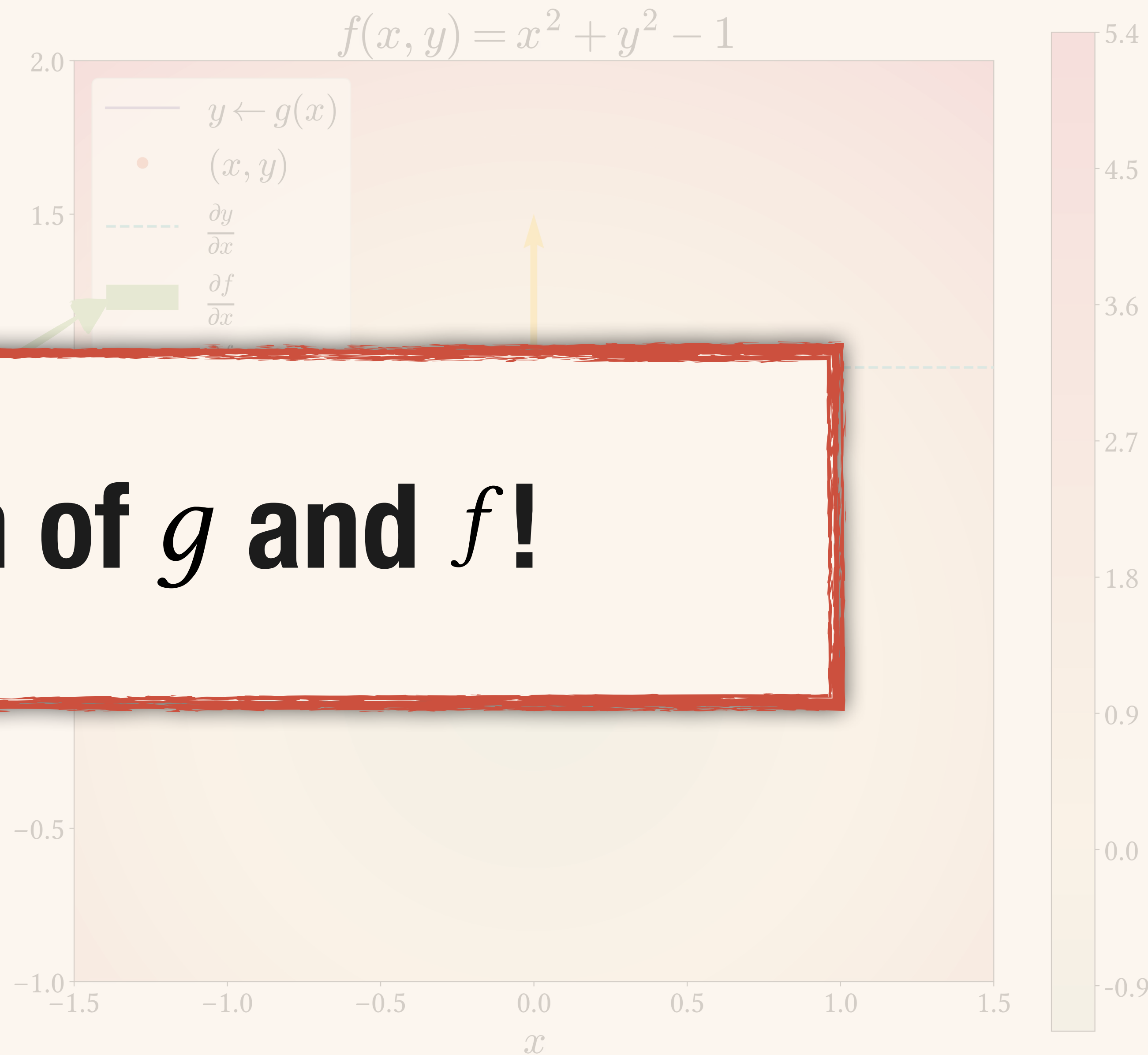
Know:

$$f$$

Want:

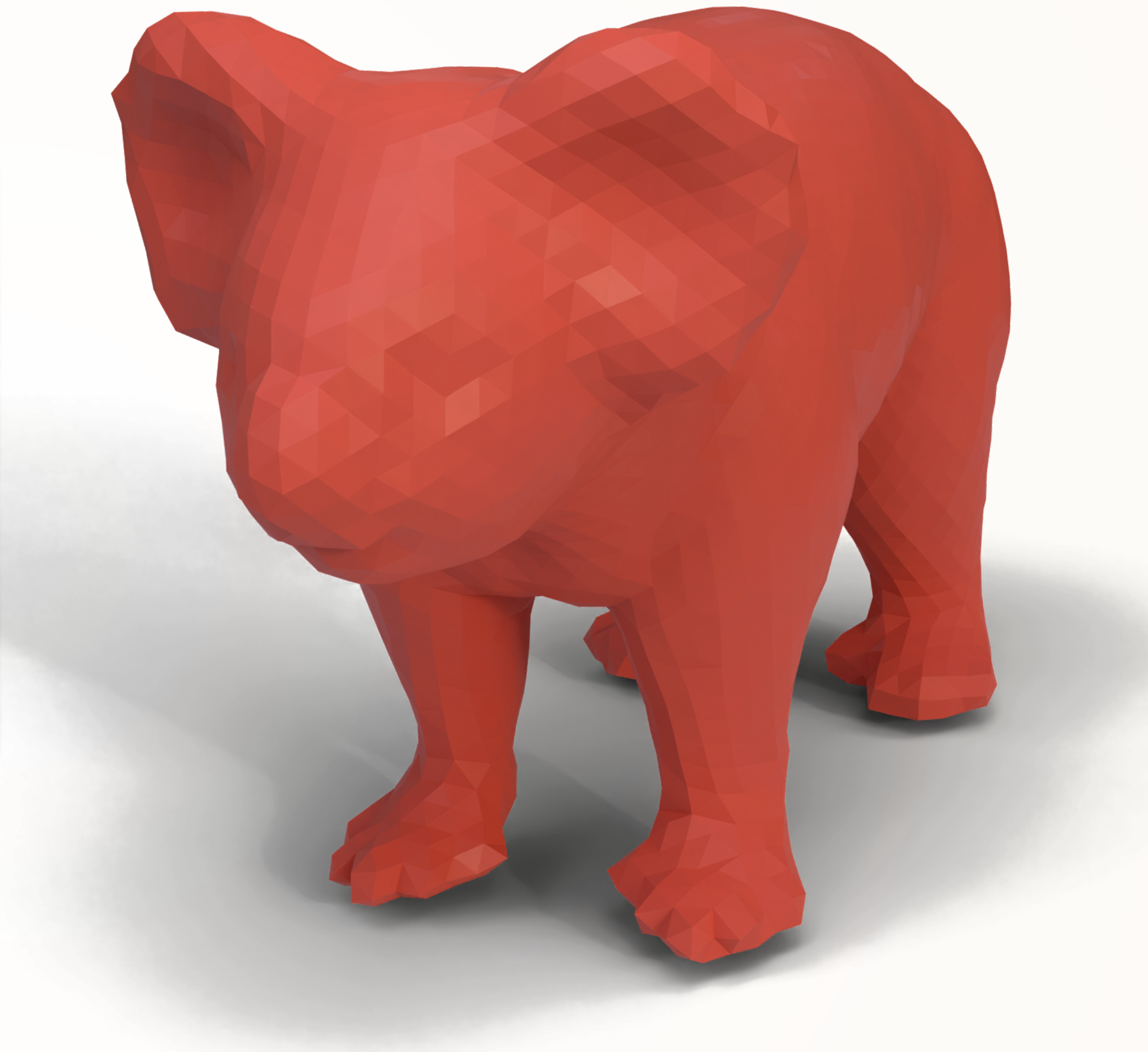
$$\frac{\partial y}{\partial x} = -y^{-1}x$$

Separation of g and f !



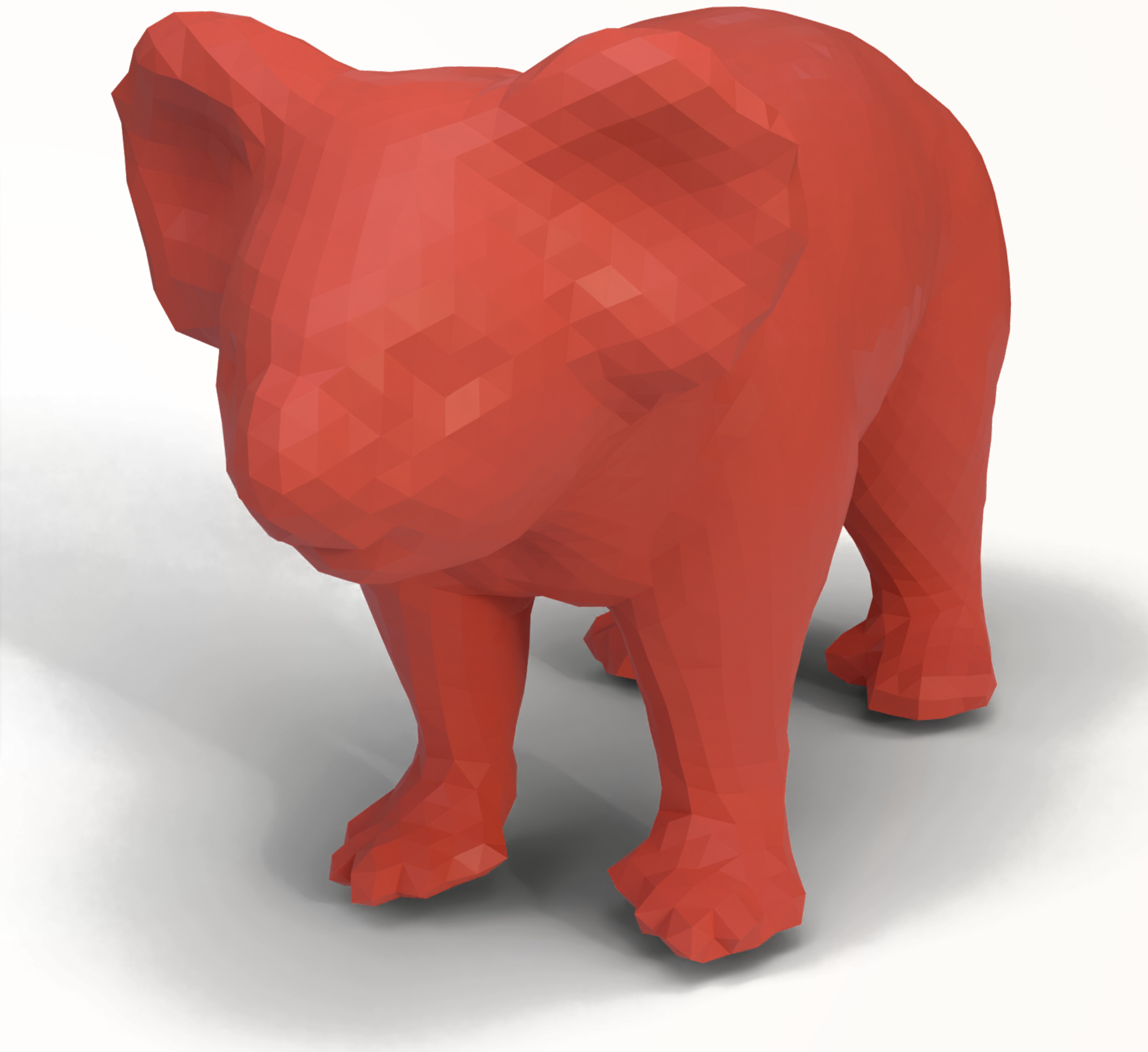
$$(M + \tau L)v' = Mv$$

$$\underbrace{(M + \tau L)}_A \underbrace{v'}_y = \underbrace{Mv}_b$$



$$(M + \tau L)v' = Mv$$

$$\underbrace{(M + \tau L)}_A \underbrace{v'}_y = \underbrace{Mv}_b$$



Have: $y \leftarrow A^{-1}b$ $x := (A, b)$

Know: $f(A, b; y) = Ay - b = 0$

Have: $y \leftarrow A^{-1}b \quad x := (A, b)$

Know: $f(A, b; y) = Ay - b = 0$

Want: $\frac{\partial y}{\partial x} = - \left[\frac{\partial f}{\partial y} \right]^{-1} \frac{\partial f}{\partial x}$

Have: $\mathbf{y} \leftarrow A^{-1}\mathbf{b} \quad \mathbf{x} := (A, \mathbf{b})$

Know: $f(A, \mathbf{b}; \mathbf{y}) = A\mathbf{y} - \mathbf{b} = 0$

Want: $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = - \left[\frac{\partial f}{\partial \mathbf{y}} \right]^{-1} \frac{\partial f}{\partial \mathbf{x}}$

Have: $\mathbf{y} \leftarrow A^{-1}\mathbf{b} \quad \mathbf{x} := (A, \mathbf{b})$

Know: $f(A, \mathbf{b}; \mathbf{y}) = A\mathbf{y} - \mathbf{b} = 0$

Want: $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = - \left[\frac{\partial f}{\partial \mathbf{y}} \right]^{-1} \frac{\partial f}{\partial \mathbf{x}}$

$$\frac{\partial f}{\partial \mathbf{y}} = A, \quad \frac{\partial f}{\partial \mathbf{b}} = -I, \quad \frac{\partial f_i}{\partial A_{jk}} = \delta_{ij} y_k,$$

Have: $\mathbf{y} \leftarrow A^{-1}\mathbf{b} \quad \mathbf{x} := (A, \mathbf{b})$

Know: $f(A, \mathbf{b}; \mathbf{y}) = A\mathbf{y} - \mathbf{b} = 0$

Want: $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = - \left[\frac{\partial f}{\partial \mathbf{y}} \right]^{-1} \frac{\partial f}{\partial \mathbf{x}}$

$$\frac{\partial f}{\partial \mathbf{y}} = A,$$

$$\frac{\partial f}{\partial \mathbf{b}} = -I,$$

$$\frac{\partial f_i}{\partial A_{jk}} = \delta_{ij} y_k,$$

Have: $\mathbf{y} \leftarrow A^{-1}\mathbf{b} \quad \mathbf{x} := (A, \mathbf{b})$

Know: $f(A, \mathbf{b}; \mathbf{y}) = A\mathbf{y} - \mathbf{b} = 0$

Want: $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = - \left[\frac{\partial f}{\partial \mathbf{y}} \right]^{-1} \frac{\partial f}{\partial \mathbf{x}}$

$$\frac{\partial f}{\partial \mathbf{y}} = A,$$

$$\frac{\partial f}{\partial \mathbf{b}} = -I,$$

$$\frac{\partial f_i}{\partial A_{jk}} = \delta_{ij} y_k,$$

Have: $\mathbf{y} \leftarrow A^{-1}\mathbf{b} \quad \mathbf{x} := (A, \mathbf{b})$

Know: $f(A, \mathbf{b}; \mathbf{y}) = A\mathbf{y} - \mathbf{b} = 0$

Want: $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = - \left[\frac{\partial f}{\partial \mathbf{y}} \right]^{-1} \frac{\partial f}{\partial \mathbf{x}}$

$$\frac{\partial f}{\partial \mathbf{y}} = A,$$

$$\frac{\partial f}{\partial \mathbf{b}} = -I$$

Have: $\mathbf{y} \leftarrow A^{-1}\mathbf{b} \quad \mathbf{x} := (A, \mathbf{b})$

Know: $f(A, \mathbf{b}; \mathbf{y}) = A\mathbf{y} - \mathbf{b} = 0$

Want: $\boxed{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}} = - \left[\boxed{\frac{\partial f}{\partial \mathbf{y}}} \right]^{-1} \boxed{\frac{\partial f}{\partial \mathbf{x}}}$

$$\boxed{\frac{\partial f}{\partial \mathbf{y}} = A,}$$

$$\boxed{\frac{\partial f}{\partial \mathbf{b}} = -I}$$



$$\boxed{\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = A^{-1}}$$

$$\frac{\partial y}{\partial b} = A^{-1}$$

$$\min_{\theta} \ell(g(\varphi(\theta)))$$

$$\frac{\partial y}{\partial b} = A^{-1}$$

$$\min_{\theta} \ell(\underbrace{A^{-1}b(\theta)}_y)$$

$$\frac{\partial y}{\partial b} = A^{-1}$$

$$\min_{\theta} \ell(\underbrace{A^{-1}b(\theta)}_y)$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \mathbf{A}^{-1}$$

$$\min_{\theta} \ell(\underbrace{\mathbf{A}^{-1} \mathbf{b}(\theta)}_{\mathbf{y}})$$

$$\frac{\partial \ell}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \theta}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \mathbf{A}^{-1}$$

$$\min_{\theta} \ell(\underbrace{\mathbf{A}^{-1} \mathbf{b}(\theta)}_{\mathbf{y}})$$

$$\frac{\partial \ell}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \theta}$$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{b}} = \boldsymbol{A}^{-1}$$

$$\min_{\theta} \ell(\underbrace{\boldsymbol{A}^{-1} \boldsymbol{b}(\theta)}_y)$$

$$\frac{\partial \ell}{\partial \boldsymbol{y}} \boldsymbol{A}^{-1} \frac{\partial \boldsymbol{b}}{\partial \theta}$$

$$\mathbb{R}^{1 \times n} \quad \mathbb{R}^{n \times n} \quad \mathbb{R}^{n \times p}$$

$$\mathbb{R}^{1 \times n} \quad \mathbb{R}^{n \times n} \quad \mathbb{R}^{n \times p}$$

$$\frac{\partial \ell}{\partial y} \left(A^{-1} \frac{\partial b}{\partial \theta} \right)$$

forward mode (JVP)

$$\left(\frac{\partial \ell}{\partial y} A^{-1} \right) \frac{\partial b}{\partial \theta}$$

backward mode (VJP)

$$\mathbb{R}^{1 \times n} \boxed{\mathbb{R}^{n \times n} \mathbb{R}^{n \times p}}$$

$$\mathbb{R}^{1 \times n} \mathbb{R}^{n \times n} \mathbb{R}^{n \times p}$$

$$\frac{\partial \ell}{\partial y} \left(A^{-1} \frac{\partial b}{\partial \theta} \right)$$

forward mode (JVP)

$$\left(\frac{\partial \ell}{\partial y} A^{-1} \right) \frac{\partial b}{\partial \theta}$$

backward mode (VJP)

p linear solves

$$\mathbb{R}^{1 \times n} \mathbb{R}^{n \times n} \mathbb{R}^{n \times p}$$

$$\mathbb{R}^{1 \times n} \mathbb{R}^{n \times n} \mathbb{R}^{n \times p}$$

$$\frac{\partial \ell}{\partial y} \left(A^{-1} \frac{\partial b}{\partial \theta} \right)$$

forward mode (JVP)

$$\left(\frac{\partial \ell}{\partial y} A^{-1} \right) \frac{\partial b}{\partial \theta}$$

backward mode (VJP)

p linear solves

$$\mathbb{R}^{1 \times n} \mathbb{R}^{n \times n} \mathbb{R}^{n \times p}$$

$$\mathbb{R}^{1 \times n} \mathbb{R}^{n \times n} \mathbb{R}^{n \times p}$$

$$\frac{\partial \ell}{\partial y} \left(A^{-1} \frac{\partial b}{\partial \theta} \right)$$

forward mode (JVP)

$$\left(\frac{\partial \ell}{\partial y} A^{-1} \right) \frac{\partial b}{\partial \theta}$$

backward mode (VJP)

p linear solves

1 linear solve

$$\mathbb{R}^{1 \times n} \boxed{\mathbb{R}^{n \times n} \mathbb{R}^{n \times p}}$$

$$\boxed{\mathbb{R}^{1 \times n} \mathbb{R}^{n \times n}} \mathbb{R}^{n \times p}$$

$$\frac{\partial \ell}{\partial y} \left(A^{-1} \frac{\partial b}{\partial \theta} \right)$$

forward mode (JVP)

$$\left(\frac{\partial \ell}{\partial y} A^{-1} \right) \frac{\partial b}{\partial \theta}$$

backward mode (VJP)

p linear solves

1 linear solve

$$\mathbb{R}^{1 \times n} \mathbb{R}^{n \times n} \mathbb{R}^{n \times p}$$

$$\mathbb{R}^{1 \times n} \mathbb{R}^{n \times n} \mathbb{R}^{n \times p}$$

$$\frac{\partial \ell}{\partial y} \left(A^{-1} \frac{\partial b}{\partial \theta} \right)$$

forward mode (JVP)

$$\left(\frac{\partial \ell}{\partial y} A^{-1} \right) \frac{\partial b}{\partial \theta}$$

backward mode (VJP)
("adjoint method")

$$\underbrace{(M + \tau L)}_A \underbrace{v'}_y = \underbrace{Mv}_b$$



Inverse Fairing (H1 Norm)

a.k.a. mesh inflation

Inverse Geometry Processing in Python

PyTorch Code

The standard PyTorch optimization loop

```
# Optimization variable
theta.requires_grad_(True)

# Optimizer
optimizer = optim.SGD([theta], lr=...)

for _ in range(num_steps):
    optimizer.zero_grad()
    y = layer(theta)      # Forward pass
    l = loss(y, target)   # Define loss
    l.backward()          # Backward pass
    optimizer.step()      # Update theta
```

Target code for Inverse Geometry Processing Problem

$$\min_{\theta} \ell(\underbrace{g(\phi(\theta))}_y)$$

```
theta.requires_grad_(True)
optimizer = optim.SGD([theta], lr=...)

for _ in range(num_steps):
    optimizer.zero_grad()
    x = phi(theta)
    y = geometric_layer(x)
    loss = objective(y)
    loss.backward()
    optimizer.step()
```

*how to design/implement an efficient
geometric_layer?*

What might go into geometric_layer?

```
def geometric_layer(x):  
    A, b = assemble_sparse_system()  
    y = sparse_solve(A, b, x)  
    return y
```

Sparse linear solve

```
def geometric_layer(x):  
    A, B = assemble_eigenproblem(x)  
    eval, evec = eigensolve(A, B)  
    y = evec @ torch.diag(eval) @ evec.T  
    return y
```

Eigen solver

```
def geometric_layer(x):  
    y = initialize(x)  
    for _ in range(max_iter):  
        y = local_step(x, y)  
    return y
```

Local steps

First attempt to write geometric_layer

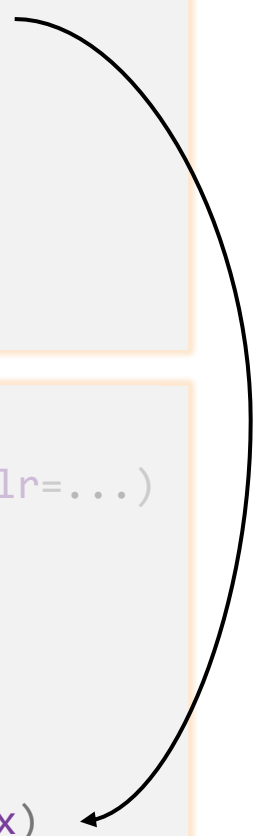
Let PyTorch trace the solver:

- Works in principle if every operation is implemented in PyTorch and differentiable
- But the cost is bad
 - Memory is $O(\text{max_iter})$ since backward pass needs access to the intermediate state $y_1, y_2, \dots, y_n$

we only care about y , not the path

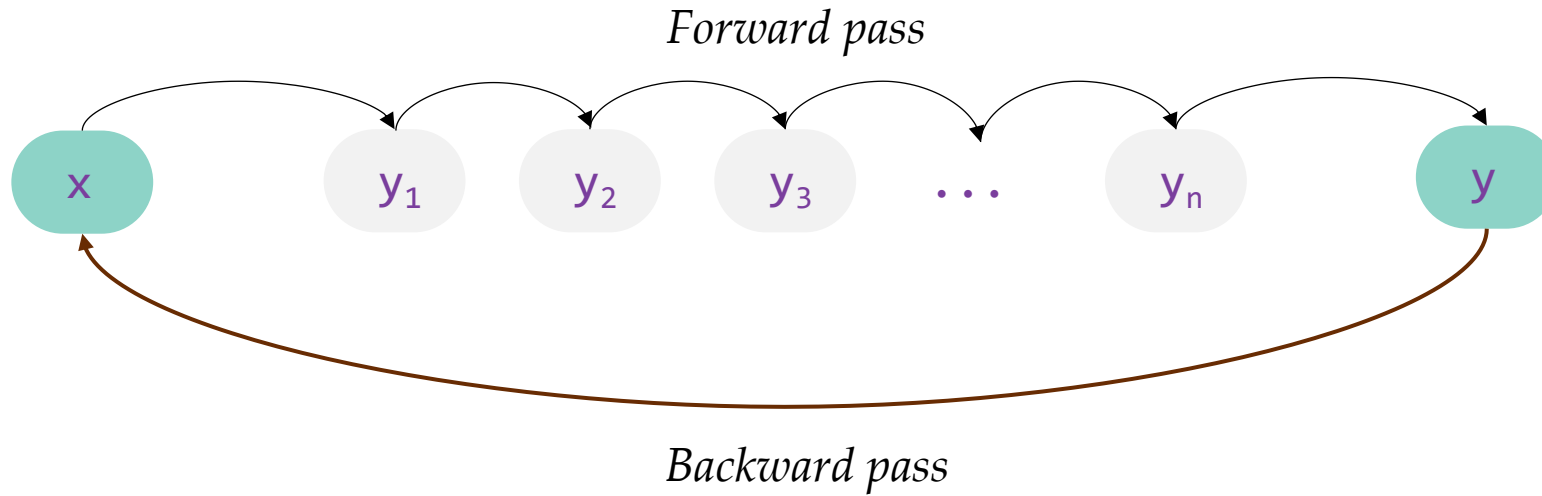
```
def geometric_layer_unroll(x):  
    y = initialize(x)  
    for _ in range(max_iter):  
        y = local_step(x, y)  
    return y
```

```
theta.requires_grad_(True)  
optimizer = optim.SGD([theta], lr=...)  
  
for _ in range(num_steps):  
    optimizer.zero_grad()  
    x = phi(theta)  
    y = geometric_layer_unroll(x)  
    loss = objective(y)  
    loss.backward()  
    optimizer.step()
```



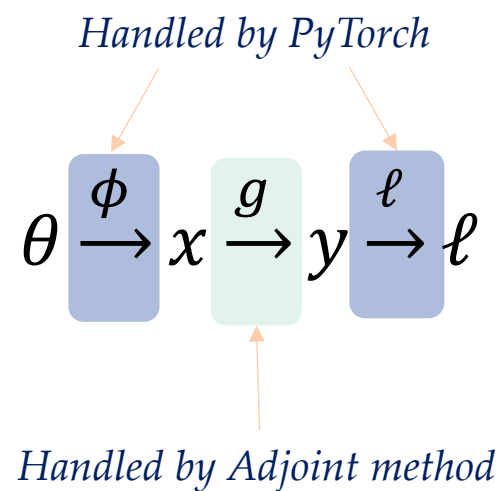
What would be a good abstraction?

Separate the *forward* from the
backward pass



The derivative pipeline

$$\min_{\theta} \ell(\underbrace{g(\phi(\theta))}_y)$$



Implementing the Adjoint method

$$\min_{\theta} \ell(\underbrace{g(\phi(\theta))}_y)$$

Define $x = \phi(\theta)$ and assume y is defined implicitly by $f(x; y) = 0$

$$\frac{\partial \ell}{\partial x} = - \underbrace{\frac{\partial \ell}{\partial y} \left[\frac{\partial f}{\partial y} \right]^{-1}}_{\lambda^T} \frac{\partial f}{\partial x}$$

First solve for λ

$$\left(\frac{\partial f}{\partial y} \right)^T \lambda = \left(\frac{\partial \ell}{\partial y} \right)^T$$

Then, compute

$$\frac{\partial \ell}{\partial x} = -\lambda^T \frac{\partial f}{\partial x}$$

We need to compute three partial derivatives

$$\frac{\partial \ell}{\partial y}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}$$

Extending `torch.autograd`

Add a new operation to the graph, with its own backward rule

$\frac{\partial \ell}{\partial y}$ in our case

```
class CustomAutogradFunction(torch.autograd.Function):  
  
    @staticmethod  
    def forward(x):  
        y = custom_forward(x)  
        return y  
  
    @staticmethod  
    def setup_context(ctx, inputs, output):  
        x, = inputs  
        ctx.save_for_backward(x, output)  
  
    @staticmethod  
    def backward(ctx, grad_output):  
        x, y = ctx.saved_tensors  
        grad_x = custom_backward(x, y, grad_output)  
        return grad_x
```



#1: incoming gradient from the outer loss $\frac{\partial \ell}{\partial y}$

$$\left(\frac{\partial f}{\partial y}\right)^T \lambda = \left(\frac{\partial \ell}{\partial y}\right)^T$$

$\frac{\partial \ell}{\partial y}$ is the gradient of the outer loss w.r.t. the solver output.

We don't compute it — PyTorch hands it to us as `grad_output`

```
class CustomAutogradFunction(torch.autograd.Function):

    @staticmethod
    def forward(x):
        y = custom_forward(x)
        return y

    @staticmethod
    def setup_context(ctx, inputs, output):
        x, = inputs
        ctx.save_for_backward(x, output)

    @staticmethod
    def backward(ctx, grad_output):
        x, y = ctx.saved_tensors
        grad_x = custom_backward(x, y, grad_output)
        return grad_x
```

#2: implicit function derivative with respect to the solver output $\frac{\partial f}{\partial y}$

$$\left(\frac{\partial f}{\partial y}\right)^T \lambda = \left(\frac{\partial \ell}{\partial y}\right)^T$$

$$f(x; y) = 0$$

$\frac{\partial f}{\partial y}$: how the implicit function moves when we perturb y (fix x). $n_y \times n_y$

Huge — but a matrix-free solve only needs $z \mapsto \left(\frac{\partial f}{\partial y}\right)^T z$

```
def implicit_fn(y, x):  
    return compute_implicit_fn(y, x)  
  
f = implicit_fn(y, x) # y, x require grad  
  
def A_T(z): #  $z \mapsto \left(\frac{\partial f}{\partial y}\right)^T z$   
    vjp_y = torch.autograd.grad(f, y, z, retain_graph=True)  
    return vjp_y  
  
lambda_ = linear_solve(A_T, grad_output)
```

#3: implicit function derivative with respect to the solver input $\frac{\partial f}{\partial x}$

$$\frac{\partial \ell}{\partial x} = -\lambda^T \frac{\partial f}{\partial x}$$

$$f(x; y) = 0$$

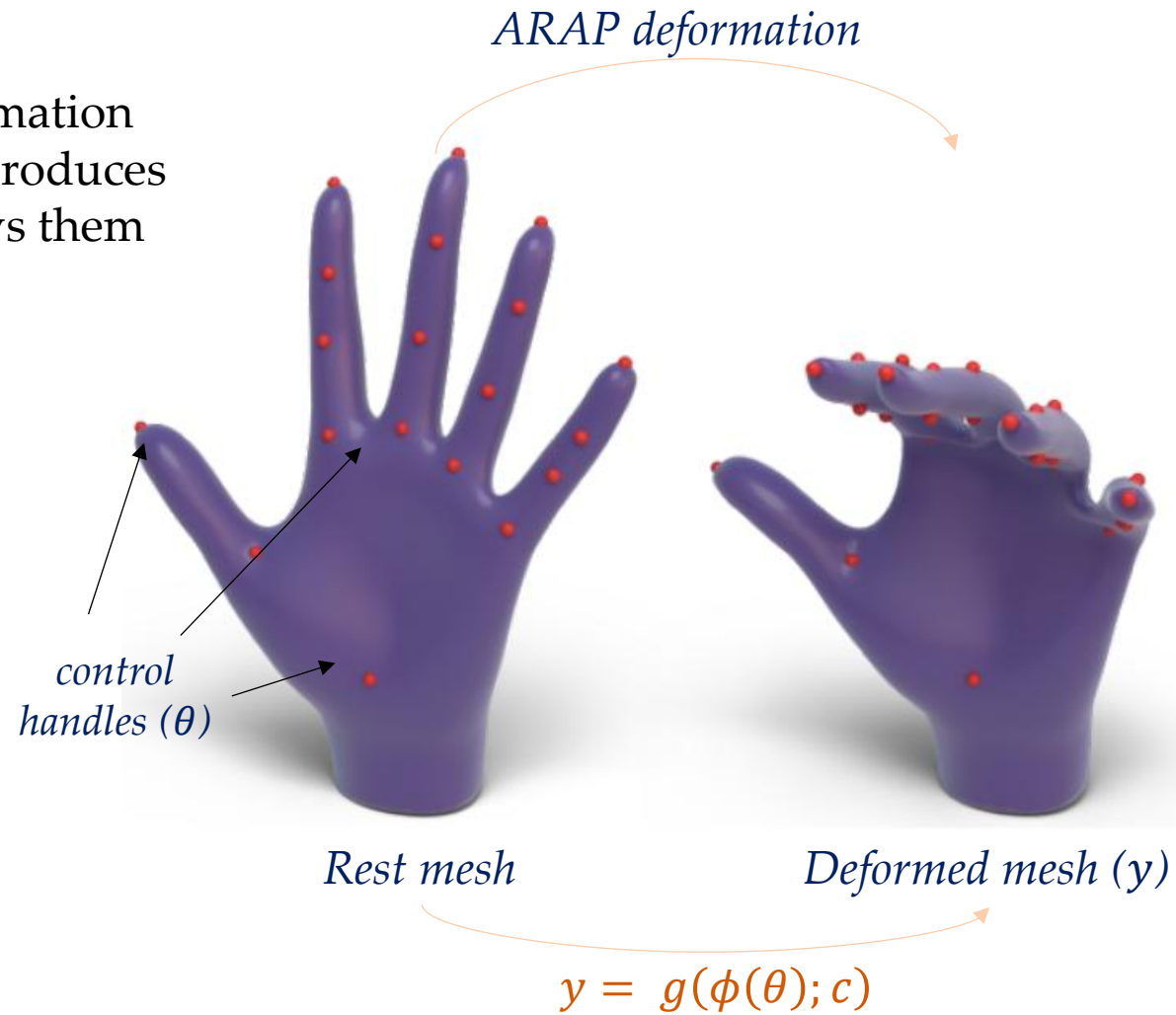
$\frac{\partial f}{\partial x}$: how the implicit function moves when we perturb x (fix y). $n_y \times n_x$

Again, don't build it — the final gradient needs only the VJP contracted against λ .

```
def implicit_fn(y, x):  
    return compute_implicit_fn(y, x)  
  
f = implicit_fn(y, x) # y, x require grad  
vjp_x = torch.autograd.grad(f, x, lambda_, retain_graph=True)  
grad_x = -vjp_x
```

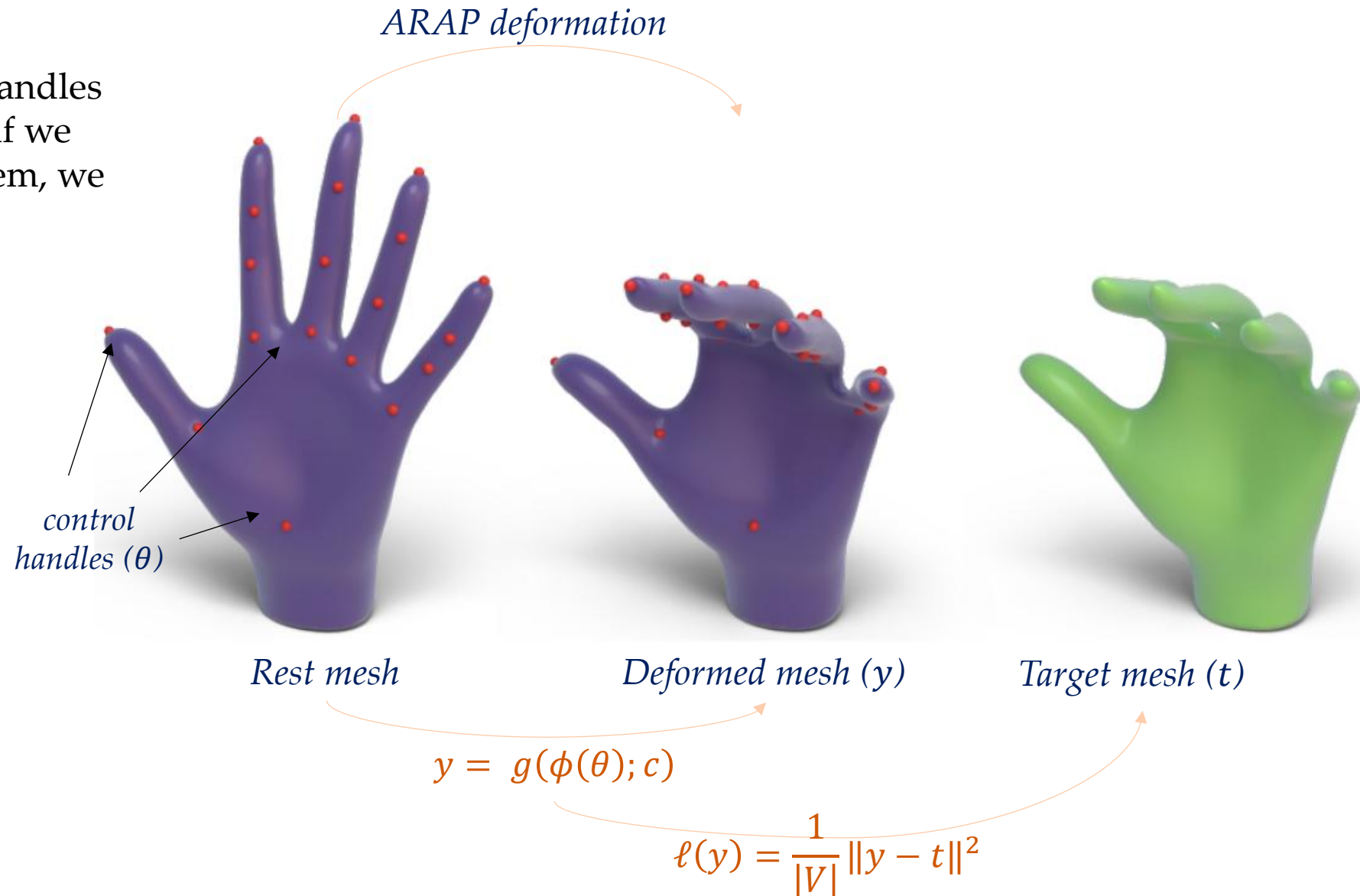

Example: Inverse ARAP deformation

Forward problem: ARAP deformation takes handle positions (θ) and produces a deformed mesh (y) that follows them



Example: Inverse ARAP deformation

Inverse problem: find control handles location on rest mesh such that if we apply ARAP deformation on them, we get the target shape.



ARAP Deformation (forward problem)

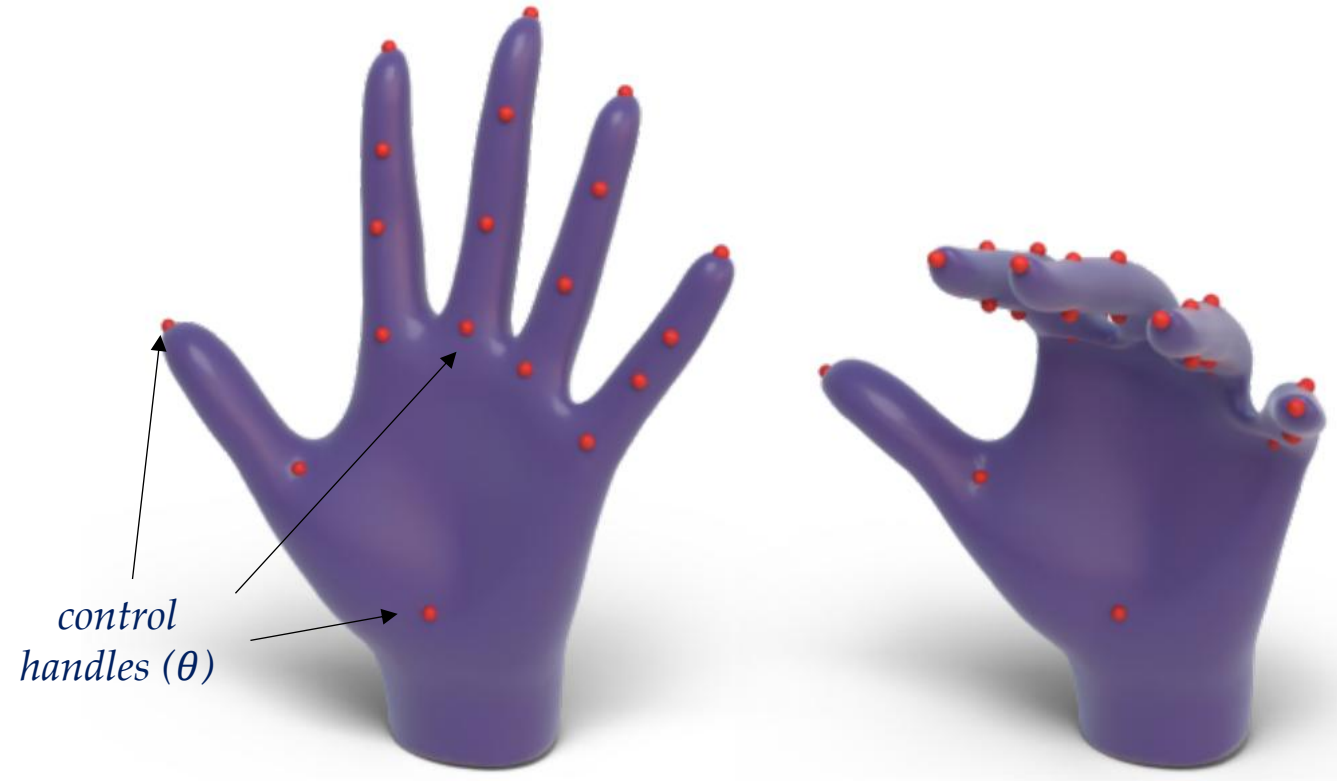
Forward problem: ARAP deformation takes handle positions (θ) and produces a deformed mesh (y) that follows them

$$\min_{y, \{R_i\}} \sum_{(i,j)} c_{ij} \|(y_i - y_j) - R_i(x_i - x_j)\|^2$$

such that:

$$y_k = \theta_k, \quad k \in H,$$

$$R_i \in SO(3)$$



Rest mesh

Deformed mesh (y)

ARAP Deformation: local-global steps

ARAP is usually solved via local-global iterations

$$\textcircled{1} \begin{aligned} S_i &= \sum_{j \in \mathcal{N}(i)} w_{ij} e_{ij} e_{ij}'^T \\ e_{ij} &:= x_i - x_j \\ e_{ij}' &:= y_i - y_j \end{aligned}$$

$$\textcircled{2} S_i = U_i \Sigma_i V_i^T$$

$$\textcircled{3} R_i = V_i U_i^T$$

Local Step

```
fit_local_rotations (...)
```

$$b_i = \sum_{j \in \mathcal{N}(i)} \frac{c_{ij}}{2} (R_i + R_j) (x_i - x_j)$$

$$L y^{k+1} = b$$

such that:

$$y_H^{k+1} = \theta_H$$

linear solver

Global Step

```
assemble_global_rhs(...)  
solve_global_system(...)
```

ARAP Deformation: local-global steps (fixed point)

$$y^{k+1} = h(x, y^k)$$

At convergence:

$$y^* = h(x, y^*)$$

Thus, the solver output is defined by the implicit function

$$f(x; y) = h(x, y) - y = 0$$

```
def arap_step(y, rest_verts, handle_idx, theta, ...):  
    rotations = fit_local_rotations(y, rest_verts, ...)  
    rhs = assemble_global_rhs(rotations, rest_verts, ...)  
    y_next = solve_global_system(rhs, handle_idx, theta, ...)  
    return y_next
```

Implicit layer forward step

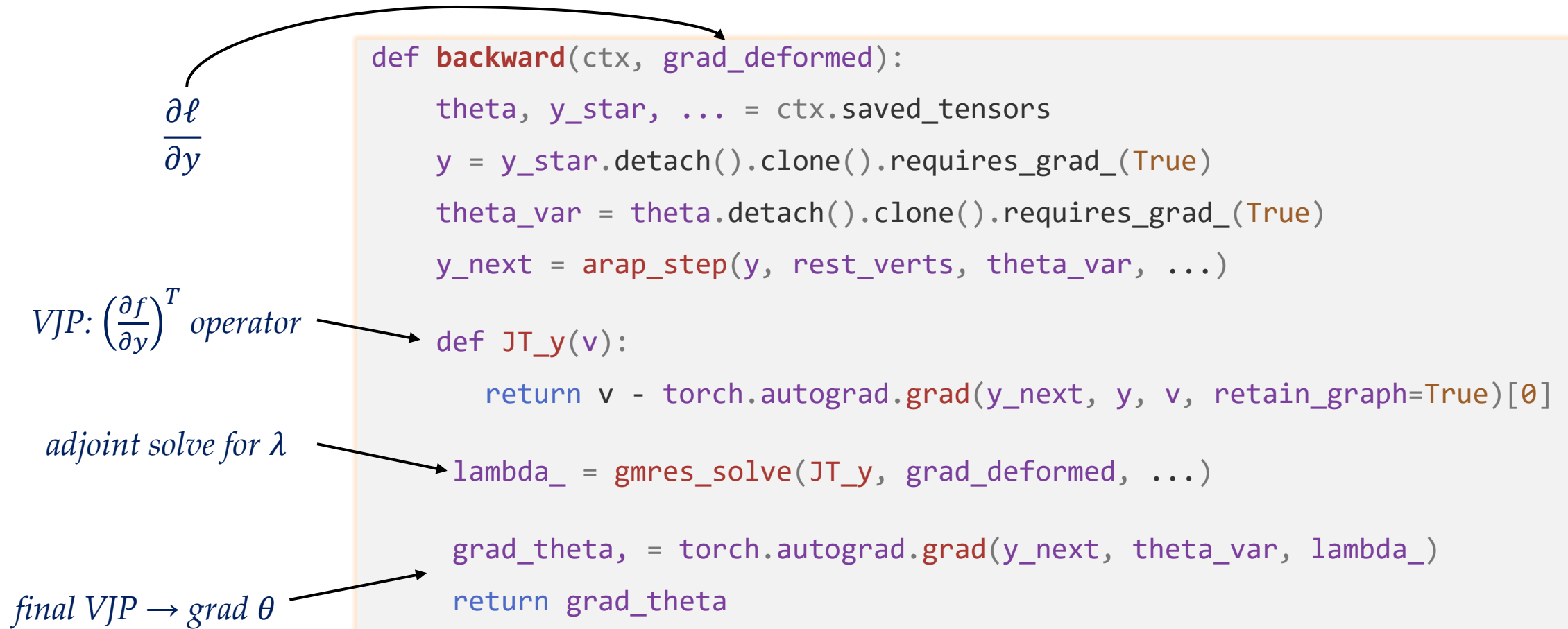
initialize the deformed mesh from the rest mesh and insert the current handle positions

each iteration performs one local-global ARAP step

For the backward pass, we only need the fixed-point condition at the solution: $y^ = g(x, y^*)$*

```
def forward(rest_verts, handle_idx, theta, ...):  
    y = rest_verts.clone()  
    y[handle_idx] = theta  
    for _ in range(fwd_max_iter):  
        y_next = arap_step(  
            y, # current deformed mesh  
            rest_verts, # fixed rest mesh  
            handle_idx, # handle constraints  
            ...)  
        error = torch.linalg.norm((y_next - y).flatten())  
        y = y_next  
        if error <= tol:  
            break  
    return y
```


Implicit layer backward step



Our implicit function

$$f(x; y) = h(x, y) - y = 0$$

Put everything together

```
# Optimization variable: handle positions
theta = verts[handle_idx]

theta = theta.requires_grad_(True)

optimizer = torch.optim.SGD([theta], lr=lr)

for _ in range(num_steps):

    optimizer.zero_grad()

    # g(x): solve ARAP deformation
    y = arap_layer(verts, theta, ...)

    # l(y): target matching loss
    loss = ((y - target_verts) ** 2).mean()

    loss.backward()

    optimizer.step()
```

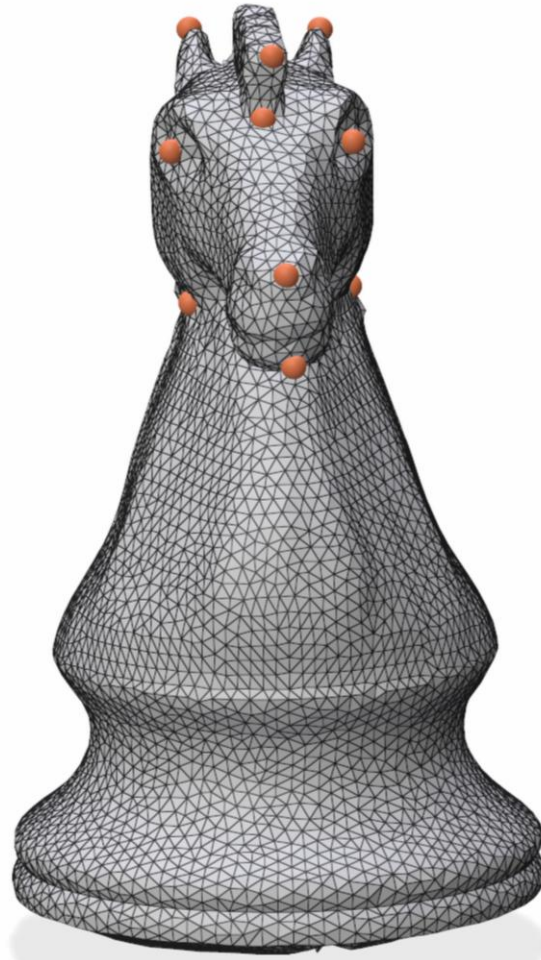
```
def arap_layer(verts, theta, ...):
    class ARAPLayer(torch.autograd.Function):
        @staticmethod
        def forward(verts, theta, ...):
            ...
            return y
        @staticmethod
        def backward(ctx, grad_deformed):
            ...
            return grad_theta

    return ARAPLayer.apply(verts, theta, ...)
```

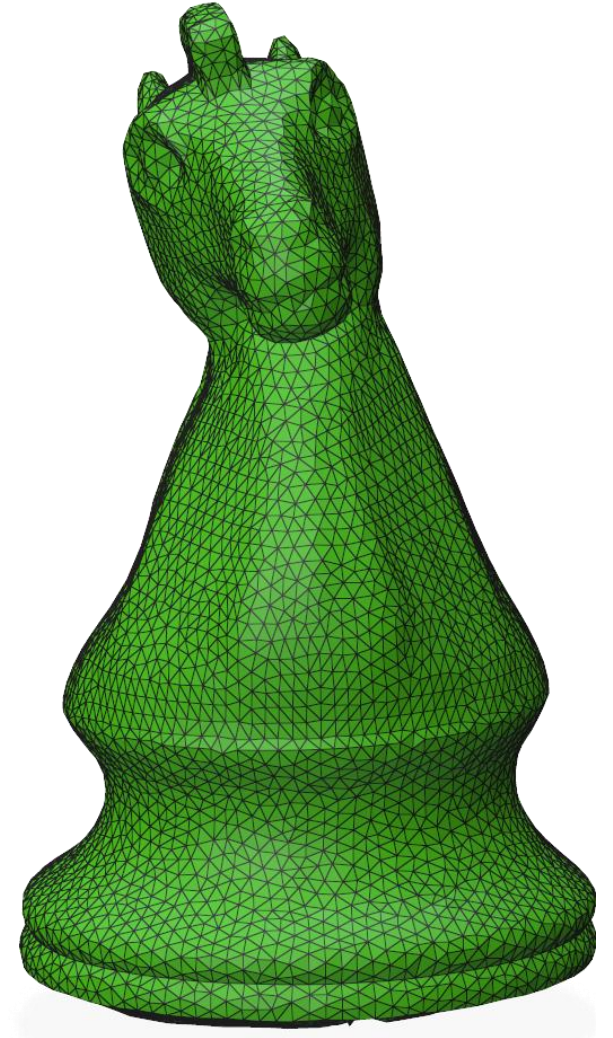
Inverse ARAP Results



Rest mesh



ARAP result after
each outer iteration



Target mesh

iskra: Inverse Geometry Processing in Python

PyTorch system for differentiable geometry processing, with automatic adjoint computation.

User provides g and f , and *iskra* builds the implicit layer



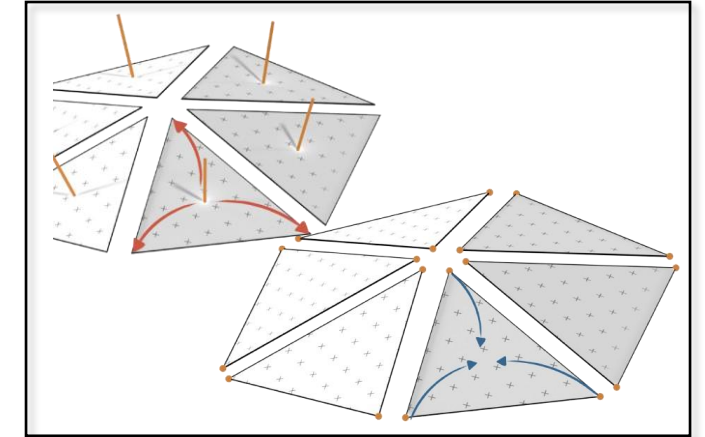
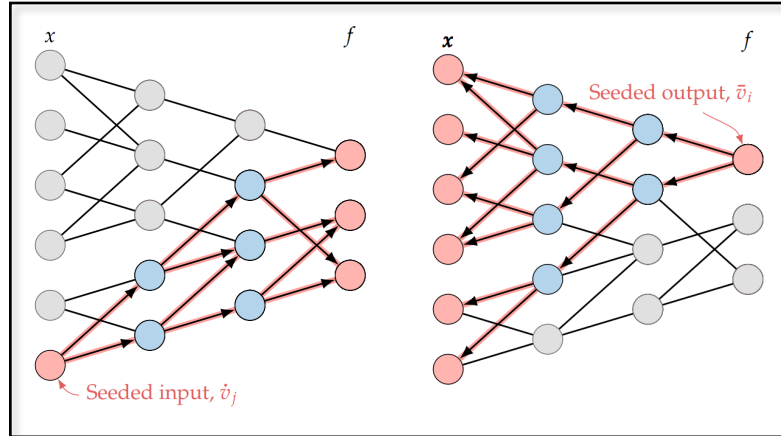
github.com/anadodik/iskra

```
def make_adjoint_layer(  
    solver_fn,           #forward solver g.  
    implicit_fn,         #implicit relation f  
    param_args,          #inputs treated as x  
    sol_args,            #outputs treated as y  
    zero_args,           #residual entries constrained to zero  
    bwd_method = "gmres",  
    gmres_init = None,  
    callback_gmres_sol = None, #Callback function that exposes  
                                the GMRES solution to the user  
  
    bwd_max_iter = 500,  
    bwd_abs_tol = 1e-6,  
    bwd_rel_tol = 1e-3,  
    verbose = False,  
)
```

Conclusion

What have we done today?

Differentiating for | through Geometry Processing

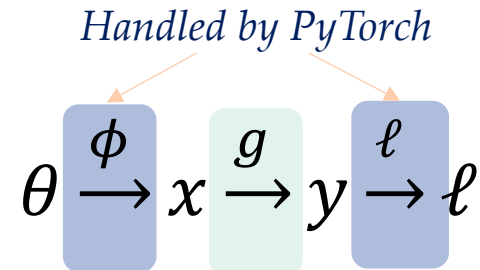


Have: $y \leftarrow g(x)$

Know: $f(x, y) = 0$

Want: $\frac{\partial y}{\partial x}$

Implicit Differentiation



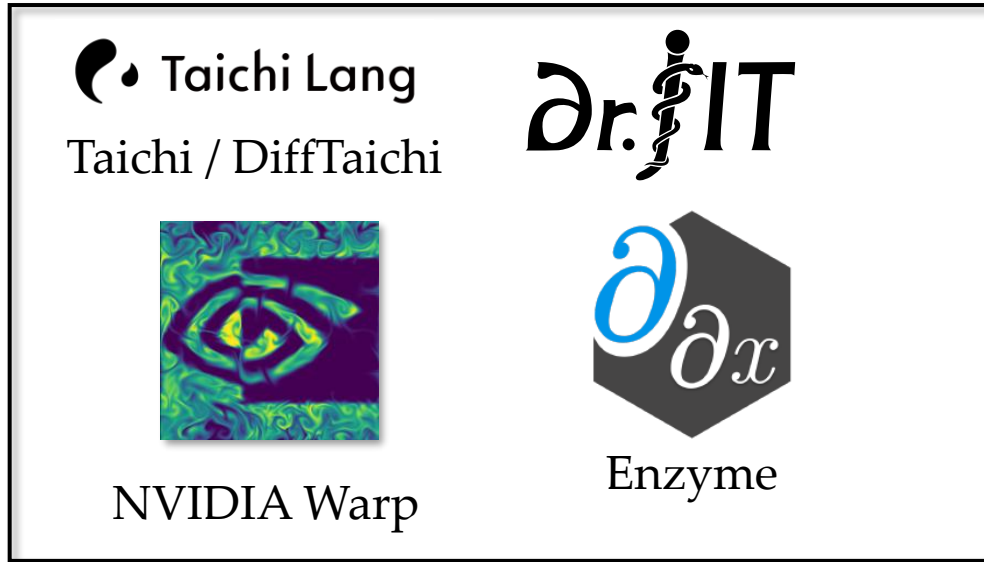
Handled by PyTorch

Handled by Adjoint method

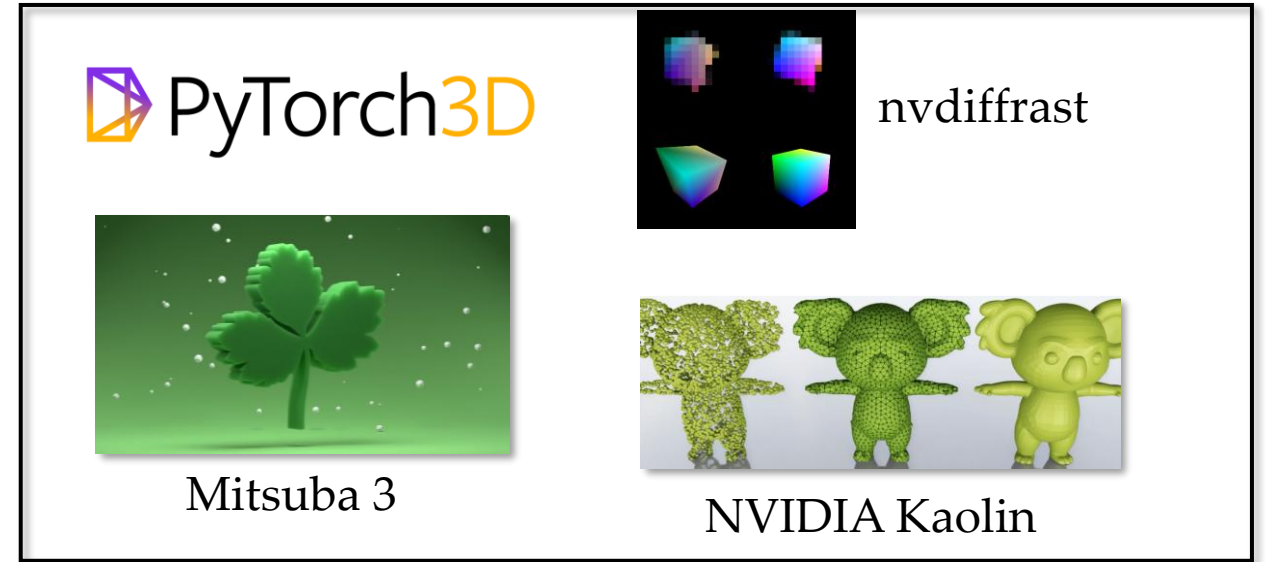


iskra

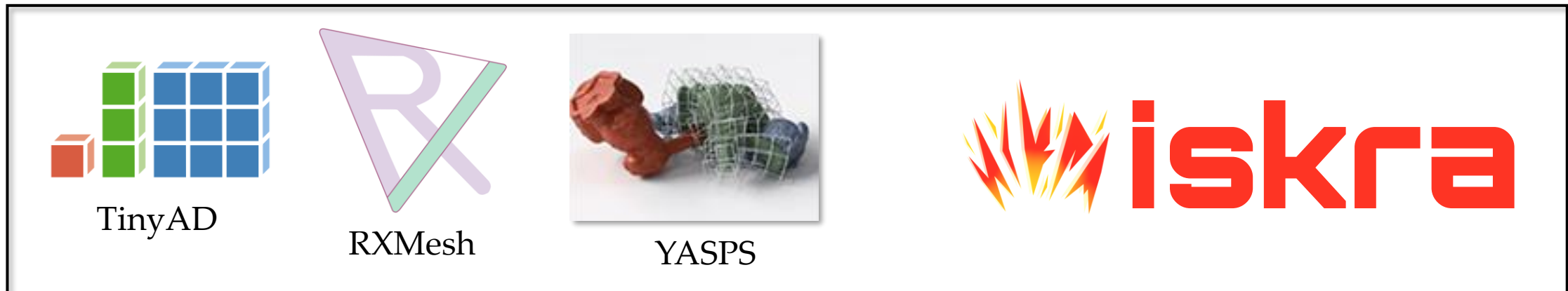
The Landscape of Differentiable Systems



General Differentiable Programming



3D Learning / Differentiable Rendering



Differentiable Geometry Processing

Open Problems

Systems

Geometry

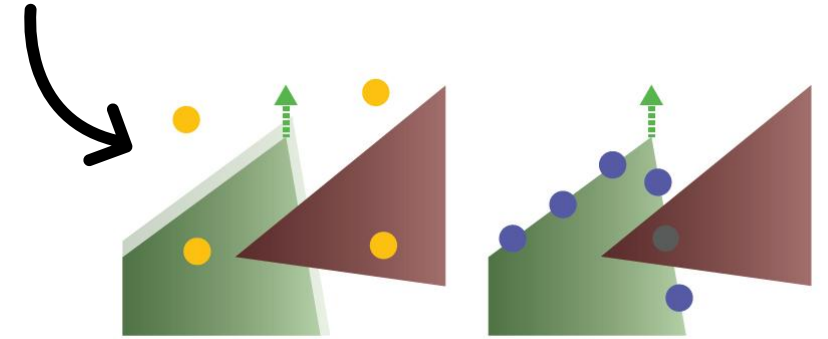


1

Differentiable Discrete
Operations w/ *Connectivity*
Change

2

Differentiating through
Degeneracies



(a) area sampling

(b) edge sampling

3

System Design for Composable
Geometry Layers

*Differentiable Monte Carlo Ray Tracing
through Edge Sampling*
Siggraph 2018
Tzu-Mao Li et al.



github.com/anadodik/iskra



 **iskra**

DIFFERENTIABLE GEOMETRY PROCESSING

IN PYTHON

ANA DODIK, AHMED H. MAHMOUD

