Distributed Multi-Agent Coordination over Cellular Sheaves

An Expository Introduction to Sheaf Laplacians, Homological Programs, ADMM, and Sheaf Neural Networks

Eric Schmid

April 2025

Abstract

This expository paper provides a self-contained introduction to the theory and applications of *cellular sheaves* and *sheaf Laplacians* (both linear and nonlinear) in the context of multi-agent coordination. We survey the mathematical foundations of sheaves on graphs, define the classical and sheaf Laplacian operators, and then explain a recently proposed framework for heterogeneous multi-agent coordination via *nonlinear homological programs* on cellular sheaves. The paper details how distributed optimisation problems for coordination can be formulated in this topological framework and solved using the Alternating Direction Method of Multipliers (ADMM) in a decentralised fashion. We provide several illustrative examples (consensus, formation, flocking) to demonstrate the framework. We then draw connections to emerging concepts in graph machine learning, showing how the sheaf-based coordination dynamics can be interpreted as a form of *sheaf neural network*. In particular, we discuss how cellular sheaves enable modelling of asymmetric interactions and nonlinear diffusion processes in learning, leading to advantages in heterogeneous and decentralised learning settings. The exposition aims to be accessible to readers with a background in linear algebra and graph theory.

Contents

1 Background and Motivation

Multi-agent systems—collections of autonomous agents that must coordinate their actions—are frequently modeled using graphs. In a graph-based model, each agent is represented as a node and pairwise interactions or constraints between agents are represented as edges. Such graph models underpin many classical coordination algorithms: for instance, consensus protocols, distributed optimization, formation control, and flocking behaviors all typically assume an underlying communication graph among agents. Graph abstractions have proven effective for relatively homogeneous systems and simple interaction rules.

However, as we push toward more complex and heterogeneous multi-agent systems, graph models alone encounter limitations. A simple graph can only indicate which agents are connected, not the detailed *type* of interaction or constraint on each connection. In many scenarios we need to encode high-dimensional data at nodes (agent states) and more complex relationships along edges (e.g. transformations, constraints, or tasks that relate the data of two agents). For example, in a team of heterogeneous robots, an edge between a ground robot and an aerial drone might carry a different kind of relationship (say, a sensor alignment or a communication relay constraint) than an edge between two ground robots. A standard graph with scalar edge weights cannot fully capture this difference in interaction structure.

To address these challenges, recent research has proposed using **cellular sheaves** to generalize graph-based coordination models. A cellular sheaf is a construct from algebraic topology that can be thought of as a graph augmented with structured data attached to nodes and edges, together with consistency maps that govern the relationships between node data and edge data. This topological framework allows encoding of richer relational structure among agents and subsystems than a simple graph alone can accommodate. By using sheaves, one can incorporate vector spaces of states at each node (potentially of varying dimensions for different agents) and encode constraints or transformation laws on each edge via linear maps between these vector spaces. Intuitively, the sheaf assigns local data spaces and specifies how local sections of data must agree on overlaps (edges), thereby generalizing the notion of "consensus" to more complex consistency conditions.

The motivation for studying multi-agent coordination on sheaves is to enable advanced decentralized coordination in modern applications like drone swarms, sensor networks, and heterogeneous robotic teams, where agents might have different roles and capabilities. In such settings, enforcing coordination involves maintaining certain consistency or alignment between high-dimensional state information, not just matching scalar values. The sheaf-theoretic approach provides a language to rigorously describe these consistency conditions and to design algorithms that ensure they are met. Recent work by Hanks *et al.* introduced a general framework for heterogeneous multi-agent coordination using cellular sheaves and a new concept called the **nonlinear sheaf Laplacian**, which extends the familiar graph Laplacian to the sheaf setting and incorporates nonlinear interaction potentials [1].

In parallel, Riess [2] explored a related line of inquiry in his thesis on lattice-theoretic multi-agent systems, developing a *discrete Hodge theory* for certain lattice-valued sheaves, including a Laplace operator (dubbed the *Tarski Laplacian*) that generalizes the graph Laplacian to ordered or non-linear data domains[2]. These efforts reflect a broader trend of leveraging topological meth-

[1] Tyler Hanks et al., "Distributed Multi-Agent Coordination over Cellular Sheaves," *arXiv preprint arXiv:2504.02049* (2025). This paper introduces the cellular sheaf framework for coordination and the nonlinear sheaf Laplacian, using it to unify tasks like consensus, formation, and flocking under a single optimization framework.

[2] Hans Riess, *Lattice Theory in Multi-Agent Systems* (Ph.D. thesis, University of Pennsylvania, 2023). Riess develops a Laplacian for lattice-valued (order-theoretic) sheaves and proves a Hodge-type theorem (the "Hodge-Tarski

ods (sheaves, cohomology, Hodge theory) in the analysis of complex networks and coordination problems.

The use of cellular sheaves in multi-agent coordination promises:

- **Richer Modeling Capability:** We can attach vector spaces (state spaces) to each agent and each interaction, allowing heterogeneity in what information is carried by different parts of the network. Constraints between agents can be represented by linear maps on these vector spaces, capturing directionality or transformations in the interaction.

- **Generalized Consensus and Coordination:** Classical consensus requires all agents' states (often scalars) to agree. Using sheaves, one can enforce more general consistency conditions (for instance, agents' states might need to satisfy a linear equation or align through a rotation matrix on each edge) rather than simple equality. This broadens the type of coordination objectives we can formalize.

- **Decentralized Algorithms:** The sheaf formalism, combined with tools like the sheaf Laplacian, yields natural generalizations of diffusion or gossip algorithms. As we will see, a *sheaf Laplacian flow* can drive the system toward satisfying edge consistency constraints just as the graph Laplacian flow drives a network to consensus. Moreover, by formulating coordination as an optimization problem (a *homological program*) on the sheaf, we can apply powerful distributed optimization methods (like ADMM) to solve it.

- **Connections to Learning:** Interestingly, these sheaf-based processes have analogies to recent Graph Neural Network (GNN) architectures. In a GNN, each layer updates node feature vectors by aggregating transformed neighbor features. In a sheaf-based coordination algorithm, each iteration updates agent states using neighbors' information and certain linear maps (the sheaf restrictions). This parallel suggests that *sheaf neural networks* could harness the additional structure of sheaves to improve learning on graphs with complex relationships[3].

The remainder of this paper is organized as follows. In Section 2, we introduce the key mathematical concepts: graphs, cellular sheaves on graphs, and sheaf Laplacians. We provide definitions and basic properties, drawing on established literature for a gentle introduction. Section 3 revisits the classical graph Laplacian and consensus dynamics, to build intuition for the sheaf generalization. Section 4 then presents the sheaf Laplacian in more detail and interprets its effect as a decentralized flow for enforcing consistency (a generalized consensus). Section 5 lays out a general framework for formulating multi-agent coordination problems using sheaves, including how to encode various coordination tasks. Section 6 summarizes the main contributions and benefits of this framework in comparison to classical approaches. In Section 7, we formally define the *homological program* for multi-agent coordination, which is an optimization problem set up on a cellular sheaf. Section 8 details how to solve such programs using ADMM in a distributed manner; we derive the ADMM algorithm step by step and discuss its convergence. Section 9 draws a connection between the iterative solution (which can be seen as message-passing on the network) and neural network architectures, effectively viewing each iteration as a "layer" of a sheaf neural network. Based on this view, Section 10 discusses the advantages of incorporating sheaves into neural network models

theorem") relating its fixed points to global sections of the sheaf. Chapter 5 of his thesis provides an introduction to sheaf theory from a combinatorial perspective, motivating the use of sheaves in multi-agent systems.

[3]See, e.g. "Sheaf Neural Networks," *arXiv:2012.06333* (2020), and Cristian Bodnar et al., "Neural Sheaf Diffusion: A Topological Perspective on Heterophily and Oversmoothing," *International Conference on Learning Representations (ICLR)* 2023. These works introduce neural network layers based on sheaf Laplacians, demonstrating advantages in handling directed or signed relationships (heterophily) on graphs.

for learning, such as the ability to handle asymmetric relations and nonlinear propagation. Finally, we conclude with a brief discussion of future directions and applications.

2 Key Concepts and Mathematical Foundations

2.1 Graphs and Data on Graphs

We begin by reviewing the notion of a graph and the idea of attaching data to a graph. For our purposes, a graph G is a pair (V, E) consisting of a set of vertices (or nodes) V and a set of edges E. Throughout, we will consider undirected graphs (suitable for modeling bidirectional communication between agents), and we assume G is simple (no multiple edges between the same pair of vertices, and no self-loops). An edge $e \in E$ connecting vertices i and j will be denoted $e = \{i, j\}$ or simply i–j. We write $i \sim j$ if $\{i, j\} \in E$. Each vertex i has a set of neighbors $N_i = \{\, j \in V : \{i, j\} \in E \,\}$.

In many applications, one associates data to the nodes or edges of a graph. For example, in a multi-agent system, each node i might have an associated state vector $x_i \in \mathbb{R}^k$ describing the agent's internal variables (position, velocity, etc.), and each edge $e = \{i, j\}$ might represent a relationship or measurement involving agents i and j.

In the context of cellular sheaves on graphs, we can formalize these data assignments using terminology adapted from algebraic topology:

- A 0-*cochain* on G is an assignment of a vector to each vertex. If each vertex i is assigned a vector $x_i \in \mathbb{R}^k$, we view $x = \{x_i\}_{i \in V}$ as a 0-cochain (with values in $\mathbb{R}^k$).

- A 1-*cochain* on G is an assignment of a vector to each edge. For instance, $y = \{y_e\}_{e \in E}$ with $y_e \in \mathbb{R}^m$ for each edge e would be a 1-cochain (with values in $\mathbb{R}^m$).

If the vector spaces (like $\mathbb{R}^k$ or $\mathbb{R}^m$) are the same for all vertices or all edges, we call this a *constant* assignment of data dimensions. In more general cases, one might have different types or dimensions of data at different nodes or edges. The mathematical structure that allows this generality (varying data spaces attached to different parts of a graph) is precisely a *sheaf*, which we define next.

2.2 Cellular Sheaves on Graphs

Informally, a sheaf can be thought of as a way to attach data (or data spaces) to each part of a space (in our case, to each vertex and edge of a graph) along with consistency requirements that specify how data on neighboring parts must relate. While sheaves originated in algebraic geometry and topology for continuous spaces, here we focus on *cellular sheaves on graphs*, which are a discrete version suitable for network systems [4].

[Cellular Sheaf on a Graph] Let $G = (V, E)$ be a graph. A **sheaf $\mathcal{F}$ on** G (sometimes called a *cellular sheaf* when G is viewed as a 1-dimensional cell complex) consists of the following:

- For each vertex $i \in V$, a vector space $\mathcal{F}(i)$ over some field (here we take all vector spaces to be real Euclidean spaces). We call $\mathcal{F}(i)$ the *stalk* of the sheaf at vertex i. Intuitively, $\mathcal{F}(i)$ is the data space associated with node i.

- For each edge $e = \{i, j\} \in E$, a vector space $\mathcal{F}(e)$ (the stalk at e). This is the data space associated with the interaction or relation between i and j.

[4] For an accessible introduction to cellular sheaves on graphs, see Jakob Hansen, "A gentle introduction to sheaves on graphs" (2020), available as a preprint. This source provides definitions and simple examples, oriented towards applications in engineering and network science.

- For each incident pair (i, e) with $i \in V$, $e \in E$, and i incident to e (meaning i is an endpoint of edge e), a linear map
$$\mathcal{F}_{i \to e} : \mathcal{F}(i) \ \to \ \mathcal{F}(e),$$
called a *restriction map*. If $e = \{i, j\}$, there are two such maps: $\mathcal{F}_{i \to e}$ mapping data from node i to the edge e, and $\mathcal{F}_{j \to e}$ mapping data from node j to the edge e. These maps specify how the local data at a node should be transformed or projected to be compatible with the data on the connecting edge.

A sheaf $\mathcal{F}$ assigns a vector space to each vertex and edge of G, and assigns to each inclusion of a vertex in an edge a linear map between the corresponding vector spaces.

We often denote a sheaf by listing the data assignments, e.g. $\mathcal{F}(i)$ for vertices and $\mathcal{F}(e)$ for edges, with the understanding that the restriction maps $\mathcal{F}_{i \to e}$ are part of the data of the sheaf. The collection $\{\mathcal{F}_{i \to e}\}$ encodes the **consistency requirements** of the sheaf: a section of the sheaf (an assignment of data to every vertex and edge) is "valid" (a *global section*) if whenever a vertex i and an incident edge e are considered, the data on i matches the data on e via the restriction map. Formally, a **global section** of $\mathcal{F}$ is a pair of families $(\{x_i \in \mathcal{F}(i)\}_{i \in V}, \{y_e \in \mathcal{F}(e)\}_{e \in E})$ such that for every edge $e = \{i, j\}$, we have

$$\mathcal{F}_{i \to e}(x_i) = y_e \quad \text{and} \quad \mathcal{F}_{j \to e}(x_j) = y_e.$$

In other words, the data y_e on edge e is the image of the data x_i on vertex i under the map to the edge, and similarly from the j side. Equivalently $\mathcal{F}_{i \to e}(x_i) = \mathcal{F}_{j \to e}(x_j)$ for all edges $e = \{i, j\}$. This is the **local consistency condition** enforced by the sheaf.

If a choice of vertex data $\{x_i\}$ can be extended to some edge data $\{y_e\}$ satisfying these conditions, then $\{x_i\}$ is said to be a *section at the vertices* (or 0-cochain) that is consistent. If no such $\{y_e\}$ exists, then $\{x_i\}$ has some inconsistency across at least one edge.

[Constant Sheaf] A basic example is the **constant sheaf** on G with fiber $\mathbb{R}^k$. This sheaf, denoted perhaps $\underline{\mathbb{R}^k}$, is defined by taking $\mathcal{F}(i) = \mathbb{R}^k$ for every vertex i and $\mathcal{F}(e) = \mathbb{R}^k$ for every edge e, and letting each restriction map $\mathcal{F}_{i \to e}$ be the identity map on $\mathbb{R}^k$. In words, each node and each edge hold a k-dimensional vector, and the consistency condition is that for an edge $e = \{i, j\}$, the vectors at i, j, and e must all agree (since $\mathcal{F}_{i \to e}$ and $\mathcal{F}_{j \to e}$ just send a vector to itself). A global section of the constant sheaf is thus an assignment of a vector $x_i \in \mathbb{R}^k$ to each node such that $x_i = x_j$ for every edge $\{i, j\}$. If the graph G is connected, the only global sections are those where all nodes have the same vector (i.e. $x_1 = x_2 = \cdots = x_n$), which is exactly the usual consensus condition.

[Orientation Sheaf or Sign Sheaf] Another instructive example is a sheaf that assigns $\mathbb{R}$ to each vertex and edge (so scalar data everywhere), but where the restriction maps might be ± 1 instead of always $+1$. For instance, suppose we have a sheaf $\mathcal{F}$ with $\mathcal{F}(i) = \mathbb{R}$ for all i, $\mathcal{F}(e) = \mathbb{R}$ for all e, and define

$$\mathcal{F}_{i \to e} = 1 \quad \text{and} \quad \mathcal{F}_{j \to e} = -1$$

for an edge $e = \{i, j\}$. This means from the perspective of agent i, the edge's value should equal x_i, but from the perspective of agent j, the edge's value should equal *minus* x_j. The consistency condition becomes $y_e = x_i = -x_j$. If we interpret x_i as some scalar measurement at node i, this sheaf is encoding that measurements at i and j should be negatives of each other across the edge e. This could represent, e.g., a scenario where two agents must hold opposite voltages or opposite opinions. A global section here means an assignment of scalars to all nodes such that whenever i and j are connected, $x_j = -x_i$. On a graph that is a single edge with two vertices, the global sections are those pairs (x_1, x_2) where $x_2 = -x_1$. On a triangle graph, a global section

would require $x_1 = -x_2, x_2 = -x_3, x_3 = -x_1$, which forces $x_1 = -x_2, x_2 = -x_3, x_3 = x_1$ (by substituting), implying $x_1 = -x_2 = x_3 = -x_1$, so x_1 must be zero; thus only the zero assignment is a global section in that case (the sheaf was too restrictive to have a non-zero global solution on a cycle of odd length).

These examples illustrate that by adjusting the restriction maps, we can enforce different kinds of consistency: exact equality (as in the constant sheaf), equality up to a sign (orientation sheaf), or in general equality up to some linear transformation. In the most general case, if $\mathcal{F}_{i\to e} : \mathcal{F}(i) \to \mathcal{F}(e)$ is not surjective, the condition $\mathcal{F}_{i\to e}(x_i) = \mathcal{F}_{j\to e}(x_j)$ can be thought of as a system of linear equations relating x_i and x_j. Sheaf theory provides a language to talk about solutions to all these local constraints collectively.

Before proceeding, we introduce some notation from algebraic topology that is useful to work with sheaves algebraically:

- A **0-cochain** valued in the sheaf $\mathcal{F}$ is a choice of a vector $x_i \in \mathcal{F}(i)$ for each vertex i. We denote the space of all 0-cochains by

  $$C^0(G;\mathcal{F}) := \bigoplus_{i\in V} \mathcal{F}(i).$$

  This is essentially the direct product of all node stalks, i.e. the space of all assignments of node data.

- A **1-cochain** valued in $\mathcal{F}$ is a choice of a vector $y_e \in \mathcal{F}(e)$ for each edge e. The space of 1-cochains is

  $$C^1(G;\mathcal{F}) := \bigoplus_{e\in E} \mathcal{F}(e).$$

- The **coboundary operator** (or restriction map at the level of cochains) is a linear map

  $$\delta_{\mathcal{F}} : C^0(G;\mathcal{F}) \to C^1(G;\mathcal{F}),$$

  defined by how it acts on a 0-cochain $x = \{x_i\}_{i\in V}$. For each edge $e = \{i, j\}$, the edge component of $\delta_{\mathcal{F}} x$ is given by

  $$(\delta_{\mathcal{F}} x)_e = \mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j).$$

  Here we must choose an orientation for each undirected edge to give a sign (we can pick an arbitrary orientation for each edge for the sake of this definition; the resulting $\delta_{\mathcal{F}}$ will depend on that choice but different choices are isomorphic). Intuitively, $\delta_{\mathcal{F}} x$ measures the *disagreement* between the data at i and j when projected onto the edge e.

With this operator, the condition for x to be extendable to a global section (with some y on edges) is precisely $\delta_{\mathcal{F}} x = 0$. If $\delta_{\mathcal{F}} x = 0$, we say x is a **cocycle** or that x is *closed* (borrowing terminology from cohomology theory). In plain terms, $\delta_{\mathcal{F}} x = 0$ means that for every edge $e = \{i, j\}$, $\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j) = 0$, i.e., $\mathcal{F}_{i\to e}(x_i) = \mathcal{F}_{j\to e}(x_j)$. So x is a section at the vertices that is consistent on all edges. Such an x exactly corresponds to a global section of the sheaf (taking $y_e = \mathcal{F}_{i\to e}(x_i)$ for each edge e yields a globally consistent assignment).

We call 1-cochains in the image of $\delta_{\mathcal{F}}$ *exact* 1-cochains. These represent edge data that come from some assignment of node data. The kernel of $\delta_{\mathcal{F}}$ (i.e., all 0-cochains x such that $\delta_{\mathcal{F}} x = 0$) are the *closed* 0-cochains, which correspond to global sections. In cohomological terms, $H^0(G;\mathcal{F}) := \ker(\delta_{\mathcal{F}})$ is the space of global sections of the sheaf $\mathcal{F}$. Since there is no 2-cochain space in a graph,

the coboundary operator out of C^1 is zero, so its kernel is all of C^1. Therefore, $H^1(G;\mathcal{F}) := \frac{C^1(G;\mathcal{F})}{\mathrm{im}(\delta_{\mathcal{F}})}$ measures the obstructions to being a global section (i.e., it represents cohomology classes of edge assignments). We will not delve deeply into cohomology here, but this viewpoint is useful in understanding certain results like Hodge decompositions.

To summarize, a sheaf on a graph gives rise to a linear operator $\delta_{\mathcal{F}}$ that measures local inconsistencies. This is analogous to the incidence matrix of a graph in classical graph theory: for a simple graph (with trivial sheaf), δ corresponds to an oriented incidence matrix, and $\delta x = 0$ means a scalar function x on vertices is constant on each connected component. The sheaf case generalizes this by including linear transformations in the incidence relations.

2.3 Sheaf Laplacians: Linear Case

Given a sheaf $\mathcal{F}$ on G, we can construct an operator analogous to the graph Laplacian, but now acting on sections of the sheaf. The **sheaf Laplacian** is defined in a way similar to the combinatorial (graph) Laplacian, using the coboundary operator $\delta_{\mathcal{F}}$.

[Sheaf Laplacian] For a sheaf $\mathcal{F}$ on G, the (0-dimensional) **sheaf Laplacian** $L_{\mathcal{F}} : C^0(G;\mathcal{F}) \to C^0(G;\mathcal{F})$ is defined by

$$L_{\mathcal{F}} := \delta_{\mathcal{F}}^T \delta_{\mathcal{F}},$$

where $\delta_{\mathcal{F}}^T : C^1(G;\mathcal{F}) \to C^0(G;\mathcal{F})$ is the adjoint (transpose) of $\delta_{\mathcal{F}}$ with respect to the standard inner products on the Euclidean stalks (assuming each stalk $\mathcal{F}(i)$ and $\mathcal{F}(e)$ is an inner product space).

In coordinates, if we choose an orientation for each edge and bases for each stalk, $\delta_{\mathcal{F}}$ can be represented as a (matrix) operator mapping the vector of all node values to the vector of all edge differences. Then $L_{\mathcal{F}}$ is represented by the matrix $B^T B$ where B is the matrix of $\delta_{\mathcal{F}}$. Thus $L_{\mathcal{F}}$ is symmetric positive semi-definite by construction (since $B^T B$ always is). More concretely, $L_{\mathcal{F}}$ acts on a 0-cochain $x = \{x_i\}_{i \in V}$ as follows: the i-th component $(L_{\mathcal{F}} x)_i$ is

$$(L_{\mathcal{F}} x)_i = \sum_{e : e = \{i,j\} \in E} \mathcal{F}_{i \to e}^T \big(\mathcal{F}_{i \to e}(x_i) - \mathcal{F}_{j \to e}(x_j) \big).$$

This formula resembles the graph Laplacian in that it is a sum over neighbors j of differences $x_i - x_j$, but here each difference is mapped into the edge space via $\mathcal{F}_{i \to e}$, and then mapped back via the adjoint of $\mathcal{F}_{i \to e}$. The result $(L_{\mathcal{F}} x)_i$ lies in the space $\mathcal{F}(i)$ (the same space as x_i), as expected for a Laplacian on the node space.

A convenient block-matrix way to write $L_{\mathcal{F}}$ is to arrange all node variables x_i into a vector $x = (x_1; \ldots; x_{|V|})$ and similarly $y = (y_e)_{e \in E}$ for edge variables. Then $\delta_{\mathcal{F}}$ can be seen as a matrix with blocks $\mathcal{F}_{i \to e}$ and $-\mathcal{F}_{j \to e}$ in the rows corresponding to edge e and columns corresponding to vertices i and j.

For vertices $i, j \in V$, define the block entries of the Laplacian matrix as:

$$[L_{\mathcal{F}}]_{ij} = \begin{cases} \sum_{e \ni i} \mathcal{F}_{i \to e}^T \mathcal{F}_{i \to e} & \text{if } i = j, \\ -\mathcal{F}_{i \to e}^T \mathcal{F}_{j \to e} & \text{if } e = \{i, j\} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

This generalizes the familiar formula for the graph Laplacian matrix $L = D - A$. In the special case where each $\mathcal{F}_{i \to e}$ is the identity (and scalar), $\mathcal{F}_{i \to e}^T \mathcal{F}_{i \to e} = 1$ and $\mathcal{F}_{i \to e}^T \mathcal{F}_{j \to e} = 1$, making $[L_{\mathcal{F}}]_{ii} = \deg(i)$ and $[L_{\mathcal{F}}]_{ij} = -1$ for adjacent vertices i, j. Indeed, we have:

[Recovery of Graph Laplacian] If $\mathcal{F}$ is the constant sheaf on G with 1-dimensional stalks and identity restriction maps (as in Example 2.2), then the sheaf Laplacian $L_{\mathcal{F}}$ is exactly the ordinary graph Laplacian L_G on G. That is, $(L_{\mathcal{F}}x)_i = \sum_{j:j\sim i}(x_i - x_j)$ for $x \in C^0(G;\mathcal{F})$.

Proof. Under a trivial sheaf, for each edge $e = \{i, j\}$, $\mathcal{F}_{i\to e}$ and $\mathcal{F}_{j\to e}$ are the identity on $\mathbb{R}$, so $\mathcal{F}_{i\to e}^T$ is also identity. Then

$$(L_{\mathcal{F}}x)_i = \sum_{e=\{i,j\}} (1)^T(1 \cdot x_i - 1 \cdot x_j) = \sum_{j:j\sim i}(x_i - x_j),$$

which is precisely the definition of the graph Laplacian applied to the function $x : V \to \mathbb{R}$. In matrix terms, $L_{\mathcal{F}}$ has $L_{ii} = \sum_{e\sim i} 1 = \deg(i)$ and $L_{ij} = -1$ if $i \sim j$, matching the standard Laplacian matrix $L_G = D - A$. □

Thus, sheaf Laplacians are a true generalization of graph Laplacians. One can alternatively think of them as graph Laplacians with a possibly different weight matrix on each edge (represented by the linear maps $\mathcal{F}_{i\to e}$). In spectral graph theory, one sometimes encounters *matrix-weighted graphs* where an edge between i, j is associated with a weight matrix W_{ij} rather than a scalar weight. The sheaf Laplacian is essentially the Laplacian of a matrix-weighted graph, where $W_{ij} = \mathcal{F}_{i\to e}^T\mathcal{F}_{j\to e}$ for edge $e = \{i, j\}$. This viewpoint has been discussed in literature on generalized network Laplacians [5].

Several important properties of the graph Laplacian carry over to the sheaf Laplacian:

- $L_{\mathcal{F}}$ is symmetric positive semidefinite. Thus all its eigenvalues are $\lambda \geq 0$.
- The kernel of $L_{\mathcal{F}}$ (the nullspace) consists of all x such that $L_{\mathcal{F}}x = 0$. Because $L_{\mathcal{F}} = \delta_{\mathcal{F}}^T\delta_{\mathcal{F}}$, we have $x^T L_{\mathcal{F}}x = \|\delta_{\mathcal{F}}x\|^2$. Therefore, $L_{\mathcal{F}}x = 0$ if and only if $\delta_{\mathcal{F}}x = 0$.

  This means that the kernel of the sheaf Laplacian equals the kernel of the coboundary operator:

  $$\ker(L_{\mathcal{F}}) = \ker(\delta_{\mathcal{F}})$$

  As we've seen, $\ker(\delta_{\mathcal{F}})$ is precisely the space of global sections of the sheaf. For a connected graph with a constant sheaf, this means $\ker(L_G)$ is the one-dimensional space of constant vectors (the classical result for graph Laplacians). For a general sheaf, $\dim\ker(L_{\mathcal{F}})$ could be larger if the sheaf has multiple degrees of freedom for consistency. For example, the sign sheaf on a graph with an even cycle has a two-dimensional space of global sections.

The sheaf Laplacian $L_{\mathcal{F}}$ acts on 0-cochains (node assignments). There is also a *1-Laplacian* which would act on 1-cochains ($L_{\mathcal{F}}^{(1)} = \delta\,\delta^T$), but in this paper we primarily care about the 0-Laplacian because it governs how node values evolve to satisfy constraints. In the context of multi-agent coordination, we are typically dealing with states at nodes and constraints at edges, so $L_{\mathcal{F}}$ on nodes is the relevant operator.

Sheaf Laplacian Dynamics. A useful way to think about $L_{\mathcal{F}}$ is as a generator of a diffusion or consensus-like process. Consider the ODE:

$$\dot{x}(t) = -L_{\mathcal{F}}\,x(t), \tag{1}$$

[5] See, Hansen and Ghrist, "Learning Sheaf Laplacians from Smooth Signals," *Proc. ICASSP 2019*, where the goal is to infer sheaf restriction maps from data.

where $x(t) \in C^0(G;\mathcal{F})$ is a time-dependent assignment of states to nodes. This is a linear system of ODEs. Expanding $\dot{x}_i = -(L_{\mathcal{F}}x)_i$ using the earlier formula,

$$\dot{x}_i = -\sum_{e=\{i,j\}} \mathcal{F}_{i\to e}^T\big(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j)\big).$$

This can be interpreted as each agent i adjusting its state x_i in the direction that reduces the inconsistency on each incident edge. Specifically, for each neighbor j of i, the term $\mathcal{F}_{i\to e}^T(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j))$ is the "force" exerted on x_i due to the inconsistency between i and j's data when viewed on edge e. The sum over neighbors accumulates all these corrections. The negative sign indicates x_i moves opposite to the gradient of inconsistency (hence reducing the difference). This is precisely a generalized consensus algorithm: if x_i and x_j are not consistent according to the sheaf, they will exchange information to become more aligned.

Because $L_{\mathcal{F}}$ is positive semidefinite, one can show that $\frac{d}{dt}\|x(t)\|^2 = -x(t)^T L_{\mathcal{F}} x(t) = -\|\delta_{\mathcal{F}} x(t)\|^2 \leq 0$. Thus $\|x(t)\|^2$ is non-increasing.

By LaSalle's invariance principle [**?**], the system will converge to the largest invariant set where $\frac{d}{dt}\|x(t)\|^2 = 0$, which corresponds to the set where $\|\delta_{\mathcal{F}} x(t)\|^2 = 0$, or equivalently, $\delta_{\mathcal{F}} x(t) = 0$. This set is precisely $\ker(L_{\mathcal{F}})$, the space of global sections of the sheaf.

Therefore, for a connected graph and a sheaf with appropriate properties, the dynamics in equation (1) will converge to a global section $x(\infty)$ such that $L_{\mathcal{F}} x(\infty) = 0$, meaning all edge constraints are satisfied. The specific global section that the system converges to depends on the initial condition $x(0)$ and the dimension of $\ker(L_{\mathcal{F}})$[6].

In closing this subsection, we note that all the above has been the *linear* theory of sheaves and Laplacians. The restriction maps $\mathcal{F}_{i\to e}$ were linear and the Laplacian is a linear operator. This suffices for modeling tasks where the constraint relationship between agent states is linear. But many coordination tasks are inherently nonlinear (e.g., maintaining a formation at a certain distance involves a quadratic constraint on positions, flocking involves nonlinear dynamics). To capture that, we move next to the concept of a **nonlinear homological program**, which introduces nonlinear edge potentials, effectively creating a *nonlinear sheaf Laplacian* in the process.

2.4 Nonlinear Homological Programs on Sheaves

The term "homological program" comes from earlier work [**?**] where optimization problems were formulated in terms of homology/cohomology of a network (for sensor network coordination). Here we adopt the formulation by Hanks *et al.*[7] to define a general optimization problem that a multi-agent coordination task can be translated into. Roughly speaking, a *nonlinear homological program* consists of:

- A graph $G = (V, E)$ representing the communication topology among agents.
- A sheaf $\mathcal{F}$ on G which specifies the data (state) spaces for each agent and each interaction and how those should match up.

[6] If $\ker(L_{\mathcal{F}})$ has dimension greater than 1, there are multiple possible equilibrium points. A simple example is standard consensus, where the system converges to all nodes having the same value, but that common value depends on the initial average.

[7] Hanks et al., 2025 (see previous footnote). In their framework, a *nonlinear homological program* generalizes the earlier linear homological programs by allowing nonlinear objective and constraint functions while still leveraging the sheaf structure.

- Convex objective functions $f_i : \mathcal{F}(i) \to \mathbb{R} \cup \{+\infty\}$ for each node (these represent local costs for each agent's own state; they can enforce constraints by taking value $+\infty$ on disallowed states).
- Convex *potential functions* $U_e : \mathcal{F}(e) \to \mathbb{R} \cup \{+\infty\}$ for each edge (these represent the cost or constraint associated with the relation between the two endpoints' data when projected to the edge).

We allow extended real-valued objective and potential functions to handle constraints as well as costs (a common trick in convex optimization: a constraint $h(x) = 0$ can be enforced by adding an indicator function $I_{\{h(x)=0\}}(x)$ which is 0 if $h(x) = 0$ and $+\infty$ otherwise). In many cases, U_e will be something like a strongly convex function that has a unique minimizer at the "allowed" relative configuration of i and j's data, and f_i might be something like a regular quadratic penalty or other cost for agent i's state deviating from a preferred value.

Given this data, we can define the optimization problem:

$$\begin{aligned} \min_{\substack{x \in C^0(G;\mathcal{F}) \\ y \in C^1(G;\mathcal{F})}} \quad & \sum_{i \in V} f_i(x_i) + \sum_{e \in E} U_e(y_e) \\ \text{s.t.} \quad & \mathcal{F}_{i \to e}(x_i) = y_e, \quad \forall e = \{i, j\} \in E, \ \forall i \text{ endpoint of } e. \end{aligned} \tag{2}$$

Here the decision variables are $x = \{x_i\}_{i \in V}$ (node assignments) and $y = \{y_e\}_{e \in E}$ (edge assignments), and the constraints enforce that (x, y) forms a global section of the sheaf (i.e. that y_e equals the restriction of x on both endpoints of e). In other words, $y = \delta_{\mathcal{F}} x$ (with appropriate sign conventions, but effectively the constraint means $\delta_{\mathcal{F}} x = 0$ as in the linear case, except here we allow an y_e which is then also optimized). If we eliminated y, we could equivalently write the problem as:

$$\min_{x \in C^0(G;\mathcal{F})} \sum_i f_i(x_i) + \sum_e U_e\big(\mathcal{F}_{i \to e}(x_i) \ - \ \mathcal{F}_{j \to e}(x_j)\big),$$

with the understanding that $e = \{i, j\}$. This form shows it is an optimization over node variables, where each edge contributes a coupling cost U_e of the difference (or discrepancy) between x_i and x_j as measured in $\mathcal{F}(e)$.

A more compact way to express (2) is using the cochain notation:

$$\begin{aligned} & \min_{x \in C^0(G;\mathcal{F})} \quad \sum_{i \in V} f_i(x_i) + \sum_{e \in E} U_e\big((\delta_{\mathcal{F}} x)_e\big) \\ = & \min_{x \in C^0(G;\mathcal{F})} \quad \sum_i f_i(x_i) + U\big(\delta_{\mathcal{F}} x\big), \end{aligned} \tag{3}$$

where $U : C^1(G;\mathcal{F}) \to \mathbb{R} \cup \{\infty\}$ is the function that on a 1-cochain $y = \{y_e\}$ evaluates to $\sum_e U_e(y_e)$. The feasible set where $\delta_{\mathcal{F}} x = 0$ (global sections) can be enforced by making $U_e(0)$ finite (e.g. 0) and $U_e(y_e) = +\infty$ for $y_e \neq 0$ if we wanted a hard constraint of consistency. But more generally, allowing U_e to be smooth and convex (say quadratic) penalizes inconsistency softly rather than hard-constraining it.

[Nonlinear Homological Program] The optimization problem (3) defined by graph G, sheaf $\mathcal{F}$, node objectives $\{f_i\}$, and edge potentials $\{U_e\}$ is called a **nonlinear homological program** (NHP), denoted

$$P = (V, E, \mathcal{F}, \{f_i\}, \{U_e\}).$$

We say P is *convex* if each f_i and U_e are convex functions. We say P is *feasible* if there exists at least one global section (i.e. $\exists x$ with $\delta_{\mathcal{F}} x = 0$) in the domain of the functions such that the objective is not $+\infty$.

When P is convex, it is a convex optimization problem that can in principle be solved by centralized solvers. But our interest is in solving it *in a distributed fashion*, leveraging the graph structure. The typical difficulty is the coupling introduced by the $\delta_{\mathcal{F}} x$ term: $\delta_{\mathcal{F}} x = 0$ couples variables of neighboring nodes. However, as we will see, this coupling is sparse (each edge only couples two nodes), and we can exploit that using methods like ADMM.

Examples of Homological Programs. We give a couple of examples to illustrate how classical multi-agent problems fit into this framework.

[Consensus as a Homological Program] Let G be a connected graph of N agents. Each agent i has a variable $x_i \in \mathbb{R}$ (scalar consensus problem). We can use the constant sheaf $\underline{\mathbb{R}}$ on G (so $\mathcal{F}(i) = \mathbb{R}$, $\mathcal{F}(e) = \mathbb{R}$, and $\mathcal{F}_{i \to e}$ identity). For node objectives, take $f_i(x_i) = 0$ for all i (the agents themselves don't have any private cost; we just want consensus). For edge potentials, take $U_e(y_e) = \frac{\rho}{2} y_e^2$ (a quadratic function penalizing differences), where $\rho > 0$ is some weight. Then the program (3) becomes

$$\min_{x \in \mathbb{R}^N} \sum_{e=\{i,j\}} \frac{\rho}{2} (\mathcal{F}_{i \to e}(x_i) - \mathcal{F}_{j \to e}(x_j))^2 = \min_x \frac{\rho}{2} \sum_{e=\{i,j\}} (x_i - x_j)^2.$$

This objective expands to $\frac{\rho}{2} x^T L_G x$, where L_G is the graph Laplacian. Because L_G is positive semidefinite and has $\mathbf{1}$ (all-ones vector) in its nullspace, this objective is minimized (to 0) by any x that is constant on each connected component of G. Thus the minimizers are exactly the consensus states $\{x_i = c, \forall i\}$ for some c. So this homological program encodes the consensus problem.

If we wanted to force consensus to a particular value, we could add a node objective, e.g. $f_1(x_1) = I_{\{x_1 = c^*\}}(x_1)$ (an indicator forcing $x_1 = c^*$) to pin one node to some c^*. Then the unique minimizer would be $x_i = c^*$ for all i. In distributed averaging or agreement problems, often each agent starts with some initial value and they converge to the average; that scenario can be formulated as a consensus optimization with additional structure (like preserving sum, which is linear and might be seen via Lagrange multipliers).

[Formation Control as a Homological Program] Consider 3 robots that need to form a triangle of specified pairwise distances. Label them 1,2,3. The communication graph is complete K_3 (each pair can measure relative distance). For simplicity, assume in 2D plane. Let the state of robot i be $x_i \in \mathbb{R}^2$ (its position). Use the constant sheaf $\underline{\mathbb{R}^2}$ (so that each edge also has $\mathbb{R}^2$ as the data space, and restriction maps identity, meaning we are expecting positions to match on edges for consistency—here we will handle desired offsets via potentials, not via the linear map). Now for each edge $e = \{i, j\}$, let d_{ij}^* be the desired vector difference from i to j. For instance, we want $x_j - x_i = d_{ij}^*$. If we set a potential

$$U_e(y_e) = \frac{k}{2} \| \, y_e - d_{ij}^* \, \|^2,$$

which is minimized when $y_e = d_{ij}^*$, then our program becomes:

$$\min_{x_1, x_2, x_3} \frac{k}{2} \Big(\|(x_2 - x1) - d_{12}^*\|^2 + \|(x_3 - x1) - d_{13}^*\|^2 + \|(x_3 - x2) - d_{23}^*\|^2 \Big).$$

This sums squared errors for each pair relative to the desired difference. The minimizers (if d_*^* are compatible) correspond to formations congruent (via translation) to the desired triangle. In fact,

there will be a 2-dimensional family of minimizers corresponding to all global translations of the formation (and possibly rotations or reflections if the d^* are only distances not oriented differences). Those correspond to non-uniqueness due to symmetry (global sections can differ by an element of H^0, which here includes a translation freedom in the plane).

Many other tasks (flocking, where edges enforce velocity alignment and distance constraints; distributed sensor localization; etc.) can be encoded similarly. The pattern is: each constraint or coupling between agents is encoded as an edge potential that penalizes deviation from the desired relation.

A key theoretical result is that if all f_i are convex and all U_e are convex, then P is a convex optimization problem. If further each U_e is differentiable and has some curvature (e.g., strictly or strongly convex), then standard results ensure nice properties like uniqueness of the minimizer. In particular:

[Convexity of Homological Program] If each $f_i : \mathcal{F}(i) \to \mathbb{R} \cup \{\infty\}$ is convex and each $U_e : \mathcal{F}(e) \to \mathbb{R} \cup \{\infty\}$ is convex (and lower bounded), then P is a convex optimization problem. Moreover, if each U_e is differentiable and strongly convex (i.e., has a unique minimizer b_e for each e) and each f_i is convex, then P has a unique minimizer (assuming feasibility).

Sketch of Proof. The objective in (3) is a sum of convex functions in (x_i) and $(\delta_{\mathcal{F}} x)$ respectively, composed with a linear map $\delta_{\mathcal{F}}$. The constraints can be incorporated into the objective via indicator functions if needed. The sum of convex functions remains convex. Strong convexity of U_e ensures the overall objective is strictly convex in directions that change any inconsistent component, so combined with convex f_i (which could be just 0, not necessarily strictly convex), the whole objective becomes strictly convex in x if the only directions in nullspace of second derivative correspond to $\delta_{\mathcal{F}} x = 0$ directions (which typically are finite-dimensional global symmetries). Under technical conditions, a unique minimizer modulo those symmetries exists (one may need to fix one reference point to break symmetry). Detailed proofs appear in Hanks et al. [1]. □

Given that P is convex (in nice cases), the next question is: how do we solve it in a *distributed* manner? In a network of agents, we want an algorithm where each agent only uses information from its neighbors to update its state, and yet the group converges to the global minimizer of P. This is where the **ADMM** (Alternating Direction Method of Multipliers) comes into play, which we cover in detail in Section 8.

Before diving into ADMM, we reflect on how the above formalism connects back to our original aims: - The sheaf $\mathcal{F}$ provides the abstraction to encode various constraints and different data types on edges.

- The f_i can encode individual objectives or constraints for each agent (for example, each robot has a cost for deviance from a nominal path, or a sensor has a cost for energy usage).

- The U_e encode the coordination objectives (like consensus, formation shape maintenance, collision avoidance if desired via inequality constraints, etc.) between agents.

- The condition $\delta x = 0$ at optimum corresponds to all coordination constraints being satisfied (the network reaches a consistent configuration).

- If we set $f_i = 0$ and each U_e as an indicator of the constraint, then solving P is basically finding a global section of the sheaf that satisfies all constraints (if one exists). If U_e are penalties rather than hard constraints, solving P yields the "closest" approximate global section balancing the costs.

This optimization view is powerful, but solving it requires some method. We now have the stage set to apply a distributed optimization algorithm.

2.5 ADMM for Distributed Optimization

The Alternating Direction Method of Multipliers (ADMM) is a well-known algorithm in convex optimization that is particularly useful for problems with separable objectives and linear constraints [**?**]. It decomposes a problem into subproblems that are then coordinated to find a global solution.

Before applying ADMM to our specific problem P, let's recall the standard form of ADMM. Consider a generic problem:

$$\min_{u,v} \; F(u) + G(v) \quad \text{s.t.} \quad Au + Bv = c,$$

where F and G are convex functions in u and v respectively, and A, B are matrices (or linear operators) and c is some constant vector. The augmented Lagrangian for this problem is

$$L_\rho(u, v, \lambda) = F(u) + G(v) + \lambda^T(Au + Bv - c) + \frac{\rho}{2}\|Au + Bv - c\|^2,$$

where λ is the dual variable (Lagrange multiplier) and $\rho > 0$ is a penalty parameter. ADMM proceeds by iteratively performing:

$$\begin{aligned} u^{k+1} &= \arg\min_u L_\rho(u, v^k, \lambda^k), \\ v^{k+1} &= \arg\min_v L_\rho(u^{k+1}, v, \lambda^k), \\ \lambda^{k+1} &= \lambda^k + \rho(Au^{k+1} + Bv^{k+1} - c). \end{aligned}$$

These steps can be interpreted as follows: first, minimize the augmented Lagrangian with respect to u while keeping v fixed; then minimize with respect to v while keeping the updated u fixed; finally, update the multiplier λ using a gradient step on the dual function.

ADMM is particularly effective when F and G lead to easier subproblems in u and v separately than the combined problem in (u, v). Additionally, in multi-agent settings, the updates can often be computed in parallel when u or v comprise independent components corresponding to different agents.

In a multi-agent context, a common approach is to introduce local copies of shared variables and then enforce consistency via constraints, which ADMM handles. We will do exactly that: each agent will maintain a local version of its state (which it optimizes in its own f_i) and there will be "edge variables" that enforce the coupling, similar to y_e we had. In fact, we already wrote P with both x and y, which is convenient for ADMM because it separates nicely: the sum of $f_i(x_i)$ and $U_e(y_e)$ is separable over i and e if x and y are treated as separate blocks of variables.

Let us set up P in the consensus form for ADMM:

$$\min_{x,z} \sum_{i \in V} f_i(x_i) + \sum_{e \in E} U_e(z_e) \quad \text{s.t. } \delta_{\mathcal{F}} x + (-I)z = 0.$$

Here we consider $z = y$ as the variable for edges, and $-Iz$ just means z_e enters with a minus sign (so the constraint is $\delta_{\mathcal{F}} x - z = 0$ which is equivalent to $\delta_{\mathcal{F}} x = z$, matching y we had). We identify $u := x$ and $v := z$ in the ADMM template, with $F(u) = \sum_i f_i(x_i)$ and $G(v) = \sum_e U_e(z_e)$. The constraint matrix A corresponds to $\delta_{\mathcal{F}}$ acting on x, and B corresponds to $-I$ acting on z, and c is zero. Each of these pieces has structure:

- $\sum_i f_i(x_i)$ is separable by agent.
- $\sum_e U_e(z_e)$ is separable by edge.
- The constraint $Ax + Bz = 0$ couples x_i and z_e for each incidence.

Applying ADMM to this formulation, we introduce a dual variable λ associated with the constraint $\delta_{\mathcal{F}} x - z = 0$. The dual variable lives in the space of 1-cochains (since the constraint is a 1-cochain equation). It can be interpreted as a vector $\lambda = \{\lambda_e\}_{e \in E}$ where $\lambda_e \in \mathcal{F}(e)^*$ (same space as z_e effectively) is a Lagrange multiplier enforcing consistency on edge e. Economically, λ_e can be seen as the "price" of disagreement on edge e; physically in a multi-agent system, it might correspond to a force or tension associated with edge e trying to enforce the constraint.

The ADMM update steps become:

$$x^{k+1} = \arg\min_{x \in C^0(G;\mathcal{F})} \Big(\sum_i f_i(x_i) + \frac{\rho}{2} \sum_e \|\mathcal{F}_{i \to e}(x_i) - \mathcal{F}_{j \to e}(x_j) - z_e^k + \frac{1}{\rho}\lambda_e^k\|^2 \Big). \tag{4}$$

$$z^{k+1} = \arg\min_{z \in C^1(G;\mathcal{F})} \Big(\sum_e U_e(z_e) + \frac{\rho}{2} \sum_e \|\mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1}) - z_e + \frac{1}{\rho}\lambda_e^k\|^2 \Big). \tag{5}$$

$$\lambda_e^{k+1} = \lambda_e^k + \rho\big(\mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1}) - z_e^{k+1}\big), \quad \forall e \in E. \tag{6}$$

We have written $\mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1})$ inside since that's $(\delta_{\mathcal{F}} x^{k+1})_e$. In practice, it's easier to combine terms: note that in (4), the quadratic term couples x_i and x_j. But we can observe that (4) actually breaks into separate minimizations for each i due to separability: each $f_i(x_i)$ plus terms for each edge incident to i. Specifically, agent i sees terms involving x_i from each edge e incident on i. If we gather those, the x_i-update is:

$$x_i^{k+1} = \arg\min_{u \in \mathcal{F}(i)} f_i(u) + \frac{\rho}{2} \sum_{e : e = \{i,j\}} \|\mathcal{F}_{i \to e}(u) - (\mathcal{F}_{j \to e}(x_j^k) - z_e^k + \frac{1}{\rho}\lambda_e^k)\|^2.$$

This is effectively a proximal update: it says agent i should choose x_i as a trade-off between minimizing its own cost $f_i(x_i)$ and being close to a value that satisfies the constraints for each neighboring edge (where neighbor j's last state plus the dual adjustments set a target). If f_i is simple (e.g. quadratic), this is often an easily computable step (like averaging or thresholding).

[Proximal Operator] Given a proper, closed, and convex function $f : \mathcal{X} \to \mathbb{R} \cup \{+\infty\}$ and a positive parameter $\rho > 0$, the *proximal operator* of f with parameter ρ is defined as:

$$\text{prox}_{\rho f}(v) := \arg\min_{x \in \mathcal{X}} \left\{ f(x) + \frac{1}{2\rho}\|x - v\|^2 \right\}$$

The proximal operator can be interpreted as finding a point that balances minimizing f while staying close to v. It generalizes the notion of projection onto a set (when f is an indicator function of that set) and includes gradient steps as a special case (when f is differentiable).

The z-update in (5) is separable by edge e:

$$z_e^{k+1} = \arg\min_{w \in \mathcal{F}(e)} U_e(w) + \frac{\rho}{2}\|w - (\mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1}) + \frac{1}{\rho}\lambda_e^k)\|^2,$$

which is the proximal operator of U_e at the "point" $\mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1}) + \frac{1}{\rho}\lambda_e^k$. If U_e is an indicator of some feasible set or has a closed form proximal, this can often be solved locally by the two agents i, j sharing the value.

Finally, the λ update is just accumulation of residual $\delta_{\mathcal{F}} x - z$. Each λ_e is updated by the two agents on edge e exchanging their x values and the current z and updating λ_e.

The ADMM algorithm allows each agent and each edge to update based only on local information:

- Agent i needs x_j^k and λ_e^k, z_e^k from edges to neighbors j.

- Edge $e = \{i, j\}$ needs x_i^{k+1}, x_j^{k+1}, and λ_e^k to update z_e.
- Then edge e also updates λ_e with new x and z.

Thus communication occurs along edges only, making this distributed.

However, a challenge arises: the z-update requires solving $\arg\min_w U_e(w) + \frac{\rho}{2}\|w - \xi\|^2$ where $\xi = (\mathcal{F}_{i\to e}(x_i^{k+1}) - \mathcal{F}_{j\to e}(x_j^{k+1}) + \frac{1}{\rho}\lambda_e^k)$. If U_e is strongly convex and smooth, one could take a gradient step or closed form. But to keep it distributed, note that solving this may not be trivial in closed form unless U_e is simple (like quadratic).

2.6 Background on Nonlinear Operators and Fixed-Point Methods

Before proceeding with the nonlinear sheaf Laplacian, we briefly review some mathematical concepts that will be essential for understanding the following derivations.

Subdifferentials and Non-smooth Optimization. When dealing with non-differentiable convex functions, the standard notion of gradient is replaced by the *subdifferential*. For a convex function $f : \mathcal{X} \to \mathbb{R} \cup \{+\infty\}$, the subdifferential at a point x, denoted $\partial f(x)$, is the set of all subgradients:

$$\partial f(x) = \{g \in \mathcal{X}^* \mid f(y) \geq f(x) + \langle g, y - x\rangle \text{ for all } y \in \mathcal{X}\}$$

where $\mathcal{X}^*$ is the dual space of $\mathcal{X}$. When f is differentiable at x, then $\partial f(x) = \{\nabla f(x)\}$, a singleton set containing just the gradient. The notation $0 \in \partial f(x)$ indicates that x is a minimizer of f, which is the first-order optimality condition for convex functions.

Gradient Flows and Dynamical Systems. A gradient flow for minimizing a differentiable function f can be expressed as the ODE:

$$\dot{x} = -\nabla f(x)$$

This system evolves in the direction of steepest descent of f. For a strongly convex function, this flow converges to the unique minimizer of f. When f is not differentiable, we can use the generalized gradient flow:

$$\dot{x} \in -\partial f(x)$$

which is a differential inclusion rather than a differential equation.

Fixed-Point Iterations. For finding a point x^* such that $x^* = T(x^*)$ for some mapping T, fixed-point iteration methods use the update rule:

$$x^{k+1} = T(x^k)$$

Under certain conditions (like T being a contraction), this iteration converges to a fixed point. Many optimization algorithms, including proximal methods, can be formulated as fixed-point iterations. The equation $x = T(x)$ is called a fixed-point equation.

Implicit and Explicit Methods. When discretizing differential equations, explicit methods (like forward Euler) compute the next state directly from the current state, while implicit methods (like backward Euler) require solving an equation involving the next state. For gradient flows, the implicit Euler discretization leads to the proximal point algorithm, which has superior stability properties compared to explicit methods.

With these concepts in mind, we now return to our ADMM algorithm for distributed coordination and examine how the nonlinear sheaf Laplacian arises naturally from the z-update step.

2.7 Nonlinear Sheaf Diffusion for the Edge Update

Hanks *et al.* provide a key insight: instead of solving the z-update in one shot, one can run an iterative *diffusion process* (like our earlier $\dot{x} = -L_{\mathcal{F}}x$ but now nonlinear) to converge to the minimizer. This is where the **nonlinear sheaf Laplacian** concept enters. The z-update condition (optimality for each edge) set to zero is:

$$0 \in \nabla U_e(z_e^{k+1}) + \rho\big(z_e^{k+1} - (\mathcal{F}_{i\to e}(x_i^{k+1}) - \mathcal{F}_{j\to e}(x_j^{k+1}) + \tfrac{1}{\rho}\lambda_e^k)\big).$$

Rearranged:

$$z_e^{k+1} = \mathcal{F}_{i\to e}(x_i^{k+1}) - \mathcal{F}_{j\to e}(x_j^{k+1}) + \tfrac{1}{\rho}\lambda_e^k - \tfrac{1}{\rho}\nabla U_e(z_e^{k+1}).$$

This looks like a fixed-point equation for z_e. If we define b_e to be the unique minimizer of U_e (since U_e strongly convex, assume $b_e = \arg\min U_e$ exists and $U_e(b_e) = 0$ by shifting), then one expects z_e^{k+1} to be somewhat in between the current discrepancy and b_e.

Rather than directly solve it, they propose to treat z not as an explicit variable to update but to integrate a dynamics that reaches this solution. Concretely, consider the continuous time dynamics:

$$\dot{z}_e = -\nabla U_e(z_e) - \rho(z_e - (\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j) + \tfrac{1}{\rho}\lambda_e)).$$

At equilibrium ($\dot{z}_e = 0$), this yields exactly the above optimality condition for z_e. And crucially, this dynamic is local (involves agent i and j and the edge e state). Moreover, one can show it converges to the optimum z_e (since it's like a gradient descent in w with step ρ on a strongly convex function). In fact, if we embed this into the updates more systematically, it becomes a two-timescale algorithm: agents update x and λ at discrete steps, and edge states z are driven by a faster dynamics to near the optimum.

Hanks *et al.* go further to eliminate the need to explicitly represent z_e at all: they incorporate this dynamics directly into x update by noticing that eliminating z and λ yields a second-order update purely in x which is equivalent to a *nonlinear diffusion equation on the sheaf*. They call this the **nonlinear Laplacian dynamics**:

$$\dot{x}(t) = -L^{\mathcal{F}}_{\nabla U} x(t), \tag{7}$$

where $L^{\mathcal{F}}_{\nabla U}$ is the *nonlinear sheaf Laplacian* defined via the relation

$$(L^{\mathcal{F}}_{\nabla U}x)_i = \sum_{e=\{i,j\}} \mathcal{F}^T_{i\to e}\nabla U_e\big(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j)\big).$$

Compare this with the linear case earlier: previously, $(L_{\mathcal{F}}x)_i = \sum_e \mathcal{F}^T_{i\to e}(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j))$. Now we simply replace the linear difference by ∇U_e of that difference. If $U_e(y) = \frac{1}{2}\|y\|^2$, then $\nabla U_e(y) = y$ and we recover linear $L_{\mathcal{F}}$. If U_e is, say, quadratic with minimizer b_e, $\nabla U_e(y) = K(y - b_e)$ for some K, and the dynamics tries to push $\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j)$ towards b_e on each edge.

Hanks *et al.* proved that under the assumption that each U_e is strongly convex with minimizer $b_e \in \operatorname{im}\delta_{\mathcal{F}}$ (meaning the desired edge outcomes are consistent with some global assignment), the nonlinear Laplacian dynamics (7) converges to a global section x^* that achieves $\mathcal{F}_{i\to e}(x_i^*) - \mathcal{F}_{j\to e}(x_j^*) = b_e$ for each edge (essentially projecting the initial state onto the feasible set of constraints) [8].

[8] Hanks et al., Theorem 2, 2025. In our notation, it states that if $b = \{b_e\}_{e\in E}$ is an edge assignment in the image of $\delta_{\mathcal{F}}$ (so constraints are consistent), then $\dot{x} = -L^{\mathcal{F}}_{\nabla U}x$ converges to a point x^∞ with $\delta_{\mathcal{F}}x^\infty = b$. Equivalently, x^∞ is the global section achieving the desired b_e on each edge. If b is not exact (in image of $\delta_{\mathcal{F}}$), then no perfect global section exists; one can show the dynamics converges to a least-squares approximate solution in that case (projection onto $\operatorname{im}\delta_{\mathcal{F}}$).

Algorithmically, we can integrate a discrete approximation of (7) as the method for computing the projection in the z-update. This yields a fully distributed algorithm where agents continuously adjust their states by exchanging information with neighbors and applying the "forces" ∇U_e.

2.8 Summary of the Distributed Algorithm

The ADMM-based distributed solution for the homological program can be described at a high level:

1. Each agent i initializes its state $x_i(0)$ (perhaps to some initial guess or measured value).
2. **Repeat until convergence:**
   (a) Each agent performs a local state update (this corresponds to the x-update of ADMM, often just computing a gradient step of f_i plus neighbor terms).
   (b) The network performs a *sheaf diffusion step*: neighbors i, j communicate and adjust their states to reduce inconsistency via something like $x_i \leftarrow x_i - \alpha \mathcal{F}_{i\to e}^T(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j))$ (this is a discrete Euler step of the nonlinear Laplacian flow, repeated a few times).
   (c) Agents update dual variables (or equivalently accumulate the inconsistency error).

Under convexity and proper parameter tuning, this process converges to the optimal solution x^* of the original problem P. The resulting x^* is then a set of states each agent can take that jointly minimize the global objective while satisfying the constraints approximately (the error goes to zero if b is reachable, or is minimized in least squares otherwise).

To ensure convergence, one often requires strong convexity or a sufficiently small step size in the diffusion. Hanks *et al.* also discuss a relaxation where if strong convexity is absent, one can still run essentially the same algorithm but may need to modify the projection step (for instance, if U_e has multiple minimizers, any point in the minimizer set could work, or add a tiny quadratic regularization to U_e to select one).

3 The Classical Graph Laplacian and Consensus Review

Before further analyzing the sheaf-based method and exploring its connection to neural networks, we briefly review the classical graph Laplacian and consensus problem to anchor our intuition.

3.1 Definition and Basic Properties of Graph Laplacian

Let $G = (V, E)$ be a graph with $|V| = n$. The (combinatorial) **graph Laplacian** L_G is the $n \times n$ matrix defined by

$$(L_G)_{ij} = \begin{cases} \deg(i) & \text{if } i = j, \\ -1 & \text{if } i \sim j \text{ (edge between } i, j), \\ 0 & \text{otherwise.} \end{cases}$$

In other words, $L_G = D - A$ where D is the diagonal matrix of degrees and A is the adjacency matrix. Equivalently, for a vector $x \in \mathbb{R}^n$, the i-th component $(L_G x)_i = \sum_{j:j\sim i}(x_i - x_j)$. This operator is symmetric and positive semidefinite. Its nullspace is the span of the all-ones vector $\mathbf{1}$ (assuming G is connected), meaning $L_G x = 0$ if and only if $x_1 = x_2 = \cdots = x_n$ (consensus).

Key properties include:

- The eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ of L_G relate to connectivity (Fiedler value $\lambda_2 > 0$ if and only if G is connected).
- The Laplacian can be seen as the matrix of quadratic form $\frac{1}{2}\sum_{i\sim j}(x_i - x_j)^2$ in the sense that $x^T L_G x = \sum_{\{i,j\}\in E}(x_i - x_j)^2$.
- The pseudoinverse L_G^+ exists for connected graphs and projects onto the subspace orthogonal to $\mathbf{1}$.

3.2 Consensus Dynamics

The simplest distributed coordination algorithm on a graph is the consensus protocol: each agent i has a scalar $x_i(t)$, and they update according to

$$\dot{x}_i(t) = -\sum_{j\in N_i}(x_i(t) - x_j(t)).$$

In vector-matrix form, $\dot{x} = -L_G x$. As mentioned, this is exactly of the form (1) for a trivial sheaf. It is well known that for a connected graph, $x(t) \to c\mathbf{1}$ as $t \to \infty$, where c is the average of the initial values (if the system is weight-balanced or we use $\frac{1}{\deg(i)}$ weights, it converges to the average; for the simple Laplacian as given, it converges to some consensus value which, due to the design, actually equalizes but does not preserve average unless the graph is regular or one uses weights). Nonetheless, the main point is convergence to agreement.

This consensus can be seen as solving the optimization $\min \sum_{i<j,\, i\sim j}(x_i - x_j)^2$, which is minimized by $x_i = x_j$ for all edges (consensus). We solved it by a subgradient method (gradient of that quadratic cost yields $-Lx$).

3.3 Example: Average Consensus

Suppose 4 agents on a line graph have initial values $[3, 1, 4, 2]$. Running $\dot{x} = -Lx$, or in discrete form $x_i \leftarrow x_i + \epsilon\sum_{j\in N_i}(x_j - x_i)$, they will converge to $[2.5, 2.5, 2.5, 2.5]$ (the average) in the limit. This illustrates Laplacian flow equalizing values.

3.4 Graph Laplacian and Sheaf Laplacian Connection

As we proved in Lemma 2.3, a graph Laplacian is a special case of a sheaf Laplacian. Conversely, every sheaf Laplacian is a graph Laplacian of a certain expanded graph (each edge in original graph could be thought of as multiple edges connecting the vector components). This connection means results from algebraic graph theory, such as spectral bounds and convergence rates, have analogues in sheaf Laplacians (often depending on the norms of restriction maps).

One such analogy: the consensus algorithm's convergence speed is related to $\lambda_2(L_G)$. For sheaf Laplacians, the convergence of $\dot{x} = -L_{\mathcal{F}} x$ is related to the smallest nonzero eigenvalue of $L_{\mathcal{F}}$. Sheaf Laplacians can have multiple zero eigenvalues if the sheaf has multi-dimensional global sections, but if we mod out those, the spectral gap gives the rate at which local inconsistencies dissipate.

This ties into multi-agent coordination: a larger spectral gap means faster agreement (or constraint satisfaction) across the network. Using appropriate sheaf design can increase this gap by effectively adding "more edges" through vector dimensions coupling (like a high-dimensional edge constraints can act like multiple parallel constraints).

4 Sheaf Laplacians and Decentralized Flows

In Section 2, we discussed how $\dot{x} = -L_{\mathcal{F}}x$ acts as a decentralized flow driving the network toward global consistency (a sheaf global section). We now delve deeper into interpreting this in the context of multi-agent systems and mention how nonlinear extensions (using ∇U) serve as decentralized controllers.

4.1 Linear Sheaf Laplacian = Decentralized Linear Controller

Consider a scenario where agents' goal is to satisfy linear equations among their states. For example, agent j should have a state that equals a fixed linear transformation of agent i's state. If that transformation is encoded in $\mathcal{F}_{i\to e}$ and $\mathcal{F}_{j\to e}$, then $L_{\mathcal{F}}x = 0$ are exactly the conditions for all those equations to hold. The protocol $\dot{x} = -L_{\mathcal{F}}x$ is then a proportional control law: each agent adjusts based on weighted differences between the left and right sides of each equation for each constraint it's involved in. The fact that this is negative feedback on the constraint error ensures stability and convergence to satisfaction of the constraints (if possible).

For example, suppose two UAVs must maintain the same altitude difference as two UGVs (ground vehicles). We can set a sheaf where UAV altitude and UGV altitude on an edge have restriction map identity (so edge data is a "height difference"). The linear consensus tries to equalize those height differences, effectively making the UAVs and UGVs coordinate altitudes. This is a bit abstract, but basically any linear decentralized control can be seen as a Laplacian on an appropriate sheaf.

4.2 Nonlinear Sheaf Laplacian = Decentralized Nonlinear Controller

When constraints are nonlinear (like distance = 5, which is $\|x_i - x_j\| = 5$), a common approach is to design a decentralized controller using gradient descent on a potential function encoding that constraint (for distance, a potential like $(\|x_i - x_j\|^2 - 25)^2$). The nonlinear sheaf Laplacian does exactly this in a structured way: $\nabla U_e(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j))$ is the gradient of the edge's potential with respect to its arguments, pulling x_i and x_j in directions to satisfy the constraint. And $\mathcal{F}^T_{i\to e}$ maps that "force" back into the direction in x_i's own space (like projecting a force vector from edge coordinate frame to agent's coordinate frame).

Thus $\dot{x} = -L^{\mathcal{F}}_{\nabla U}x$ is nothing but a network of agents each feeling forces from each incident constraint, where each constraint exerts equal and opposite forces on its two participants (ensuring consistency with Newton's third law, in a sense, if $\mathcal{F}^T_{i\to e}$ and $\mathcal{F}^T_{j\to e}$ produce opposite effects).

This perspective bridges to physical analogies: one can think of each edge constraint as a nonlinear spring between some linear functions of agent states. Then $L^{\mathcal{F}}_{\nabla U}x = 0$ is the static equilibrium of those springs, and the dynamics is a damped relaxation to equilibrium.

In multi-agent coordination, many problems (consensus, formation, flocking) can be interpreted as such potential-minimizing systems:

- Consensus: spring potential $\propto (x_i - x_j)^2$ along each communication link.
- Formation: spring potential $\propto (\|x_i - x_j\| - d_{ij})^2$ to maintain distance.
- Flocking (alignment): potential to align velocities (like an edge potential $\propto \|v_i - v_j\|^2$).
- And so on.

Therefore, the nonlinear sheaf Laplacian flow provides a unified model for these as gradient systems (which are easier to analyze for stability etc.).

5 Framework for Multi-Agent Coordination

Now we step back and outline a systematic procedure to model and solve a multi-agent coordination problem using the tools discussed.

5.1 System Model

We consider N agents labeled $1, \ldots, N$. Each agent i has:

- A state vector $x_i \in \mathbb{R}^{n_i}$ (dimension n_i could differ per agent).
- Dynamics or control authority: in this paper's scope, we assume we can directly set the state or the state is static during a coordination solving phase (i.e., we focus on the steady-state or algebraic coordination problem rather than transient dynamics). If dynamic, one might incorporate that into the sheaf as a time-extended sheaf (see Example 4 in Section 2.3 where a trajectory was a section on a path graph).
- Possibly an individual objective or cost function $f_i(x_i)$ capturing preferences (like how far from some nominal state).

The inter-agent interactions are given by a communication/interaction graph $G = (V, E)$ where each edge $\{i, j\}$ indicates agent i and j can directly exchange information or influence each other.

Crucially, to capture heterogeneous interactions, we introduce a sheaf $\mathcal{F}$ on G. How to design $\mathcal{F}$? Typically:

- Choose $\mathcal{F}(i) = \mathbb{R}^{n_i}$, matching the agent's state space.
- For each edge $e = \{i, j\}$, decide what relationship it enforces. For example:
  - If it's a consensus-type link for some component of state, $\mathcal{F}(e)$ could be a subspace of $\mathbb{R}^{n_i}$ and $\mathbb{R}^{n_j}$ that represents the quantity to agree on. $\mathcal{F}_{i\to e}, \mathcal{F}_{j\to e}$ would be projection maps onto that subspace. (E.g., if agents have $(\mathit{position}, \mathit{temperature})$ as state and the edge is to agree on temperature, then $\mathcal{F}(e) = \mathbb{R}$ and $\mathcal{F}_{i\to e}$ picks the temperature coordinate from $\mathbb{R}^{n_i}$).
  - If it's a formation link ensuring a certain relative position, $\mathcal{F}(e)$ might equal $\mathbb{R}^d$ (the physical space) and $\mathcal{F}_{i\to e}(x_i) = P_{ij}x_i$ is some projection of agent i's state to a position (maybe identity if x_i itself is position, or perhaps something like if x_i includes orientation, we might want relative orientation, etc).
  - If it's a constraint that agent i's output should feed into agent j's input (like in a supply-demand network or task scheduling), $\mathcal{F}_{i\to e}$ could be the output scalar and $\mathcal{F}_{j\to e}$ the input scalar, and $\mathcal{F}(e) = \mathbb{R}$ to equate them.

This step is perhaps the most artful: one must decide how to represent the multi-agent coordination requirements in terms of local constraint mappings. Tools like a requirement graph or knowledge of the task are needed.

Once $\mathcal{F}$ is set, any assignment of states to agents can be checked for consistency via $\delta_{\mathcal{F}}x$. If $\delta_{\mathcal{F}}x = 0$, the agents perfectly satisfy all the relations encoded by edges.

5.2 Coordination Objective

Sometimes, just satisfying constraints is not enough; we might also optimize a performance criterion. For example, agents might want to minimize energy usage while maintaining formation, or minimize deviation from their preferred positions while reaching consensus.

This is where the node objective $f_i(x_i)$ come in. We gather those from the problem statement. These could be:

- Quadratic penalties $(1/2)\|x_i - \hat{x}_i\|^2$ for straying from some reference $\hat{x}_i$.
- Indicators or constraints like x_i must lie in some set.
- 0 if no specific cost for agent aside from constraints.

Additionally, sometimes there are global objectives like maximize the average of something or minimize total variance. If they decompose per agent or can be assigned to edges, we incorporate accordingly (if truly global in a non-separable way, ADMM still can handle if represented as a star graph or by introducing extra variables, but we won't complicate with that here).

5.3 General Procedure

Given the above, the steps are:

1. **Model with a sheaf:** Define G and $\mathcal{F}$ to encode agent capabilities and required relationships.

2. **Set up the homological program:** Write $P = (V, E, \mathcal{F}, \{f_i\}, \{U_e\})$. For each edge constraint, choose a suitable potential function U_e. If the task is a hard constraint (like exactly maintain distance d), one can use a barrier or a very stiff quadratic around d. If soft, then softer penalty.

3. **Ensure convexity or handle non-convexity:** Many coordination tasks (like distance constraints) lead to non-convex U_e (a quadratic $\|x_i - x_j\|^2$ is convex in (x_i, x_j), but if we want $\|x_i - x_j\| = d$ exactly, that's non-convex or we could square it and penalize differences from d^2 which is convex in x_i, x_j but has multiple minima including potentially $x_i = x_j$ and so on). In general, if the problem is not convex, ADMM still can be applied as a heuristic but no guarantee. Hanks et al. mostly considered convex (or convexified) cases. For exposition we lean on convex examples.

4. **Apply ADMM (distributed):** Use the algorithm from Section 8 to solve. Translate it into pseudocode for agents: each agent iteratively updates its state and communicates with neighbors. Alternatively, derive the closed-form dynamics if possible.

5. **Execute the strategy:** If this is an online control scenario, agents would continuously update their control inputs according to the solving algorithm and eventually reach coordination.

6. **Validate constraints:** At convergence, verify if $\delta x \approx 0$ (all constraints satisfied within tolerance) and evaluate the cost.

This structured approach helps ensure no constraint is forgotten and every multi-agent interaction is accounted for in the model.

6 Main Contributions

The sheaf-based coordination framework offers several notable contributions and advantages compared to classical graph-based methods:

1. **Unified Modeling of Heterogeneous Constraints:** It provides a single formalism to model a wide range of multi-agent coordination problems. Classical approaches often treat consensus, formation, sensor fusion, etc., separately, each with its own custom algorithm. In contrast, by using cellular sheaves, all these problems become instances of finding consistent sections under certain constraints. This unification illuminates deep connections between problems (e.g., consensus and formation are both about aligning sections of a sheaf) and allows one to transfer insights and techniques across domains.

2. **Rich Expressiveness:** Sheaves allow the modeling of vector-valued and structured data on networks. This means we can naturally handle *asymmetric or directed interactions* (via non-symmetric restriction maps), multi-dimensional relationships (each edge can enforce a vector equation), and constraints that are not simply pairwise equality (they can be linear transformations or even nonlinear relations when coupled with potentials). This goes beyond what a weighted graph Laplacian can represent.

3. **Topology-Aware Consistency:** By leveraging the machinery of cohomology (through δ and H^0, H^1), we gain access to tools like Hodge theory to reason about when coordination constraints have solutions and how "far" a given state is from any feasible solution (measured by cohomology classes). This is particularly useful in detecting and quantifying *inconsistent loops or global obstructions* in the network constraints, something graph-theoretic approaches struggle to formalize. For example, if agents have cyclic dependencies that cannot all be satisfied (an unsolvable constraint loop), $H^1(G;\mathcal{F}) \neq 0$ will reveal that.

4. **Distributed Solver with Theoretical Guarantees:** The framework doesn't just pose the problem but also provides a systematic way to solve it via ADMM and nonlinear sheaf Laplacian flows. Under convexity assumptions, we have convergence guarantees to global optima[9]. Even for certain nonconvex problems, the approach (as a heuristics) tends to perform well by exploiting problem structure. The distributed algorithm (Algorithm 1 in [1]) clearly delineates the local computations and neighbor communications, facilitating implementation.

5. **Link to Neural Network Perspective:** As we elaborate in Section 9, unrolling the iterations of the distributed solver yields a neural network-like architecture. This connection means we can potentially tune or learn parts of the coordination protocol (e.g., the edge potential parameters) using data, and conversely apply insights from deep learning (like initialization or normalization strategies) to improve coordination algorithms. It sets the stage for a synergy between rigorous optimization-based control and learning-based adaptation.

These contributions make the sheaf-based approach a promising paradigm for complex multi-agent coordination going forward. It generalizes classical consensus and graph Laplacian methods (recovering them as special cases), and offers a pathway to integrate additional structure and objectives without losing the ability to analyze or solve the problem.

[9]In particular, ADMM for convex problems is known to converge under mild conditions. See Boyd et al., "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundations and Trends in Machine Learning* 3, no. 1 (2011): 1–122, for ADMM convergence theory. Hanks et al. extend this to the sheaf setting and prove convergence and optimality for the homological program solutio.

7 General Multi-Agent Coordination Homological Program

For clarity, we formally summarize the formulation of a general multi-agent coordination problem as a homological program:

$$P^* \;=\; \operatorname*{arg\,min}_{x \in C^0(G;\mathcal{F})} \; \sum_{i \in V} f_i(x_i) \;\text{ s.t. }\; \delta_{\mathcal{F}} x = 0 \;,$$

where

- $G = (V, E)$ is the agent communication graph;
- $\mathcal{F}$ is a cellular sheaf on G capturing the data dimensions and consistency maps;
- $f_i(x_i)$ is the local cost for agent i (convex, proper, lower semicontinuous);
- The constraint $\delta_{\mathcal{F}} x = 0$ means x is a global section of the sheaf (all inter-agent consistency requirements are satisfied).

If needed, the constraint can be relaxed or penalized by introducing edge potentials U_e as described earlier, yielding the unconstrained form:

$$P = \operatorname*{arg\,min}_{x \in C^0(G;\mathcal{F})} \; \sum_{i} f_i(x_i) + \sum_{e} U_e\big((\delta_{\mathcal{F}} x)_e\big) \;,$$

which for sufficiently large penalty (or in the limit of hard constraints) recovers the constrained problem. This P is the general multi-agent coordination homological program.

In practice, to use this framework one goes through the modeling steps exemplified in Section 5: identify the appropriate sheaf (data structure) and costs from the problem requirements, then set up P. Once P is set up, we turn to solving it in a distributed manner.

8 Solving Homological Programs with ADMM

We now outline the solution procedure using ADMM, following the approach of Hanks *et al.* [1]. The key idea is to split the problem across agents and edges so that each can be handled locally, and a simple iteration enforces agreement.

8.1 Assumptions for Distributed Optimization

To ensure convergence of the ADMM-based solver, we assume:

1. **Convexity:** Each $f_i : \mathcal{F}(i) \to \mathbb{R} \cup \{\infty\}$ is convex (and closed, proper), and each $U_e : \mathcal{F}(e) \to \mathbb{R} \cup \{\infty\}$ is convex. This makes the overall objective convex.
2. **Strong Convexity of Potentials:** Each edge potential U_e is strongly convex and differentiable, with a unique minimizer b_e (interpreted as the "target" consistent value on that edge. Moreover, assume there is no fundamental inconsistency: the set of global sections achieving b_e on each edge is non-empty (i.e. $b = \{b_e\}$ lies in $\operatorname{im} \delta_{\mathcal{F}}$).
3. **Feasibility or Slater's condition:** There exists at least one assignment x such that $f_i(x_i) < \infty$ for all i and $\delta x = 0$ (or in the penalized version, the constraints are soft so feasibility is trivial). Basically, the problem is well-posed.

These conditions allow us to apply ADMM and guarantee it converges to the unique optimal solution of P^*.

8.2 Consensus Form Reformulation

We introduce local copies for the coupled variables to get an ADMM-friendly form. Let $z = \{z_e\}_{e \in E}$ be auxiliary variables intended to equal $\delta_{\mathcal{F}} x$. The constraint becomes $\delta_{\mathcal{F}} x - z = 0$. We incorporate hard constraints with an indicator function $\chi_C(z)$ for $C := \{z \,|\, z = 0\}$, or if using U_e, it's already in the objective. For conceptual clarity, assume the hard form:

$$\min_{x,z} \sum_i f_i(x_i) + \chi_C(z) \quad \text{s.t.} \quad \delta_{\mathcal{F}} x - z = 0 \ .$$

This is equivalent to our original problem. The augmented Lagrangian (with scaled dual variable $y = \{y_e\}_{e \in E}$ where $y_e \in \mathcal{F}(e)$ for edge constraints) is:

$$\mathcal{L}_\rho(x, z, y) = \sum_i f_i(x_i) + \chi_C(z) + \frac{\rho}{2} \| \, \delta_{\mathcal{F}} x - z + y \, \|_2^2 \ ,$$

where y is essentially the dual variable divided by ρ (this is the typical "scaled" form of ADMM [5]). The ADMM updates are then:

$$x^{k+1} = \arg\min_x \ \sum_i f_i(x_i) + \frac{\rho}{2} \|\delta_{\mathcal{F}} x - z^k + y^k\|^2 \ , \tag{8}$$

$$z^{k+1} = \arg\min_z \ \chi_C(z) + \frac{\rho}{2} \|\delta_{\mathcal{F}} x^{k+1} - z + y^k\|^2 \ , \tag{9}$$

$$y^{k+1} = y^k + \delta_{\mathcal{F}} x^{k+1} - z^{k+1} \ . \tag{10}$$

Because of separability, (8) breaks into independent problems for each x_i (each f_i plus quadratic terms involving x_i and its neighbors' data), and (9) is independent for each edge (since $\chi_C(z)$ decomposes over edges, enforcing each $z_e = 0$). In fact, (9) yields simply

$$z^{k+1} = \Pi_C(\delta_{\mathcal{F}} x^{k+1} + y^k) \ ,$$

i.e. $z^{k+1} = \delta_{\mathcal{F}} x^{k+1} + y^k$ projected onto C (which is zero, so $z^{k+1} = 0$ and effectively $\delta_{\mathcal{F}} x^{k+1} + y^k$ gets added to y). This simplifies the updates:

$$x_i^{k+1} = \arg\min_{x_i} \ f_i(x_i) + \frac{\rho}{2} \sum_{e \ni i} \|\mathcal{F}_{i \to e}(x_i) - (\mathcal{F}_{j \to e}(x_j^k) - y_e^k)\|^2,$$

$$y_e^{k+1} = y_e^k + (\mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1})) \ ,$$

where $e = \{i, j\}$ and an orientation is assumed. We dropped z entirely (since the projection simply enforces $z_e = 0$ each time).

If we instead had the soft constraint version with U_e in objective, the z-update would minimize $U_e(z_e) + \frac{\rho}{2}\|z_e - (\mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1}) + y_e^k)\|^2$ for each edge e. That gives the optimality condition:

$$\partial U_e(z_e^{k+1}) + \rho\big(z_e^{k+1} - [\mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1}) + y_e^k]\big) \ni 0 \ .$$

One can interpret this as computing a proximal operator of U_e. Solving this exactly may require a centralized step or iterative method, which is where the idea of using a *diffusion process* to compute it arises (next subsection).

8.3 Nonlinear Sheaf Diffusion for the Edge Update

To compute z^{k+1} (or equivalently to enforce δx^{k+1} satisfies the constraint) in a distributed way, one can simulate the continuous-time nonlinear heat flow:

$$\dot{w}_e(t) = -\nabla U_e(w_e) - \rho\big(w_e - [\mathcal{F}_{i\to e}(x_i^{k+1}) - \mathcal{F}_{j\to e}(x_j^{k+1}) + y_e^k]\big) \ ,$$

for each edge e, which converges to $w_e^* = z_e^{k+1}$. This is essentially a gradient descent in w for the objective in (9). As Hanks et al. show, running this to convergence is equivalent to solving a **nonlinear Laplacian equation** on the networ. In particular, assembling all these w_e equations corresponds to the nonlinear sheaf Laplacian dynamics:

$$\dot{x} = -L_{\nabla U}^{\mathcal{F}} x \ ,$$

which was given in (7) earlier. Intuitively, this diffusion allows neighboring agents to repeatedly adjust their tentative values to reach a consensus that minimizes the edge potentials. In implementation, one may not run it to full convergence every iteration; instead, interleaving one or a few diffusion steps between x-updates can be effective and still converge (this becomes a synchronous relaxation method).

The result is that the z-update (or equivalently enforcing consistency) can be done with purely local neighbor-to-neighbor interactions, without any centralized coordination. Theorem 8.1 in [1] ensures that if the desired consistent edge values b_e are attainable, this diffusion will find the projection of the current state onto the constraint manifold (global consistency.

8.4 Summary of the Distributed Algorithm

Putting the pieces together, a high-level description of the distributed ADMM algorithm for the homological program is:

Algorithm: Distributed Sheaf-ADMM

1. Initialize $x_i(0)$ for all agents (e.g., to some initial measurement or guess), and dual variables $y_e(0) = 0$ for all edges.

2. For $k = 0, 1, 2, \ldots$ (iterate until convergence):

   (a) **Local primal update:** Each agent i receives the current dual variables y_e^k from each incident edge $e = \{i, j\}$ (or equivalently receives y_e from a designated orientation). Then agent i updates its state:

   $$x_i^{k+1} := \arg\min_{x_i} f_i(x_i) + \frac{\rho}{2} \sum_{j \in N_i} \|\mathcal{F}_{i\to(i,j)}(x_i) - \mathcal{F}_{j\to(i,j)}(x_j^k) + y_{(i,j)}^k\|^2 \ ,$$

   which it can solve on its own (often a simple quadratic minimization or proximal operation).

   (b) **Sheaf diffusion (consistency enforcement):** For each edge $e = \{i, j\}$, the two agents i, j engage in a local exchange to adjust their states toward satisfying the constraint. This can be done by iterating:

   $$x_i \leftarrow x_i - \eta\, \mathcal{F}_{i\to e}^T \nabla U_e(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j)),$$

   $$x_j \leftarrow x_j - \eta\, \mathcal{F}_{j\to e}^T \nabla U_e(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j)),$$

for a few small steps $\eta > 0$. In practice, i and j might repeatedly apply these updates using the latest information from each other until a certain tolerance or number of micro-iterations is reached. (This is a discrete implementation of the nonlinear heat equation on the edge.)

(c) **Dual update:** Each edge $e = \{i, j\}$ updates its dual variable:

$$y_e^{k+1} := y_e^k + \mathcal{F}_{i \to e}(x_i^{k+1}) - \mathcal{F}_{j \to e}(x_j^{k+1}) \ ,$$

which is the discrepancy after the x-update (the diffusion step is trying to minimize this, so ideally it is small). This y_e is then sent to agents i and j to be used in the next iteration.

Each iteration involves agents computing updates using only neighbor information, and dual variables (which can be seen as "message variables") being exchanged between neighbors. Convergence is determined, for instance, by the norm of primal residuals $\|\delta x^k\|$ and dual residuals $\|x^k - x^{k-1}\|$ falling below a threshold, as per standard ADMM criteria [5].

8.5 Convergence and Optimality

Under the assumptions stated in Section 8.1, the algorithm converges to the global optimum x^* of the convex program P^*. In the hard-constraint formulation, this means that eventually $\delta_{\mathcal{F}} x^* = 0$ (all constraints satisfied) and x^* minimizes $\sum_i f_i(x_i)$ subject to that. In the soft formulation, it means x^* minimizes the augmented objective with U_e penalties (which often implies a good approximate satisfaction of constraints, or exact if penalties were enforced as barriers).

Hanks *et al.* also discuss relaxing the strong convexity assumption: if U_e are merely convex (not strongly), the solution may not be unique on the constraint manifold (there could be a family of equally optimal consistent states). In that case, ADMM can still converge, but the dual variables y_e might not converge to a unique value. One remedy is adding a tiny regularization (strong convexification) to each U_e to pick one solution. Alternatively, one can show that even without strong convexity, the primal variables x still converge to *a* solution (though possibly not uniquely determined by initial conditions if the problem has symmetry). This is a minor technical point — in most practical cases, constraints like consensus or formation have unique solutions except for trivial symmetries (like a common offset or rotation), which do not impede the functioning of the system.

By combining the power of convex optimization with sheaf-theoretic consistency constraints, we obtain a distributed algorithm that generalizes many known coordination protocols (for example, it reduces to the usual consensus algorithm when $\mathcal{F}$ is trivial and f_i are quadratic). We next turn to interpreting these procedures through the lens of neural networks, which offers additional insight and potential for extension.

9 From Sheaves and Laplacians to Neural Networks

It is insightful to interpret the iterative distributed algorithm as a form of neural network that *learns or converges* to a solution. Recently, researchers have introduced **Sheaf Neural Networks (SNNs)** as a generalization of Graph Neural Networks (GNNs) that operate on cellular sheaves rather than on plain graph. In this section, we draw parallels between our framework and SNNs, illuminating how multi-agent coordination can be seen as (or embedded into) a neural network architecture.

9.1 Sheaf Neural Networks: Definition and Structure

A Sheaf Neural Network layer, informally, takes an input assignment of feature vectors to the nodes of a graph (a 0-cochain with values in some feature sheaf) and produces an output assignment (a transformed 0-cochain) by a combination of:

- A linear transformation on each node's feature (analogous to a weight matrix in a neural layer, acting independently on each node).
- A propagation of information along edges that respects the sheaf structure. Typically, this involves applying restriction maps to node features, exchanging information across the edge, possibly applying some edge-specific weighting or transformation, and then aggregating back into node updates.
- A nonlinear activation function applied to the node features.

For example, an SNN as defined in [6] uses the sheaf Laplacian in the following way: one can form a "sheaf convolution" operator $H = I - \epsilon L_{\mathcal{F}}$ for small ϵ, which is analogous to the graph convolution $I - \epsilon L_G$ in GNNs. Stacking such operations with intermediate nonlinearities yields an SNN.

In our context, consider the process of one ADMM iteration as a mapping $(x^k, y^k) \mapsto (x^{k+1}, y^{k+1})$. This mapping is reminiscent of a two-layer update:

- The x-update can be seen as each node taking its current state x_i^k and neighbor messages y^k (which carry a summary of neighbor states from the previous iteration) and producing a new state x_i^{k+1}. If f_i is quadratic, this is an affine combination of x_i^k and neighbors' contributions, possibly with a nonlinearity if f_i had one (like a clipping if domain constrained). This is analogous to a neuron update: sum weighted inputs (neighbors' states) and apply a (proximal) nonlinearity.
- The dual update can be seen as another layer where each edge (which could be regarded as a separate computational unit) receives the two endpoint states x_i^{k+1}, x_j^{k+1} and outputs an updated message y_e^{k+1}. The formula $y_e^{k+1} = y_e^k + (\mathcal{F}_{i\to e}(x_i^{k+1}) - \mathcal{F}_{j\to e}(x_j^{k+1}))$ is essentially an accumulation of inconsistency (like an error signal). One could interpret y_e as analogous to a hidden layer that stores "memory" about the edge consistency (like a recurrent connection).

If we unroll these iterations for a fixed number of steps T, we have a feedforward network of depth T, where the parameters of the network are determined by f_i, U_e, $\mathcal{F}_{i\to e}$, and ρ. Interestingly, these parameters are not learned in the coordination setting; they are fixed by the problem. But one could envision learning some of them (for instance, if f_i had unknown weights that could be tuned to improve performance or adapt to environment).

Thus, the sheaf-based coordination algorithm itself *is a kind of neural network* — specifically, a monotone operator network aimed at solving an optimization. This viewpoint is related to the concept of **algorithm unrolling** in machine learning, where one takes an iterative algorithm and treats a finite number of its iterations as a neural network layer sequence, possibly fine-tuning it using data.

9.2 Node State Updates as Neuron Activations

Examining (8), the node update step for agent i:

$$x_i^{k+1} = \mathrm{prox}_{f_i/\rho}\Big(x_i^k - \sum_{j\in N_i} \mathcal{F}_{i\to e}^T(y_e^k + \mathcal{F}_{i\to e}(x_i^k) - \mathcal{F}_{j\to e}(x_j^k)) \Big) ,$$

where prox denotes the proximal operator (which is identity minus gradient of f_i if f_i is differentiable). This equation closely resembles the update of a neuron that receives input (x_i^k and neighbor terms), performs a linear combination, and then applies a nonlinear function (the prox, which could be e.g. a truncation if f_i encodes bounds, or a shrinkage if f_i is an ℓ_1 penalty, etc.). In the simplest case $f_i(x) = \frac{1}{2}\|x\|^2$, prox is just identity minus a linear term, and the update becomes $x_i^{k+1} = x_i^k - \alpha \sum_{j \in N_i} \mathcal{F}_{i\to e}^T(\mathcal{F}_{i\to e}(x_i^k) - \mathcal{F}_{j\to e}(x_j^k) + y_e^k)$, which is an affine function of the inputs (no nonlinearity). If f_i enforces constraints like $x_i \geq 0$ (nonnegativity of some variables), then the prox includes a ReLU-like clip at 0. So the x-update layer can incorporate activation functions naturally depending on the form of f_i.

9.3 Sheaf Diffusion as Message-Passing Layer

The diffusion step (Section 8.3) can be viewed as a message-passing layer. In graph neural networks, a typical layer update is:

$$h_i^{new} = \sigma\Big(W \cdot h_i^{old} + \sum_{j \in N_i} W' \cdot h_j^{old}\Big)$$

for some weight matrices W, W' and nonlinearity σ. In our case, during diffusion each agent's state is adjusted by something coming from each neighbor: $-\eta \mathcal{F}_{i\to e}^T \nabla U_e(\mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{j\to e}(x_j))$. This is a message from neighbor j to i, which depends on x_i, x_j, and the function ∇U_e. In an SNN, one could have a learnable function or attention mechanism on each edge; here ∇U_e plays that role, determining how much influence the neighbor's difference has. For example, if we set $U_e(y) = \frac{1}{2}\|y\|^2$, then $\nabla U_e(y) = y$ and the message is just $-(\mathcal{F}_{i\to e}^T \mathcal{F}_{i\to e}(x_i) - \mathcal{F}_{i\to e}^T \mathcal{F}_{j\to e}(x_j))$. If U_e is something like a saturated loss that grows slowly after a point, ∇U_e might reduce large differences' effect (similar to clipping gradients, which is analogous to robust message passing).

The diffusion is essentially a message-passing layer enforcing consistency, and it is naturally incorporated in the ADMM iteration. In a purely feed-forward SNN, one might include a similar diffusion or use the sheaf Laplacian in each layer to mix features.

9.4 Role of Projection (Consensus) and Dual Variables

The Euclidean projection Π_C in the ADMM algorithm (which we implemented via diffusion) has an analogue in neural networks as well: it's like a normalization or constraint satisfaction step between layers. Some neural network architectures include normalization layers (like batch norm or layer norm) to enforce certain conditions (zero mean, unit variance) at each layer. Here, Π_C ensures that after each full iteration, the "intermediate" z is in the feasible set of constraints. We can think of it as a layer that enforces the sheaf constraint approximately at every step. If one unrolled infinitely many layers, the final output would be in the constraint set exactly. With a finite number, it's approximately satisfying constraints (much like how a sufficiently deep GNN can approximate broad functions, but a shallow one might not exactly solve a complex constraint, though it gets closer with each layer).

The dual variables y_e can be seen as carrying the memory of past inconsistencies, which improves convergence (like momentum in optimization). In a network sense, y_e are additional hidden features on edges accumulating information across layers. One might imagine an architecture where edges have hidden units that interact with node units. Indeed, some recent GNN models introduce edge features that evolve alongside node features. Here, y_e is exactly that: an edge feature updated as $y_e^{k+1} = y_e^k +$ (neighbor difference). This resembles a recurrent connection (like an LSTM gating, albeit simpler linear accumulation) ensuring that if a constraint hasn't been satisfied yet, the "pressure" to satisfy it builds up (encoded in y_e).

9.5 Optimization View vs. Learning View

A striking difference between our ADMM-unrolled network and a typical learned neural network is that our network's "weights" and non-linearities are derived from an optimization problem, not trained from data. We are effectively doing *predictive coding* where the "prediction" is the consistent optimal state and the "error" is driven to zero via iterative feedback (dual updates). On the flip side, if we had multiple coordination tasks or unknown parameters in f_i, U_e, we could imagine learning those parameters by gradient descent through this network. For instance, if $U_e(y)$ had a parameter that we want to tune so that the system converges faster, we could treat convergence speed or error as a loss and backpropagate through the unrolled iterations to adjust that parameter.

This cross-pollination of ideas suggests a future direction where control strategies for multi-agent systems are designed with neural network principles (modularity, trainable components) and conversely, neural networks for graph-structured data incorporate principled constraints via sheaf Laplacians (ensuring learned representations respect certain invariances or physical laws).

The correspondence is:

Coordination Algorithm	**Neural Network Analogy**
Agent states x_i^k	Neuron activations/feature vectors
Dual variables y_e^k	Edge hidden units / error signals
Local objective f_i	Neuron activation function (via prox)
Edge potential U_e	Message function / attention along edge
Iteration steps	Network layers (unrolled in time)
Converged x^*	Network output after many layers

Understanding these analogies can inspire hybrid methods: one could use a few iterations of the theoretical ADMM (ensuring feasibility and decent optimum) and then use a neural network to fine-tune to real-world data or uncertainties, or use a neural network to warm-start ADMM. Or simply leverage hardware and software from deep learning to implement distributed solvers more efficiently.

10 Advantages of Sheaf Neural Networks for Learning and Coordination

Considering the above connections, we list some specific advantages that using sheaf neural network concepts (and, by extension, our sheaf coordination framework) brings:

10.1 Modeling Asymmetric and Heterogeneous Interactions

Standard GNNs often assume homogeneous scalar weights on edges (or at best a fixed vector of weights for a given edge type). Sheaf Neural Networks, however, inherently handle **asymmetric interactions**: since each edge has distinct restriction maps for each endpoint, the influence of node i on node j can differ from j on i. This is crucial for capturing *heterophily* in networks (cases where connected nodes may have very different states rather than similar). Empirical studies have shown SNNs excel in graphs with directed edges or where the relationship is not "mirror-symmetric". In multi-agent coordination, this means we can rigorously model situations like one agent's output being another's input (a directed dependence) or interactions that are not identical in both directions (like one agent might enforce a constraint on another but not vice versa). The use of sheaves thus broadens the class of systems and communication patterns that can be handled seamlessly.

10.2 Nonlinear Diffusion and Improved Expressivity

By incorporating nonlinear potentials U_e, the message-passing in the network becomes nonlinear in the differences between agents' states. This **nonlinear diffusion** mechanism allows the network to implement more complex behaviors than a linear Laplacian does. For example, it can learn to ignore small differences (below some threshold) and only act strongly on large disagreements, or vice versa, akin to a robust or adaptive diffusion. From a learning perspective, this gives SNNs a richer function class to approximate. Recent work titled "Sheaf Diffusion Goes Nonlinear" demonstrates that introducing nonlinearity in the sheaf propagation can significantly enhance performance of GNNs on certain tasks (e.g., learning functions on graphs that normal GNNs struggle with due to oversmoothing or heterophily. In coordination, a nonlinear diffusion means the system can handle constraints that only need to kick in beyond certain thresholds (like collision avoidance might be negligible when far apart but strongly repulsive when too close, which is naturally modeled by a nonlinear potential).

10.3 Decentralized Learning and Adaptation

A sheaf neural network can be executed in a decentralized manner: each layer's computation uses only local neighbor information. This means, in principle, an SNN can run on the multi-agent system itself. Agents could run a neural network that was trained (possibly offline, or even online in a continual learning way) to achieve coordination under certain conditions (including learned behaviors or heuristic improvements). This is different from standard central training of controllers. In essence, **decentralized learning** is enabled by the locality of SNN layers. Moreover, one could personalize the behavior for each agent by having agent-specific functions f_i or features, which the framework naturally accommodates (each node can have its own parameters in the neural net or its own part of the objective in control terms).

An exciting possibility is to use reinforcement learning or gradient-based learning to adjust the local objective functions f_i or potentials U_e based on performance feedback, all while maintaining the sheaf constraint structure. Because the sheaf formalism ensures consistency, any learned policy inherently respects the critical constraints (no need to relearn physical laws or communication protocols – they are "baked in" to the architecture). This makes safe learning more feasible: the network will not easily produce a grossly inconsistent action because the architecture itself penalizes that heavily.

The marriage of sheaf-based optimization with neural network methodology offers:

- *Greater modeling power* (through asymmetric, high-dimensional relations).
- *Enhanced algorithmic performance* (through nonlinear message functions and the ability to incorporate learned elements).
- *Maintained decentralization and interpretability*, since the core is still an optimization with a known objective, and convergence is analyzable.

This is a rapidly evolving area of research, and initial studies are promising in showing that SNNs can outperform classical GNNs on certain graph prediction tasks by better handling structured relation. Likewise, in multi-agent experiments, one can expect that incorporating sheaf-based design leads to more robust and flexible coordination behaviors than purely linear approaches.

11 Conclusion and Future Directions

In this paper, we presented an expository overview of cellular sheaves and their associated Laplacians, and showed how they provide a powerful framework for multi-agent coordination. We started from basic definitions (sheaves on graphs, sections, and Laplacians) and built up to advanced applications (nonlinear homological programs and distributed ADMM solvers). Throughout, we highlighted the intuition behind the mathematics: sheaves generalize graphs by attaching vector spaces and constraints, and sheaf Laplacians generalize graph Laplacians to enforce these constraints in a dynamical or optimization context.

We demonstrated how classical coordination tasks like consensus and formation control can be formulated in this framework, and how a unified distributed algorithm can solve all of them by exploiting convexity and locality. A significant portion of our discussion was devoted to drawing connections between this algorithm and neural network concepts, which serve to deepen understanding and pave the way for new hybrids of learning and control.

Several key takeaways are:

- Cellular sheaves allow encoding of complex agent relationships, overcoming limitations of simple graph models (especially for heterogeneous systems).
- The nonlinear sheaf Laplacian provides a principled way to design distributed controllers that achieve global objectives while respecting local constraints, bridging spectral graph theory with nonlinear control.
- ADMM on sheaf-structured problems yields a message-passing algorithm that is naturally decentralized and comes with convergence guarantees under convexity assumptions.
- Viewing the iterative solver as a neural network unveils opportunities to integrate learning, enabling systems that can potentially adapt to unknown conditions by tuning their local cost functions or interaction potentials.

Future Directions: This area is ripe with avenues for further research. A few prominent ones include:

1. *Extensions to Time-Varying and Stochastic Environments:* Real multi-agent systems operate in changing conditions. One can extend the sheaf framework to time-evolving sheaves (where the graph or restriction maps change over time) or uncertain data (where measurements are noisy, suggesting a need for stochastic potentials or Bayesian interpretations).
2. *Higher-Dimensional Sheaves:* We mainly discussed sheaves on graphs (1-dimensional cell complexes). The theory extends to sheaves on higher-dimensional cell complexes (hypergraphs or simplicial complexes). This could model multi-agent interactions involving more than two agents at once (e.g., a constraint among a trio of agents). The concept of a sheaf Laplacian also extends (leading to higher Hodge Laplacians). Distributed coordination on such complexes (perhaps using tools from higher-order network theory) is an exciting frontier.
3. *Learning Restriction Maps from Data:* In some cases, one might not know the exact linear relationship between agent states that should hold; one could try to learn the linear maps $\mathcal{F}_{i\to e}$ from demonstrations of coordinated behavior (as was explored in work on learning sheaf Laplacians from signals on graphs by Hansen & Ghrist). This would reverse-engineer the constraints that a group of agents is implicitly following.

4. *Hardware Implementation and Scalability:* As systems grow large (e.g., swarms of hundreds of drones), implementing the sheaf computations efficiently is important. Graph GPU libraries and neural network hardware could be leveraged given the similarities we've noted. Investigating the scalability and real-time performance of these algorithms in large networks is crucial for practical adoption.

5. *Integration with Control Theory:* Finally, connecting this more with continuous control: for instance, using the sheaf Laplacian flow as a controller in continuous time, or analyzing stability in the presence of communication delays and quantization. The rich structure may allow more precise statements about robustness (since $\ker L_{\mathcal{F}}$ characterizes the steady-state agreement exactly, and one can possibly quantify how perturbations move the system off that manifold).

We believe that the cellular sheaf perspective will become increasingly important as networks and systems become more complex and require integrated approaches from topology, algebra, optimization, and learning. By being both a comprehensive tutorial and a pointer to recent advances (such as those by Hanks et al. and Riess), we hope this paper serves as a stepping stone for researchers and practitioners to explore and apply sheaf-based methods in their own domains.

References

[1] Tyler Hanks, Hans Riess, Samuel Cohen, Trevor Gross, Matthew Hale, and James Fairbanks. *Distributed Multi-Agent Coordination over Cellular Sheaves.* arXiv:2504.02049, 2025 (under review).

[2] Hans Riess. *Lattice Theory in Multi-Agent Systems.* Ph.D. thesis, University of Pennsylvania, 2023. Also available as arXiv:2304.02568.

[3] Jakob Hansen. *A Gentle Introduction to Sheaves on Graphs.* Preprint, 2020. Available at `http://www.jakobhansen.org/publications/gentleintroduction.pdf`.

[4] Justin M. Curry. *Sheaves, Cosheaves and Applications.* Ph.D. thesis, University of Pennsylvania, 2014. Available as arXiv:1303.3255.

[5] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers." *Foundations and Trends in Machine Learning* 3(1):1–122, 2011.

[6] Jakob Hansen and Thomas Gebhart. "Sheaf Neural Networks." In *NeurIPS 2020 Workshop on Topological Data Analysis and Beyond*, 2020. arXiv:2012.06333.

[7] Federico Barbero, Cristian Bodnar, Haitz Sáez de Ocáriz Borde, Michael M. Bronstein, Petar Veličković, and Pietro Liò. "Sheaf Neural Networks with Connection Laplacians." In *Proceedings of the ICML 2022 Workshop on Topology, Algebra, and Geometry in Machine Learning*, PMLR 196:1–13, 2022. arXiv:2206.08702.

[8] Cristian Bodnar, Francesco Di Giovanni, Benjamin Paul Chamberlain, Pietro Liò, and Michael M. Bronstein. "Neural Sheaf Diffusion: A Topological Perspective on Heterophily and Oversmoothing in GNNs." In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022. arXiv:2202.04579.

[9] Jakob Hansen and Robert Ghrist. "Learning Sheaf Laplacians from Smooth Signals." In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2019)*, pp. 5446–5450. DOI: 10.1109/ICASSP.2019.8683709.