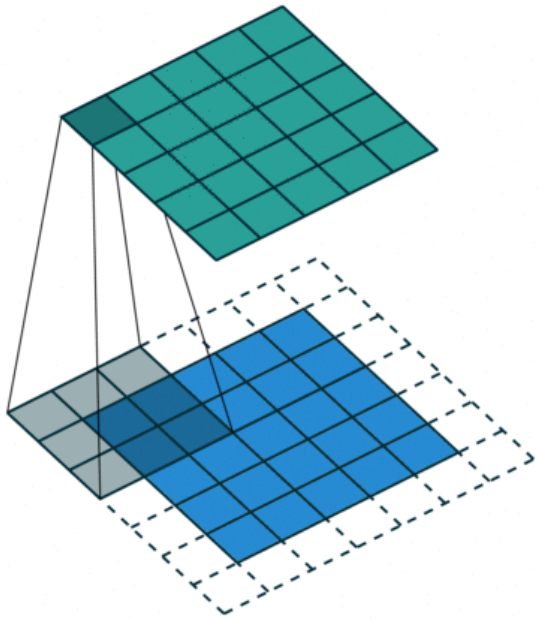
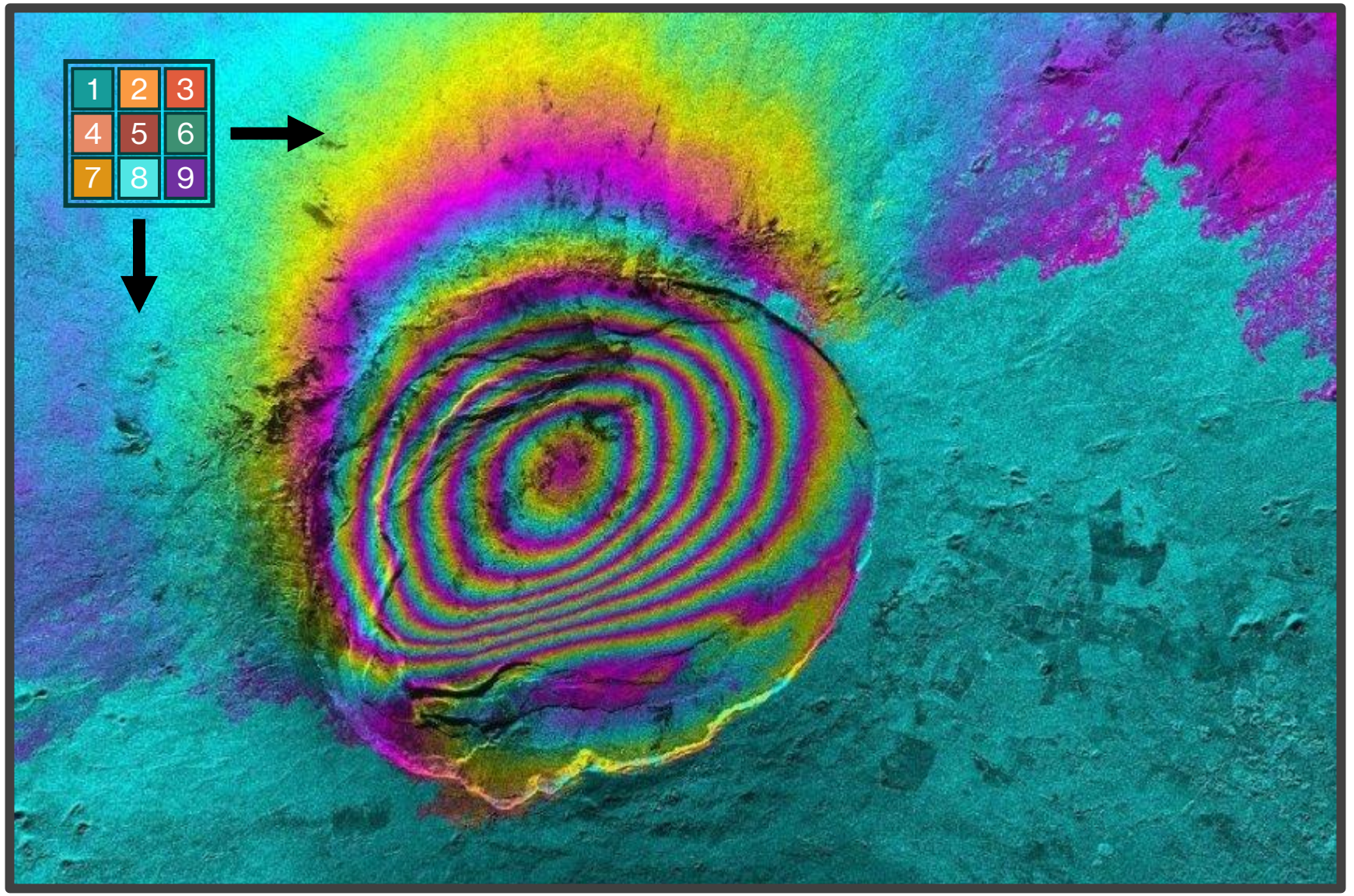
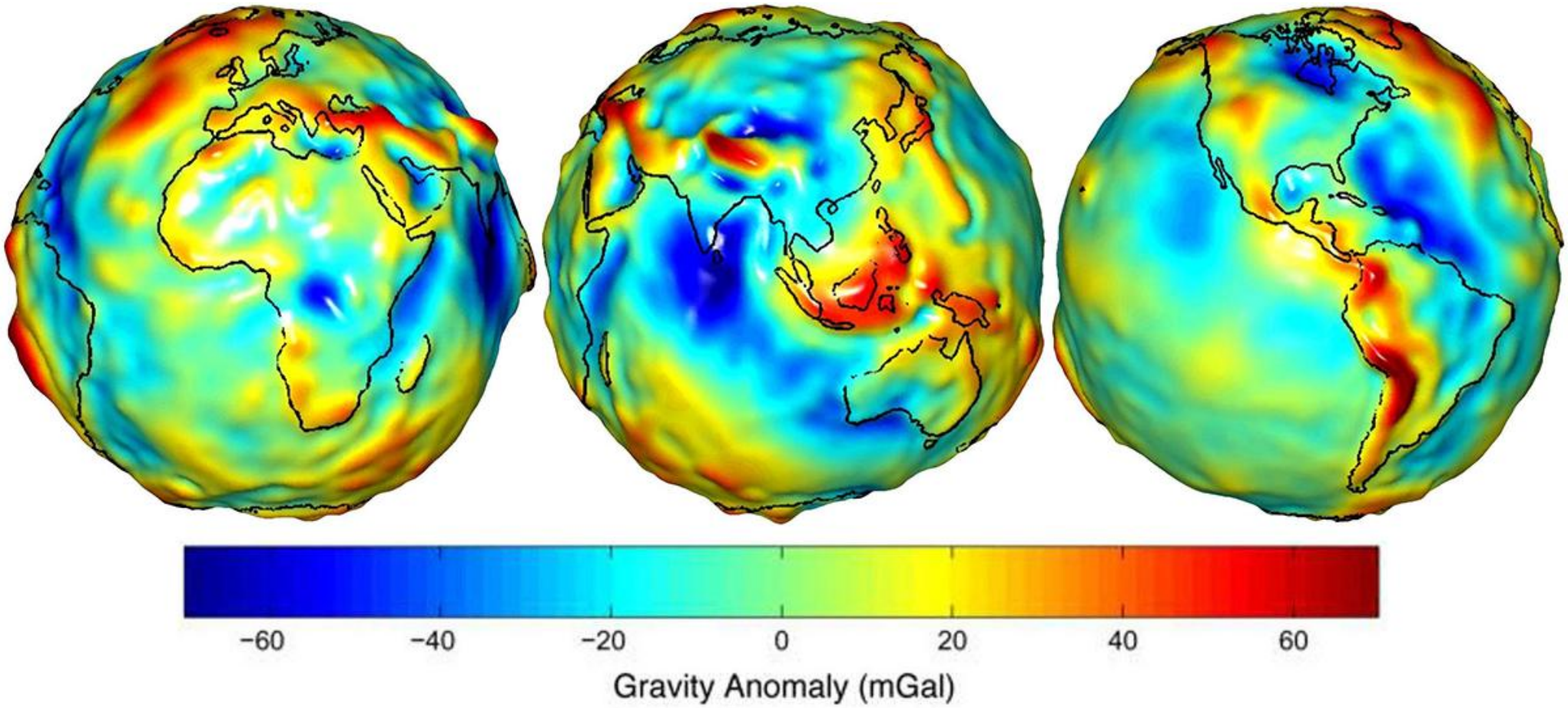
Graph Neural Networks

Martijn van den Ende
Université Côte d'Azur, Géoazur
La Sapienza Università di Roma

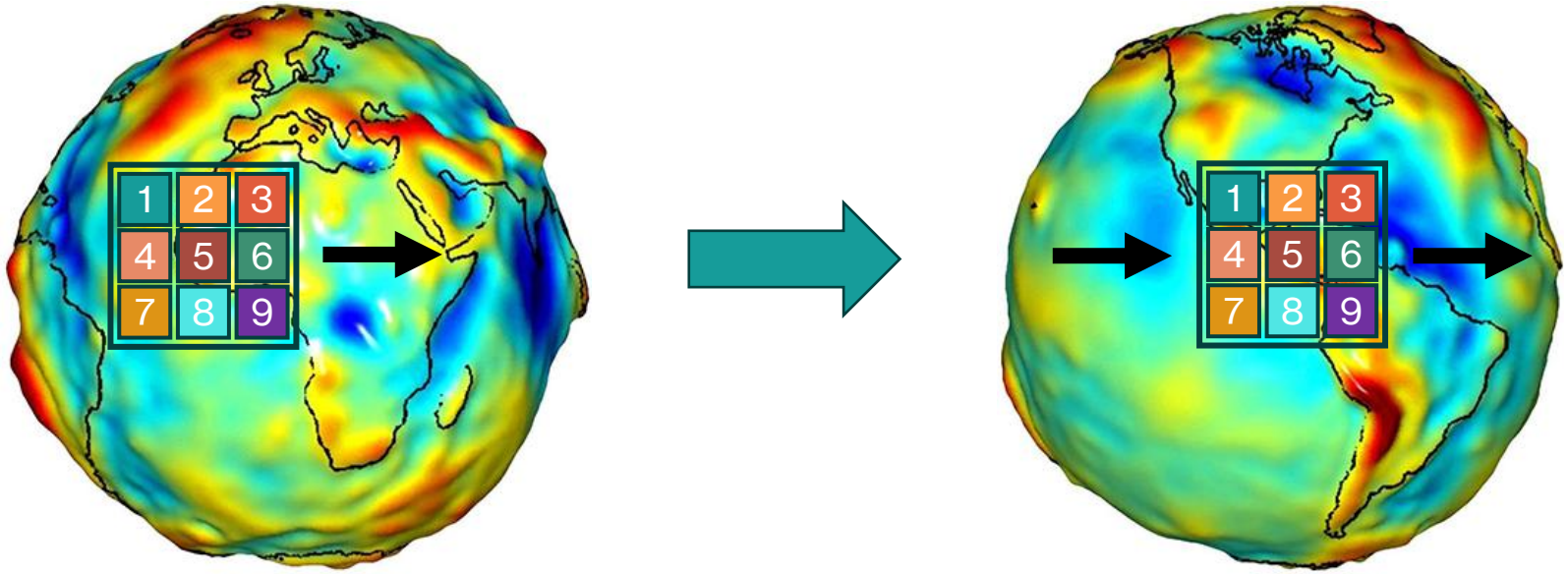




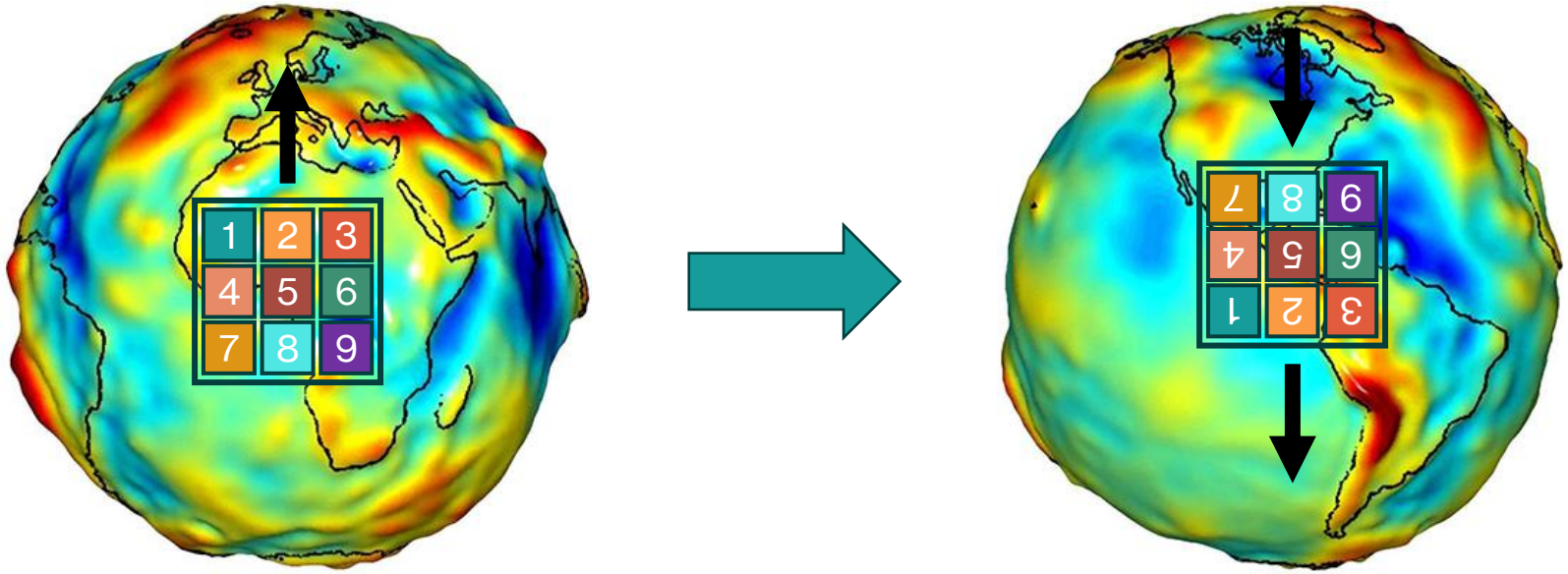
Dumoulin & Visin (2016), arXiv:1603.07285



Translating kernel along the equator (“horizontal”)



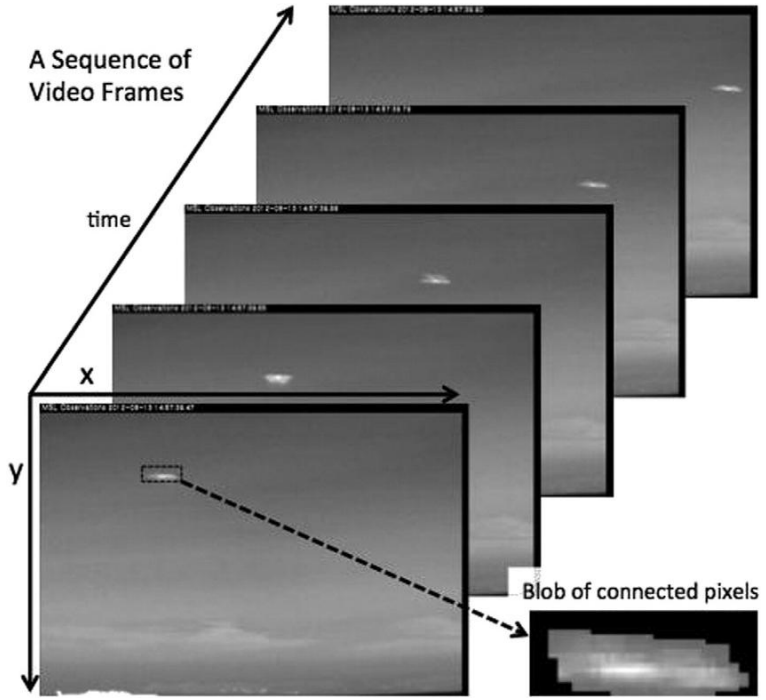
Translating kernel over the poles (“vertical”)



Learning from Euclidean Objects

Euclidean “data objects” are regularly structured and are space-filling (not sparse)

- Images (2D) and volumes (3D)
- Time-series (1D)
- Videos (2+1D)
- Tabular data (0D)

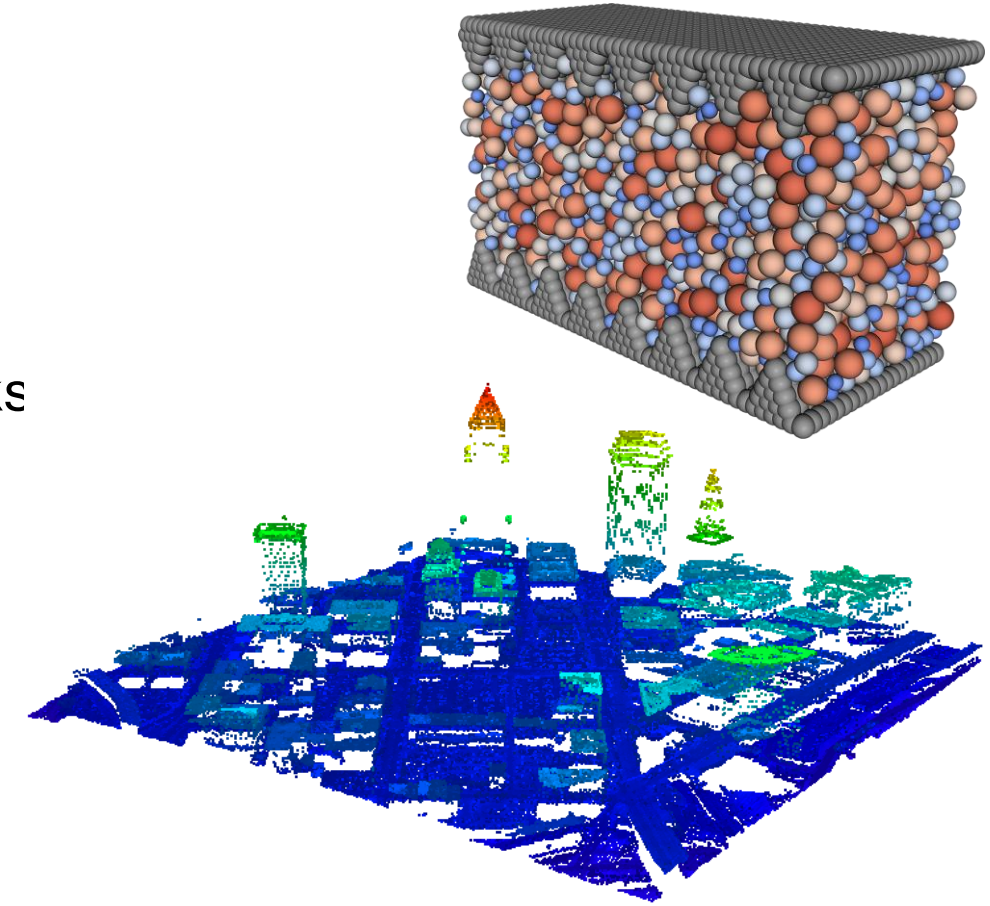


Matzner et al. (2015)

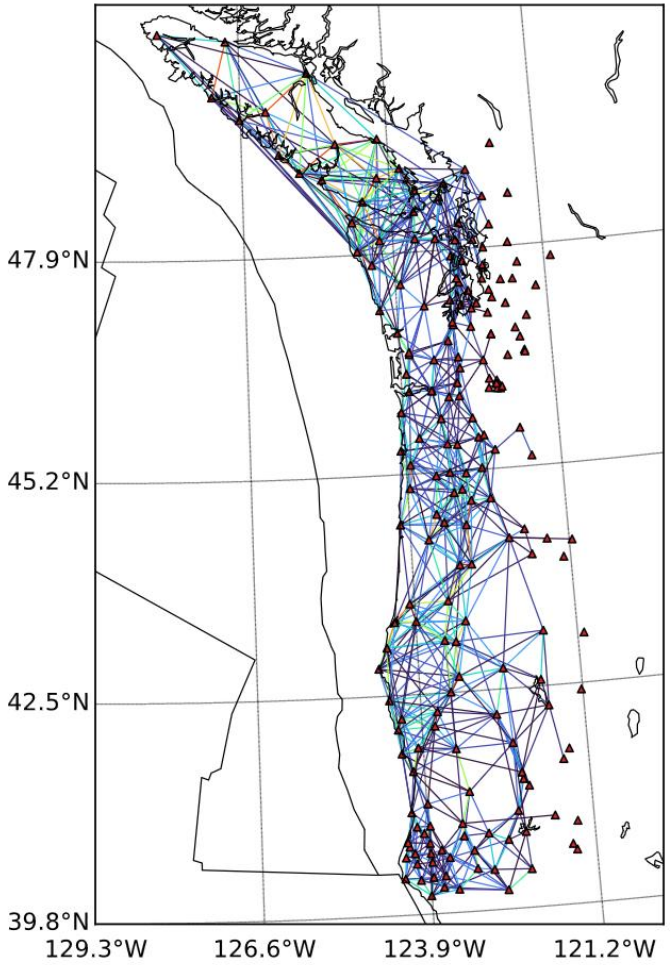
Learning from Non-Euclidean Objects

Non-Euclidean objects are more general

- LiDAR point clouds
- Earthquake catalogues
- Seismic / geodetic networks
- Molecular structures
- Discrete Element Models



Costantino et al. (2024)



Learning from Non-Euclidean Objects

Naïve solution: forcing non-Euclidean objects into a Euclidean form...

- Interpolating point clouds onto a regular grid
- Arranging seismograms in an image sorted by latitude
- Computing binned statistics (e.g., # of earthquakes per day per grid cell)
- Computing global features (# of C-atoms, H-atoms, etc.)

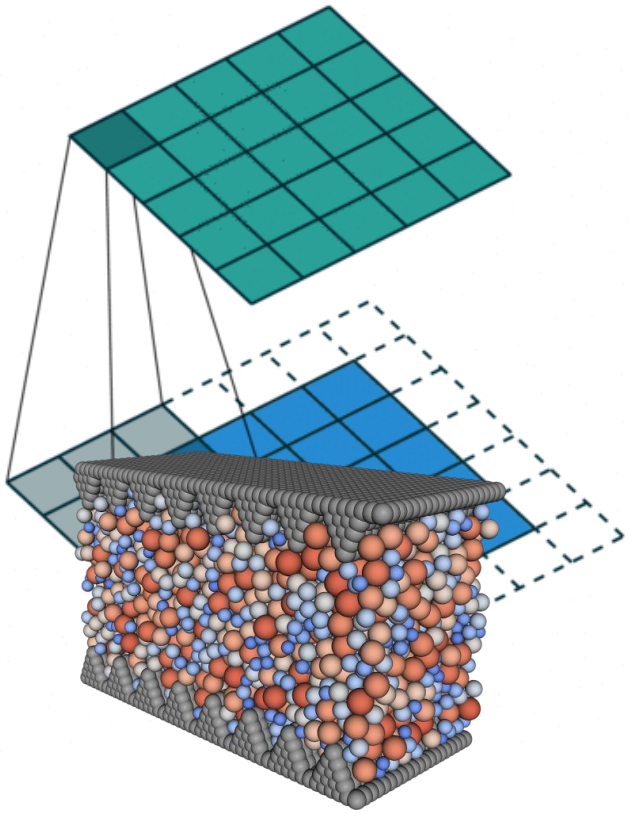
Better solution: generalise learning methods to manifolds

Objective: Convolutions on a Graph

- Generalise the Deep Learning “convolution” operation to arbitrary manifolds (discrete manifold = graph)
- Convolution theorem for 1D Euclidean spaces:

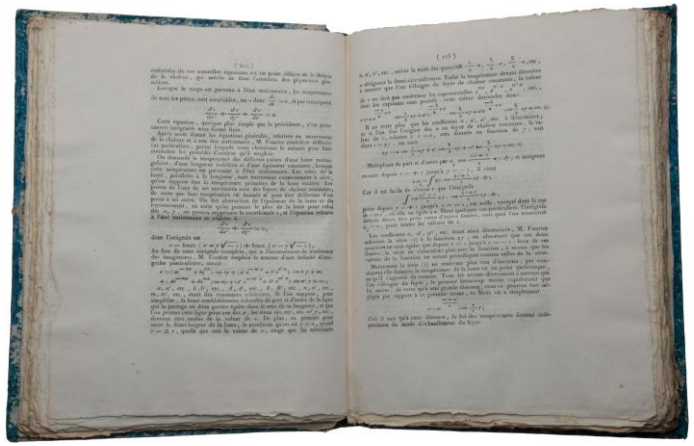
$$(f * g)[k] = \sum_i f[i]g[k-i] = \mathcal{F}^{-1}[\mathcal{F}(f)\mathcal{F}(g)]$$

(convolution between data f and kernel g is the inverse Fourier transform of the element-wise product between the Fourier transform of f and that of g)



What is the Fourier transform of a graph?

The Fourier Transform



"Mémoire sur la propagation de la chaleur dans les corps solides"
($12,500 on www.sophiararebooks.com)

Joseph Fourier tried to find a solution to the heat equation:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -\Delta u = \mathrm{div}(\nabla u) = \left(\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + \frac{\mathrm{d}^2u}{\mathrm{d}y^2} + \frac{\mathrm{d}^2u}{\mathrm{d}z^2}\right)$$

"Laplacian"

The *Fourier Transform* $\mathcal{F}: u(x) \to U(k)$ decomposes a function into a superposition of plane waves:

$$U(k) = \int_{-\infty}^{+\infty} u(x) e^{-i2\pi kx} \mathrm{d}x \approx \sum_n u[n] e^{-i2\pi \frac{kn}{N}}$$

Basis of the Fourier Transform

The Fourier Transform

The Laplacian ("2nd derivative") of a plane wave gives you just another plane wave:

$$\Delta e^{-i2\pi xk} = -4\pi^2 k^2 e^{-i2\pi xk}$$

Or in matrix notation:

$$\mathbf{\Delta\Phi = \Phi\Lambda} \rightarrow \boxed{\mathbf{\Delta = \Phi\Lambda\Phi^*}}$$

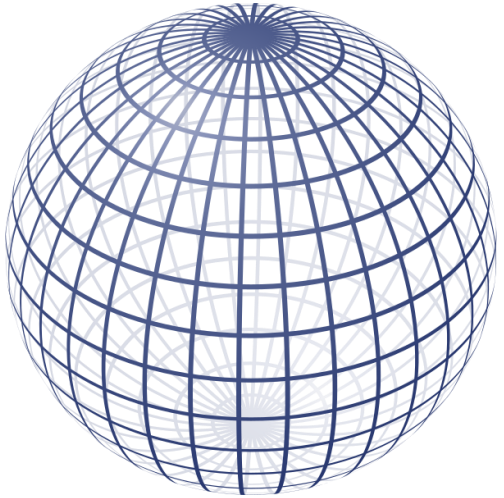
Eigenvectors $\mathbf{\Phi}$ and eigenvalues $\mathbf{\Lambda}$

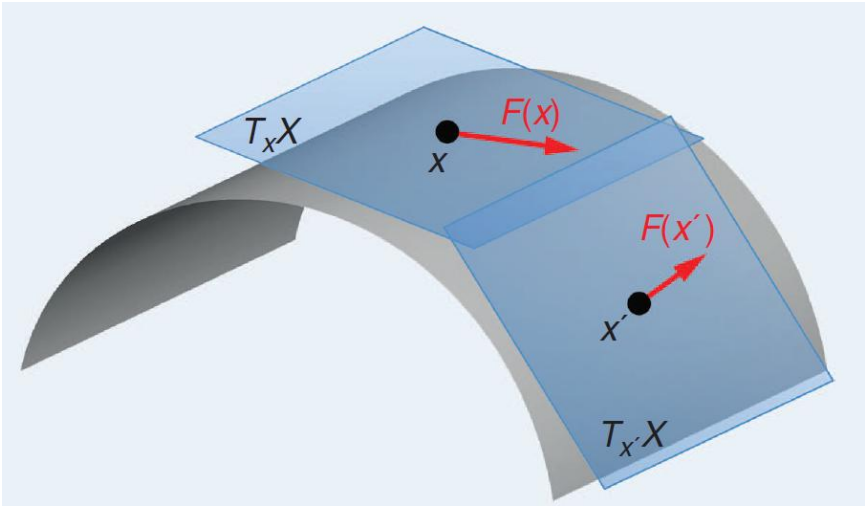
In other words: the *basis* of the Fourier Transform (plane waves in Euclidean space) is spanned by the eigenvectors of the Laplacian

In other other words: define the Fourier Transform in terms of the eigenvectors of the Laplacian operator (for a given manifold)

Manifold Laplacians

- Common interpretation of a smooth manifold (X): some shape embedded in 3D space
- Any point x has a corresponding tangent plane T_x that defines local orientation in a Euclidean space
- A gradient operator ∇ is a map from a scalar field f on the manifold to the tangent space: $\nabla: L(X) \to L(T)$
- The adjoint operator is the divergence: $\mathrm{div}: L(T) \to L(X)$
- **Laplacian** $\Delta: L(X) \to L(X), \quad \Delta f = -\mathrm{div}(\nabla f)$





Bronstein et al. (2017),
Geometric Deep Learning: Going beyond Euclidean data

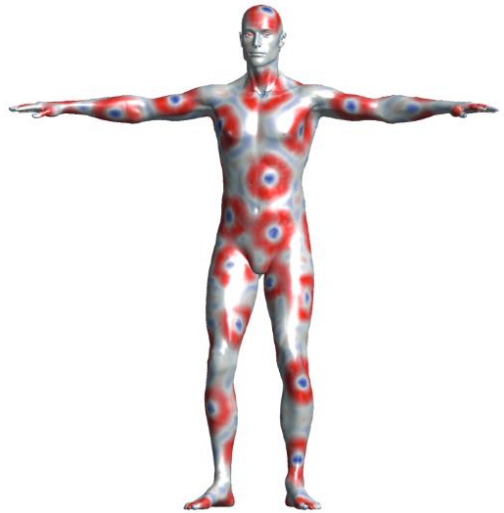
Connection to Geometric Deep Learning

1. Define the Laplacian of a particular manifold ($\Delta = -\mathrm{div}\,\nabla$)
2. Find the K smoothest basis functions ϕ_k (eigenvectors)
3. Compute convolution between manifold (data) f and learnable kernel g:

$$(g * f)[x] = \sum_k \langle g, \phi_k \rangle \langle f, \phi_k \rangle \phi_k^*(x)$$

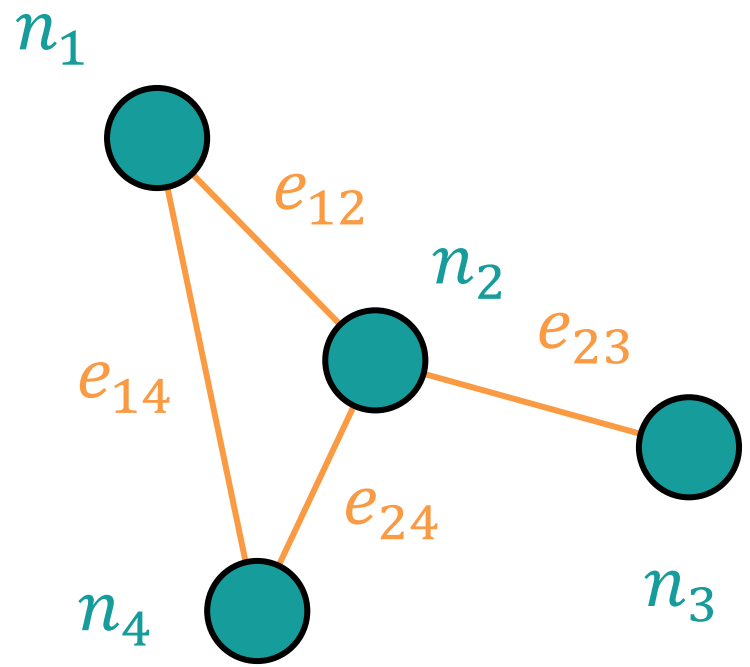
or

$$\mathbf{Gf} = \mathbf{\Phi}\,\mathrm{diag}(\langle \mathbf{g}, \mathbf{\Phi} \rangle)\,\mathbf{\Phi}^\dagger \mathbf{f}$$



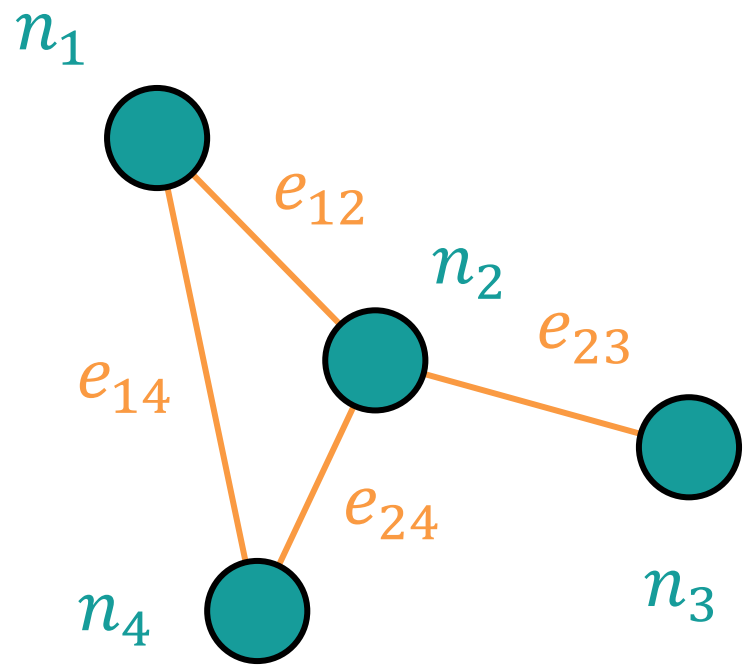
Connection to Graphs

- Graphs are discrete manifolds
- Manifold/graph defined by a set of nodes N and edges $\mathcal{E}$ (= tangent space)
- Gradient of a graph: $\nabla: N \rightarrow \mathcal{E}$
- Divergence: $\mathrm{div}: \mathcal{E} \rightarrow N$
- Laplacian: $\Delta: N \rightarrow N$
  - (Laplacian maps a graph to a new graph)



Connection to Graphs

- A node i carries an attribute a_i
- An edge connecting nodes i and j carries a weight w_{ij}
- Gradient: $(\nabla n)_{ij} = n_i - n_j$
- Divergence: $(\mathrm{div}\, e)_i = \frac{1}{a_i} \sum_j w_{ij}\, e_{ij}$
- Laplacian: $(\Delta n)_i = \frac{1}{a_i} \sum_j w_{ij} \left(n_j - n_i\right)$

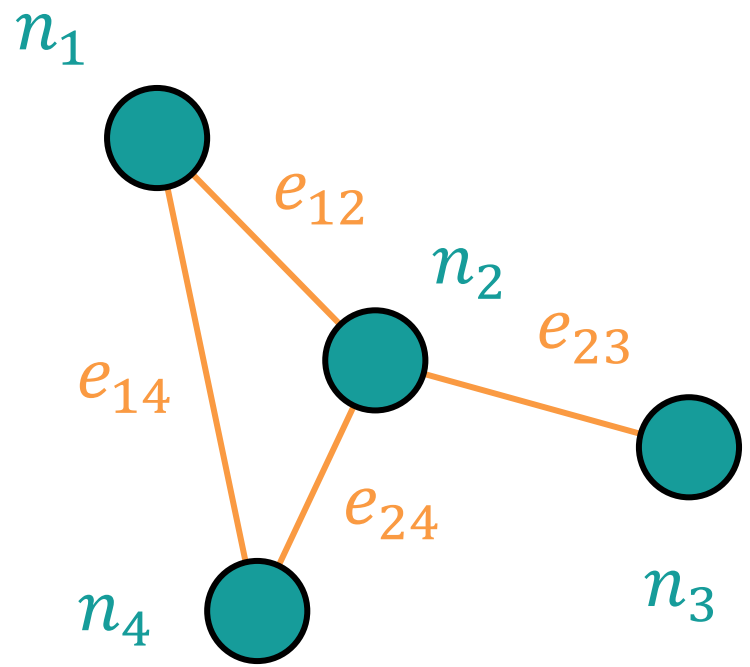


Connection to Graph NNs

- Laplacian: $(\Delta n)_i = \frac{1}{a_i}\sum_j w_{ij}\left(n_j - n_i\right)$
- "Normalised" Laplacian:

$$\mathbf{\Delta n} = \underbrace{\mathbf{A}^{-1}(\mathbf{D} - \mathbf{W})}_{\mathbf{\Delta}}\mathbf{n}$$

- Find eigenvectors $\mathbf{\Phi}$ of $\mathbf{\Delta}$, initialise convolution kernel $\mathbf{g}$, apply convolutions, update $\mathbf{g}$ with gradient descent...



Graph convolution operation:

$$\mathbf{Gn} = \mathbf{\Phi}\,\mathrm{diag}(\langle \mathbf{g}, \mathbf{\Phi} \rangle)\,\mathbf{\Phi}^{\mathrm{T}}\mathbf{n}$$

(see Bruna et al. (2013), *Spectral networks and locally connected networks on graphs*)

Why it doesn't work...

Input data

Data convolved with kernel

Same data convolved with same kernel (but different basis Φ)

Example from Masci et al. (2016), *Geometric Deep Learning*

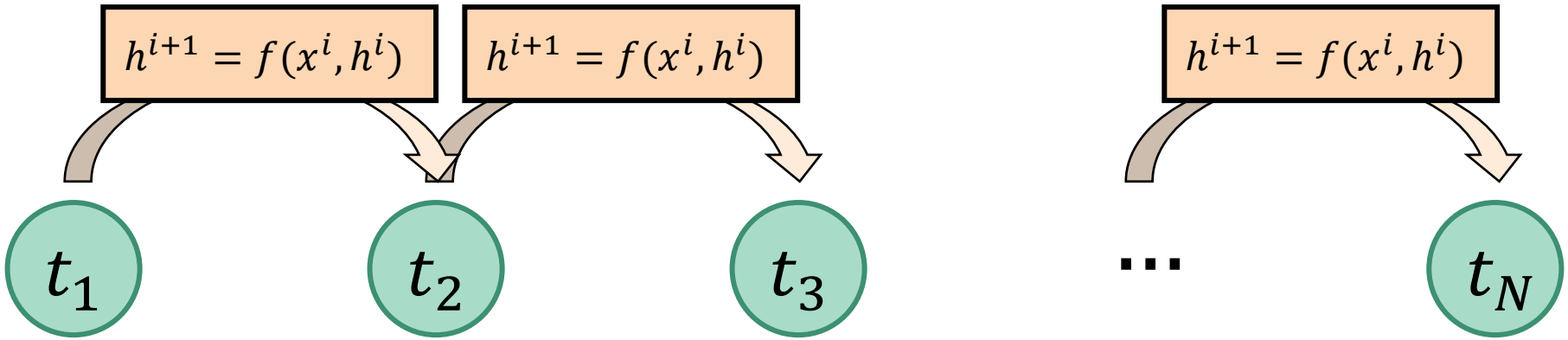
Why it doesn't work... (very technical...)

- Basis of Laplacian is generally non-unique (e.g. isometries)
- Learned kernel coefficients are basis-dependent
- Eigenproblem can be poorly conditioned (unstable), and back-propagation through this operation is inefficient
- Partial solution: spectrum-free convolution
  - A polynomial of Laplacians leads to a polynomial of eigenvalues
  - $\mathrm{diag}(\langle \mathbf{g}, \mathbf{\Phi} \rangle)$ can be written as a polynomial expansion of rank r, which acts as a "local" spectrum (at most r hops away from a given node)
  - Chebyshev polynomials are more stable: *ChebNet* (Defferrand et al., 2016)

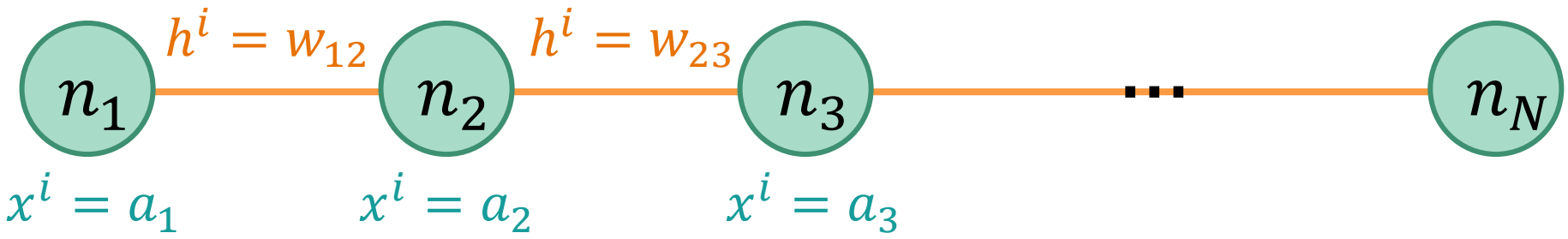
Alternative Approach

- In Euclidean spaces, one can choose to perform convolution in the spectral or the spatial/temporal domain.
  - I.e., $f * g = \mathbf{G}\mathbf{f} = \mathcal{F}^{-1}[\mathcal{F}(f)\mathcal{F}(g)]$, with $\mathbf{G}$ being a Toeplitz matrix
- Equivalent approach for graphs: *Message Passing*

Graphs and Recurrent Neural Networks



(Directed) chain graph:



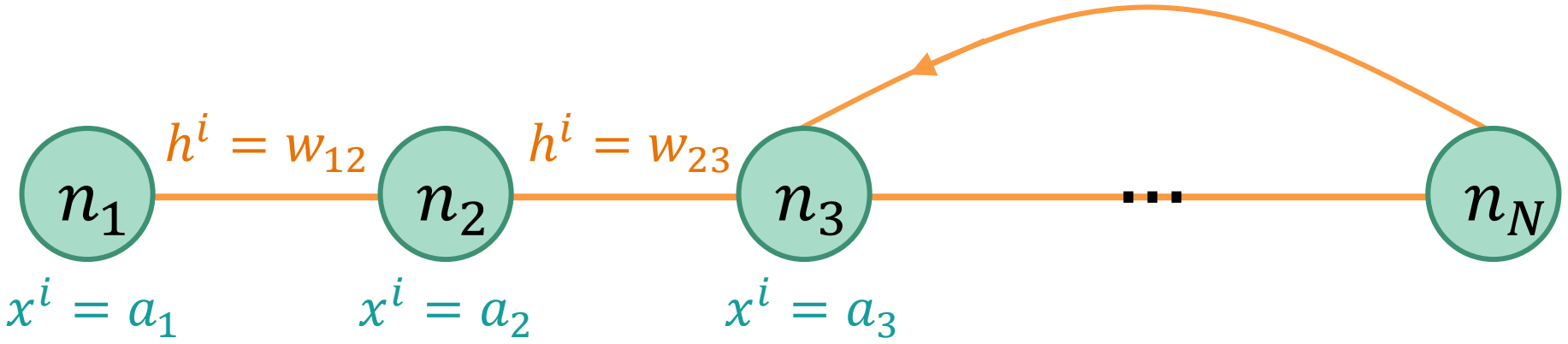
Graphs and Recurrent Neural Networks

Adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \mathbf{1} & 0 \end{bmatrix}$$



Message Passing Graph NNs

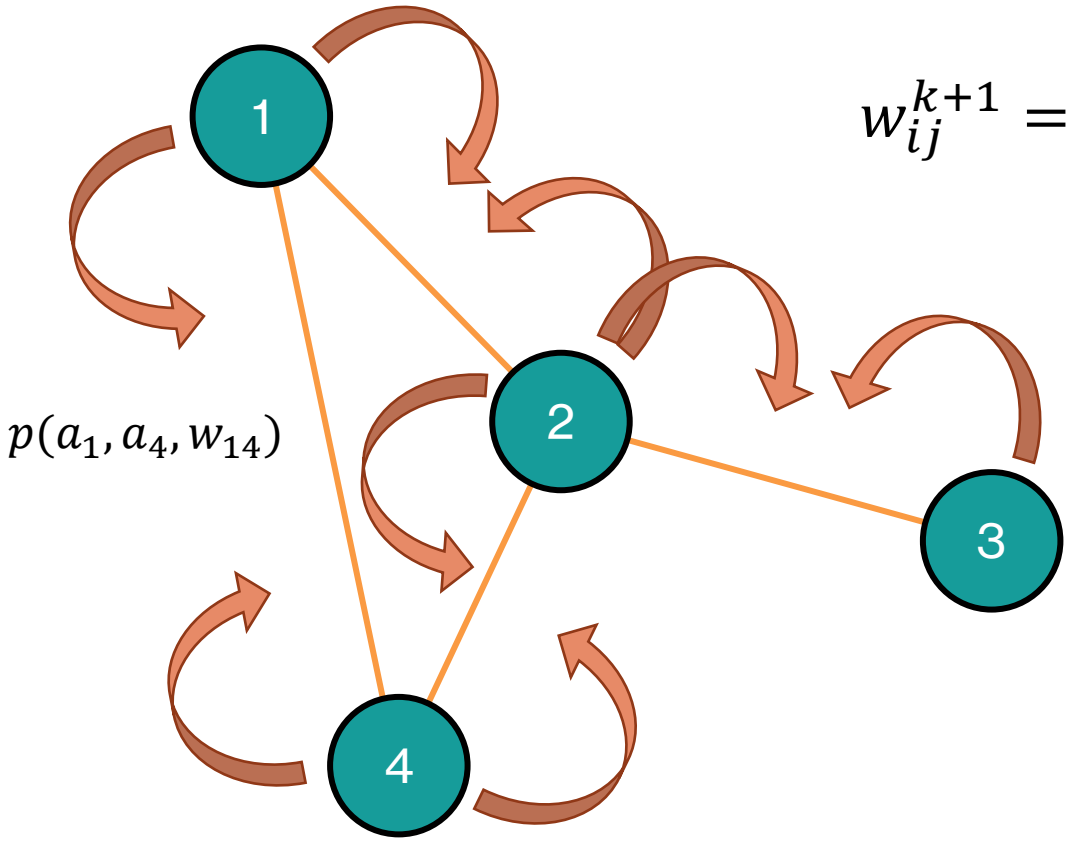
Extend RNNs (Gori et al., 2005) or LSTMs (Li et al., ICLR 2016) to arbitrary graphs:

1. At "time step" t_k, take attributes (a^k) of connected nodes and map them to their corresponding edges (= update edge weights w^{k+1})
2. For each node, aggregate the updated edge weights, and map them back onto the connected nodes (= update node attributes a^{k+1})
3. Repeat until a steady-state is achieved (Gori et al.) or for a fixed number of steps (Li et al.). Taking K steps implies that information can propagate a distance of K hops

Message Passing Graph NNs

Map $p: N \rightarrow \mathcal{E}$

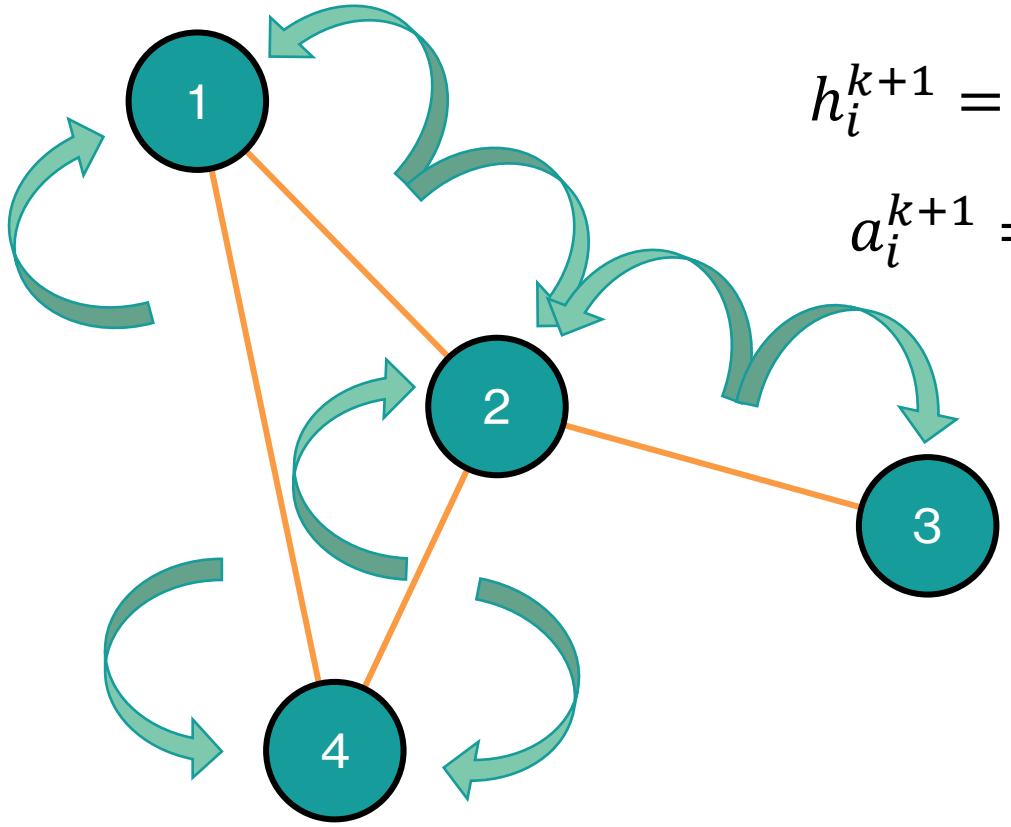
$$w_{ij}^{k+1} = p(a_i^k, a_j^k, w_{ij}^k)$$



Message Passing Graph NNs

Map $q: \mathcal{E} \to N$

$q(a_1, s(\{w_{12}, w_{14}\}))$



$$h_i^{k+1} = s\left(\left\{w_{ij}^{k+1}\right\}_{j \in J}\right)$$

$$a_i^{k+1} = q(a_i^k, h_i^{k+1})$$

Message Passing Graph NNs

Important: the functions p, q, s are the same everywhere on the graph

Analogy with Euclidean CNNs: the same kernel is used everywhere in the image (same parameters everywhere)

Message Passing Graph NNs

Algorithm 1 Steps of computation in a full GN block.

function GRAPHNETWORK(E, V, $\mathbf{u}$)
  for $k \in \{1 \dots N^e\}$ **do**
    $\mathbf{e}'_k \leftarrow \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right)$ ▷ 1. Compute updated edge attributes
  end for
  for $i \in \{1 \dots N^n\}$ **do**
    let $E'_i = \left\{\left(\mathbf{e}'_k, r_k, s_k\right)\right\}_{r_k=i,\ k=1:N^e}$
    $\bar{\mathbf{e}}'_i \leftarrow \rho^{e\rightarrow v}\left(E'_i\right)$ ▷ 2. Aggregate edge attributes per node
    $\mathbf{v}'_i \leftarrow \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right)$ ▷ 3. Compute updated node attributes
  end for
  let $V' = \left\{\mathbf{v}'\right\}_{i=1:N^v}$
  let $E' = \left\{\left(\mathbf{e}'_k, r_k, s_k\right)\right\}_{k=1:N^e}$
  $\bar{\mathbf{e}}' \leftarrow \rho^{e\rightarrow u}\left(E'\right)$ ▷ 4. Aggregate edge attributes globally
  $\bar{\mathbf{v}}' \leftarrow \rho^{v\rightarrow u}\left(V'\right)$ ▷ 5. Aggregate node attributes globally
  $\mathbf{u}' \leftarrow \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right)$ ▷ 6. Compute updated global attribute
  return $\left(E', V', \mathbf{u}'\right)$
end function

Algorithm compiled by Battaglia et al. (2018), *Relational inductive biases, deep learning, and graph networks*

Equivalence Spectral and Message Passing

- Message passing from nodes to edges: gradient operation $(\nabla n)_{ij}$
- Aggregation from edges to nodes: divergence operation $(\mathrm{div}\, e)_i$
- A single Message Passing "time step": $(\mathrm{div}\, \nabla n)_i \equiv (\Delta n)_i$ → Laplacian
- Formal equivalence between Message Passing and "spectrum-free" graph convolutions was demonstrated by Gilmer et al. (ICML 2017)
- Connection with the theory underlying generalised Fourier Transforms is gone, but the desired properties (stability, tractability, locality, ...) remain

Concrete example

- Earthquake catalogue with each earthquake representing a node in a graph, with attributes $a_i = (x, t)_i$ (hypocentre location, and origin time) and labels M_i (magnitude)
- Create edges between nodes if two earthquakes occur within a certain radius and time span of each other ($\|x_i - x_j\| < x_c$, $\|t_i - t_j\| < t_c$)
- Define for the n-th model layer:
  - Edge update function (perceptron / "fully-connected"): $w_{ij}^{n+1} = p_n(\dots) = \mathrm{MLP}(a_i^n, a_j^n, w_{ij}^n)$
  - Aggregation function: $h_i^{n+1} = s_n(\dots) = \max\left\{w_{ij}^{n+1}\right\}_{j \in J}$
  - Node update function: $a_i^{n+1} = q_n(\dots) = \mathrm{MLP}(a_i^n, h_i^{n+1})$
  - Graph layer: $\ell_n = q_n \cdot s_n \cdot p_n$
- Define $\mathrm{GNN} = \ell_N \cdot \ell_{n-1} \dots \ell_0 \cdot \mathbf{a}^0$, and learning objective $E\left[(M_i - \mathrm{GNN}((x, t)_i)^2\right]$ (i.e., predict earthquake magnitude from space-time relationships with regional seismicity)

Other considerations

1. Target labels can be assigned to nodes or to edges. Node attributes represent features in the graph, edge weights represent relationships between the features
   - **Example**: a complicated velocity model can be represented by (x, y, z) location attributes on the nodes and Δt_{ij} travel time "weights" on edges connecting nodes. Travel-time tomography now becomes a shortest-path problem on a graph, instead of solving a suitable wave equation.
2. Graphs can have "global" attributes that are assigned directly to the graph rather than to individual nodes/edges
   - **Example**: the source properties of a single earthquake (global attribute) as inferred from waveforms recorded by a seismic network (node attributes)
3. Creating a graph with all-to-all connectivity allows information to propagate over the entire graph in a single step, but it has a much higher computational complexity than implementing multiple graph convolution layers in sequence. Also, deeper models are more expressive...

Pytorch Geometric

github.com/pyg-team/pytorch_geometric

PyG

Edge from node 0 to node 1

Edge from node 1 to node 0

Edge from node 1 to node 2

Edge from node 2 to node 1

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1, 1, 2],
                           [1, 0, 2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
>>> Data(edge_index=[2, 4], x=[3, 1])
```

Adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Attributes for nodes 0, 1, 2

Optionally set other quantities like `edge_attr=` for edge weights and `y=` for labels

Example from https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html

Pytorch Geometric

github.com/pyg-team/pytorch_geometric

PyG

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = GCNConv(dataset.num_node_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)
```

First-order spectral implementation of Kipf & Welling (ICLR, 2017)

Edge weights is an optional argument (set to 1 otherwise)

Example from https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html

Pytorch Geometric

github.com/pyg-team/pytorch_geometric

PyG

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCN().to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
```

Example from https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html

Pytorch Geometric

github.com/pyg-team/pytorch_geometric

PyG

```python
from torch_geometric.nn import MessagePassing

x = ...           # Node features of shape [num_nodes, num_features]
edge_index = ...  # Edge indices of shape [2, num_edges]

class MyConv(MessagePassing):
    def __init__(self):
        super().__init__(aggr="add")

    def forward(self, x, edge_index):
        return self.propagate(edge_index, x=x)

    def message(self, x_i, x_j):
        return MLP(x_j - x_i)
```

$$x_i^{k+1} = \sum_{j \in J} \mathrm{MLP}(x_j^k - x_i^k)$$

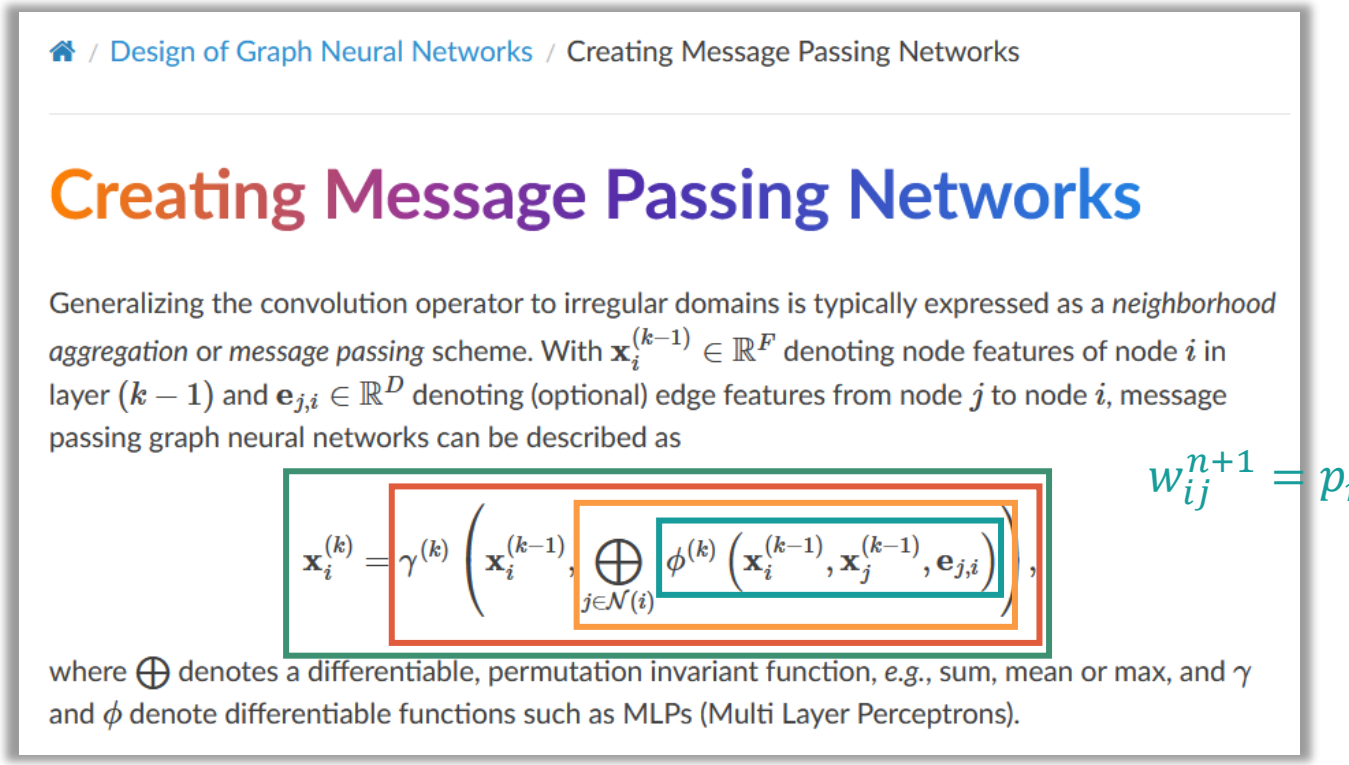
See https://pytorch-geometric.readthedocs.io/en/latest/tutorial/create_gnn.html

Pytorch Geometric

github.com/pyg-team/pytorch_geometric

PyG

$$\ell_n = q_n \cdot s_n \cdot p_n$$



Design of Graph Neural Networks / Creating Message Passing Networks

Creating Message Passing Networks

Generalizing the convolution operator to irregular domains is typically expressed as a *neighborhood aggregation* or *message passing* scheme. With $\mathbf{x}_i^{(k-1)} \in \mathbb{R}^F$ denoting node features of node i in layer $(k-1)$ and $\mathbf{e}_{j,i} \in \mathbb{R}^D$ denoting (optional) edge features from node j to node i, message passing graph neural networks can be described as

$$\mathbf{x}_i^{(k)} = \gamma^{(k)}\left(\mathbf{x}_i^{(k-1)}, \bigoplus_{j \in \mathcal{N}(i)} \phi^{(k)}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)\right),$$

where $\bigoplus$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ and ϕ denote differentiable functions such as MLPs (Multi Layer Perceptrons).

$$w_{ij}^{n+1} = p_n(\dots) = \mathrm{MLP}(a_i^n, a_j^n, w_{ij}^n)$$

$$a_i^{n+1} = q_n(\dots) = \mathrm{MLP}(a_i^n, h_i^{n+1})$$

$$h_i^{n+1} = s_n(\dots) = \max\left\{w_{ij}^{n+1}\right\}_{j \in J}$$

See https://pytorch-geometric.readthedocs.io/en/latest/tutorial/create_gnn.html

Take-home messages

1. Graph NNs represent a generalisation of traditional Euclidean learning methods (CNNs, RNNs, transformers, ...)

2. Don’t force your non-Euclidean data (seismic networks, point clouds, earthquake catalogues) into a Euclidean framework. Design an appropriate GNN instead

3. Spectral methods have a nice theoretical basis but suffer from significant computational drawbacks. Message Passing methods work better in practice

Further reading

- Bronstein et al. (2017), *Geometric Deep Learning: Going beyond Euclidean data*, https://doi.org/10.1109/MSP.2017.2693418 (introductory)
- Masci et al. (2016), *Geometric Deep Learning* (lecture notes), https://doi.org/10.1145/2988458.2988485 (more in-depth)
- Battaglia et al. (2018), Relational inductive biases, deep learning, and graph networks, https://doi.org/10.48550/arXiv.1806.01261 (introductory review of graph learning methods)
- PyTorch Geometric documentation: https://pytorch-geometric.readthedocs.io/en/latest/
- JAX GNN library (Jraph): https://github.com/google-deepmind/jraph

Fourier Transform and Laplacians

Define the Laplacian operator as the divergence of the gradient:

$$\Delta: L(X) \to L(X), \qquad \Delta f = -\mathrm{div}(\nabla f)$$

Dirichlet energy (measure of smoothness):

$$E = \oint \|\nabla f(x)\|^2 \mathrm{d}x = \oint f(x)\Delta f(x)\mathrm{d}x$$

Fourier Transform and Laplacians

1. Basis of Fourier Transform is spanned by the smoothest functions that can represent a given function (operator)
2. Smoothness is measured by the Dirichlet energy
3. Dirichlet energy of a manifold involves the integral over the Laplacian
4. Discrete sampling of manifold (= matrix): basis of the FT $\mathbf{\Phi}_k$ is spanned by the eigenvectors of the Laplacian $\mathbf{\Delta}$

$$\min_{\mathbf{\Phi}_k} \operatorname{trace}\left(\mathbf{\Phi}_k^{\mathrm{T}} \mathbf{\Delta} \mathbf{\Phi}_k\right) \quad \text{s.t. } \mathbf{\Phi}_k^{\mathrm{T}} \mathbf{\Phi}_k = \mathbf{I} \quad \rightarrow \quad \mathbf{\Delta} \mathbf{\Phi}_k = \mathbf{\Phi}_k \mathbf{\Lambda}_k$$

Minimise Dirichlet energy

Constraint: Orthonormal basis

Eigendecomposition of Laplacian

Key message

Fourier Transform of a manifold is defined by the eigenvectors of its Laplacian