AN INTRODUCTION TO COHOMOLOGY AND ITS APPLICATIONS

CHUANHAO LI , HAOLIN LI , YAXIN LIU , L. BERTÓK , AND O. WEBB
SUPERVISOR: KAIBO HU

Abstract. Cohomology is a tool combining abstract mathematics from many fields; algebraic topology, group theory, graph theory and geometry intersect to provide a branch of mathematics capable of yielding powerful insights into a range of applications. This report shows how these abstract theorems are built up and can be used to study various systems. The wide use of graphs provides a broad canvas to put Cohomological formalism to work; here it is shown how graph cohomology groups provide a view into the structure of the topological space. Further to this, it is demonstrated how Cohomology can be seen as the foundation of ranking algorithms such as Hodge Rank.

Key words. Cohomology, Graphs, Topology, Groups, Hodge Rank, Topological data analysis

1. Introduction. Traditional applied mathematics has extensively relied on partial differential equations (PDEs) to model a myriad of physical phenomena, and computational mathematics has primarily focused on their numerical solutions [1]. Nevertheless, the effective application of PDEs demands a substantial and precise understanding of the phenomena being modeled, which is not always feasible. For numerous datasets, we often possess only a rudimentary measure of similarity between data points and an overarching sense of the dataset's distribution[2].

In contexts where differential equations fall short due to a lack of detailed information, topology emerges as a potent alternative. Topology, the study of properties preserved through continuous deformations, offers a unique perspective for understanding complex structures. This is where cohomology, a central concept in algebraic topology, becomes invaluable. Cohomology provides a powerful framework for classifying topological spaces up to homotopy equivalence, extending its utility well beyond the confines of pure mathematics. It has profound implications in theoretical physics, especially in string theory and quantum field theory, by revealing insights into the cohomological properties of spaces and the underlying structures of complex systems.

With the emergence of topological methods in data analysis, cohomology has proven to be a crucial interdisciplinary tool. The versatility of cohomological approaches allows for meaningful analysis of data where traditional mathematical models may be insufficient. In particular, the Hodge Laplacian on a graph and the concept of Hodge rank have become central to topological data analysis [2], offering novel insights into the structure and features of complex datasets. These methods are particularly relevant in the fields of algebraic geometry and number theory, where they contribute to a deeper understanding of sheaves, schemes, and étale cohomology [3].

The utilization of cohomological techniques in data science signifies a paradigm shift, underscoring the significance of topological perspectives in the analysis of high-dimensional data. This shift is evident in the increasing reliance on computational methods to navigate the abstract landscape of cohomology. This article seeks to explore this integration of computational tools with cohomological theories, thereby highlighting how algorithmic strategies can enhance our comprehension of topological invariants and facilitate the application of cohomology in various domains. We aim to detail specific computational techniques that have been pivotal in advancing cohomological studies, thereby enabling a more profound and nuanced exploration of this expansive field.

2. Theoretical Background. Consider the property

$$\mathrm{AB} = \mathbf{0}\,. \tag{2.1}$$

(*think*: curl grad = **0**)

By definition, this is equivalent to the property

$$\mathrm{im(B)} \subseteq \ker(\mathrm{A})\,, \tag{2.2}$$

and the *cohomology group* is defined via these two sets as the quotient space

$$\ker(\mathrm{A})/\mathrm{im(B)}\,. \tag{2.3}$$

Elements of this space are *equivalence classes* [2] or *cosets*, e.g. $x + \mathrm{im(B)}$, for a given $x \in \ker(\mathrm{A})$. This is quite an elegant theoretical definition, but we get into trouble if we try to go about computing it. It may serve us well to derive some equivalent, simpler algebraic expression. This can be done by picking a *harmonic representative* $x_H \in x + \mathrm{im(B)}$ from each class such that $x_H \perp \mathrm{im(B)}$. By the identity $\mathrm{im}(B)^\perp = \ker(B^*)$, we have that $x_H \in \ker(\mathrm{A}) \cap \ker(\mathrm{B}^*)$. It can be show that there is a natural correspondence between x_H and its equivalence class and so

$$\ker(\mathrm{A})/\mathrm{im(B)} \cong \ker(\mathrm{A}) \cap \ker(\mathrm{B}^*)\,. \tag{2.4}$$

One may then form the *Hodge Laplacian* as

$$\mathrm{A}^*\mathrm{A} + \mathrm{B}\mathrm{B}^*\,,$$

and it can be further shown that indeed

$$\ker(\mathrm{A}^*\mathrm{A} + \mathrm{B}\mathrm{B}^*) = \ker(\mathrm{A}) \cap \ker(\mathrm{B}^*)\,. \tag{2.5}$$

As such, an x_H as defined previously is a solution to the *Laplace equation*

$$(\mathrm{A}^*\mathrm{A} + \mathrm{B}\mathrm{B}^*)x_H = 0\,, \tag{2.6}$$

which justifies calling it a *harmonic* representative. Now to compute the cohomology, all we need is to somehow find the operators A and B, form the Hodge Laplacian and compute its kernel. As we are dealing with finite dimensional spaces, linear operators are equivalent to matrices and the adjoint is simply the transpose, i.e. $A^* = A^\top$.

Now we describe the vector spaces under consideration. A *graph* is a pair (V, E) where V is the set of vertices and E is the set of edges. We can draw a graph, say, as in Fig. 1. This implicitly also leads to the notion of n-*cliques* of a graph, that is *complete* subgraphs (Fig. 2). We have indicated the 3-cliques as T in Fig. 1, but note that this is not part of the definition of a graph, merely some additional structure we have noticed on it.

To be able to describe some graph theoretic notion of cohomology, we need to somehow define the operators A and B in some sensible way, such that they satisfy Eq. 2.1. In fact, A and B are required to be *boundary operators* (such as curl or grad) and as such Eq. 2.1 can be stated as *the boundary of a boundary is zero* [4]. This is what we would like to enforce. To this end, we can consider the sets V, E, T, ..., as vector spaces over $\mathbb{R}$, with the right operations. We can then define scalar valued functions on them, say, on the set of vertices $f : V \to \mathbb{R}$, the set of edges $X : E \to \mathbb{R}$ or on the 3-cliques $\Phi : T \to \mathbb{R}$. (Notice, that $E \subseteq V \times V$, $T \subseteq V \times V \times V$ etc.) We require these functions to be *alternating*, that is,

$$X(i, j) = -X(j, i) \tag{2.7}$$

for all $\{i,j\} \in E$ and

$$\Phi(i,j,k) = \Phi(j,k,i) = \Phi(k,i,j) = -\Phi(i,k,j) = -\Phi(k,j,i) = -\Phi(j,i,k) \tag{2.8}$$

for all $\{i,j,k\} \in T$. The reason for using alternating forms is that while graphs have a notion of *cliques*, topology has closely related notion of *simplices*. An important distinction (at least to our goals) is that simplices have an *orientation*, while our cliques are undirected. Thus introducing alternating functions on them defines, in an unambiguous way, some parity, that is, direction. These functions are called 0-, 1-, 2-*cochains* and are a discrete form of *differential forms* on manifolds [5]. (*NB* k-cliques are called $(k-1)$ simplices in topology.)

We can then introduce operators on our newly defined functions, for example graph theoretic analogues of grad, div, and curl. One may (but not by necessity) define these as such

$$(\operatorname{grad} f)(i,j) = f(j) - f(i)\,, \tag{2.9}$$

$$(\operatorname{div} X)(i) = \sum_{j=1}^{n} \frac{w_{ij}}{w_i} X(i,j)\,, \tag{2.10}$$

$$(\operatorname{curl} X)(i,j,k) = X(i,j) + X(j,k) + X(k,i)\,. \tag{2.11}$$

Note, that now we are operating on spaces of functions, using the L_2 inner product. Thus $\operatorname{grad} : L_2(V) \to L_2(E)$, while $\operatorname{div} : L_2(E) \to L_2(V)$ and $\operatorname{curl} : L_2(E) \to L_2(T)$. Operators defined this way do in fact have the required property: $\operatorname{curl}\operatorname{grad} = \mathbf{0}$. This allows us now to form the *graph Helmholtzian* and by the fact that $\operatorname{div} = \operatorname{grad}^*$ we have

$$\Delta_1 = \operatorname{curl}^* \operatorname{curl} - \operatorname{grad}\operatorname{div} = \operatorname{curl}^* \operatorname{curl} + \operatorname{grad}\operatorname{grad}^* \tag{2.12}$$

Which is analogous to the *Hodge Laplacian*. In general these operators are called *coboundary operators* and an operator from k-simplices to $(k+1)$-simplices is denoted by δ_k. Now our required property is

$$\delta_k \delta_{k-1} = 0\,, \tag{2.13}$$

and the k-th cohomology group of the graph G is defined as

$$H^k(G) = \ker(\delta_k)/\operatorname{im}(\delta_{k-1})\,. \tag{2.14}$$

It can be shown, similarly as before, that

$$\ker(\delta_k)/\operatorname{im}(\delta_{k-1}) \cong \ker(\delta_k^*\delta_k + \delta_{k-1}\delta_{k-1}^*)\,. \tag{2.15}$$

Now all that is necessary is to compute the cohomology is to define, in an unambiguous way, the cochains.

3. Computing Graph Cohomology Groups. Once sufficient understanding of the topological nature of coboundary operators has been developed, the graph cohomology groups can be computed. Graphs can take many forms, and it is necessary to limit the scope of the graphs being used [6]. The code that has been developed applies only to non-planar graphs, where all edges form cycles of dimension no larger than 3.

Despite this simplification, the utility of such a graph is still great; these graphs are made up of a mesh of 3-cycles, and are therefore applicable in finite element methods [7] - computing the cohomology can therefore tell us about the nature of the mesh and whether it is suitable for a given problem.

Expanding to planar graphs, or graphs with cycles of dimension greater than 3, would greatly increase complexity and run time of the code. When finding the faces of a graph, given that the graph is non planar, and is made up of 3-cycles, the developed code finds faces by following edges back to a starting point. For a triangle there are three edges, or two vertices to traverse before returning to the starting vertex. If there is no restriction on the dimension of a cycle, then it is necessary to search for vertices connected by a very large number of edges. Similarly, for a non-planar graph, due to topological embedding, the faces of the graph given simply the adjacency matrix are ambiguous [8]. An example can be seen in Figure 3, where, for the same adjacency matrix, the graph can be embedded and shown to have 4 faces. Although easily recognized here, potential applications could have more subtle violations.

3.1. The Algorithm. Since the given code will be taking in the adjacency matrix for a graph (a 1D simplical complex) the coboundary operators are only applicable for mapping vertices to edges, and edges to faces; given there are no other topological objects in the space of a higher dimension than faces. Therefore, it is only H^0 and H^1 that are computed. To form the coboundary operator matrix for H^0, the following algorithm is applied:

Algorithm 3.1 Form Coboundary Operator Matrix

Input: Adjacency Matrix A ▷ Entries indicate edge between vertices i, j
Initialize C as a zero matrix of size $|E| \times |V|$
for each edge $e_k = (v_{ia}, v_{ib})$ in E **do**
    Identify vertices v_{ia} and v_{ib} that e_k connects
    Find the index i of v_{ia} in V
    Find the index j of v_{ib} in V
    Set $C[k][i] = -1$ ▷ Assuming the edge is directed from v_{ia} to v_{ib}
    Set $C[k][j] = 1$ ▷ If the graph is undirected, choose any direction
end for
return C

A similar algorithm is applied to find H^1, although this will iterate through faces to find edges that are connected through that face. As later discussed, it is the computation of the faces that comes at the expense of significant computational cost. Once these coboundary operator matrices have been formed, the cohomology groups can be calculated as shown in Algorithm 3.2 below.

In the final steps, the pivot and non-pivot columns are used to represent the kernel and image in order to compute H^0 and H^1.

3.2. Computational Cost. With the aforementioned restrictions, the computational cost of the developed code will likely be less than that for more complicated

Algorithm 3.2 Compute Graph Cohomology Groups

1: **Input:** Adjacency Matrix A ▷ Entries indicate edge between vertices i, j
2: Extract Upper Triangular Adjacency
3: Generate Coboundary Operator for H^0: C^0
4: Identify Cycles
5: Generate Coboundary Operator for H^1: C^1
6: Compute RREF of C^0, C^1
7: Identify Pivot and Non-Pivot Columns
8: Compute H^0 & H^1 Cohomology Groups
9: **return** H^0 & H^1

code which searches for cycles of any dimension on a planar graph. Not all planar graphs will have a well defined embedding [9], so exact computational cost for a planar graph is ambiguous. When searching for 3-cycles, the code iterates through the vertices to search for those connected by edges - $O(n^3)$ - to calculate the kernel and image the matrices are put into RREF - $O(n^3 + n^2)$. The total computational cost is therefore:

$$C(n) = O(2n^3 + n^2)\,. \tag{3.1}$$

Other computations such as assigning values to the coboundary operator matrices are overshadowed compared to the cost of finding the faces and computing the RREF of the coboundary operator matrices, so these have been ignored. The cost seems high given the restrictions made, although the task of finding all 3-cycles for a given graph, along with then finding the reduced echelon form of the matrices would demand a high computational cost. Logistically, three loops over the vertices to find 3-cycles would result in an n^3 dependence. There are no assumptions regarding connectivity between vertices; they can be connected in any combination provided they form 3-cycles.

Considering ways to reduce computational cost, simplifications based on connectivity between adjacent vertices is limited given the aforementioned lack of assumptions. A different input of graph data may reduce this complexity, although would perhaps require more prepossessing, and would move away from the conventional use of an adjacency matrix in graph theory. Generally, greater awareness of the graph being analysed, such as the number of edges per vertex or the connectivity between vertices, would allow for a reduction in computational cost. But to compute the graph cohomology groups of any non-planar graph of 3-cycles, $C(n) \sim O(n^3)$ is to be expected.

Results can be seen in Figure 4 showing the time taken to compute the graph cohomology groups for a random graph of 3-cycles with n vertices. To guide the reader, a plot of $2n^3 + n^2$ has also been included for comparison. The time was an average over a number of different matrices.

3.3. Literature Comparison. Given the novelty of the task at hand, literature, or previous examples of computation of graph cohomology groups is sparse. There is more literature regarding homology groups, although to verify these results a comparison to code finding the faces of a graph has been included.

In this example [10], faces of a planar graph were found under assumptions of existence of the graph embedding. It inputs an ordered list of vertices and edges in a coordinate system such that their location is known. The finds spaces through sorting the vertices by polar angle and traversing the space clockwise or anti-clockwise

in order to find adjacent connected vertices. This is all implemented in C++.

Results are shown in Figure 5. Plots of our face finding loop, along with known functions have been included. The literature algorithm is cited as having a computational cost of $C(n) = nlog(n)$. It is clearly faster than our face finding loop - developed in Python - for all sizes of graph (number of vertices). To ensure comparison, edges between vertices were chosen at random in accordance with the restrictions, and time readings were taken over many repeats to ensure a range of edges and faces.

Discrepancies between the two face finding algorithms arise chiefly from the nature of the graphs that they are applied on. The Python algorithm, is looping through to find all combinations of connections between vertices, this is necessary given on a non-planar graph any two vertices can be connected. The C++ algorithm approaches in a more creative manner, given restrictions on connectivity between vertices. Assuming the unique embedding of the graph will greatly restrict the uses of this C++ algorithm, but these assumptions pay off in the form of greatly increased performance. Finally, it should be mentioned that a compiled language (C++) will often be faster than an interpreted one such as Python [11]; type checking, memory management and low level access lead to differences in performance that are seen at low graph sizes.

4. Literature Review. The framework of Hodge theory and its connection to graph theory has been extensively studied, leading to significant developments in topological data analysis. In this review, we highlight key concepts and theorems that are foundational to understanding the Hodge Laplacian on graphs.

DEFINITION 4.1 (Cochain Complex). *A sequence of cochain groups $\{C^k\}$, connected by coboundary operators $\{\delta^k\}$, that forms a complex, i.e., $\delta^{k+1} \circ \delta^k = 0$ for all k.*

THEOREM 4.2 (Hodge Decomposition). *For a finite-dimensional inner product space V and a self-adjoint, linear operator $\Delta : V \to V$, the space can be decomposed into the direct sum of the kernel of Δ and the image of Δ.*

THEOREM 4.3 (Kernel of the Hodge Laplacian). *Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ with $AB = 0$, then the kernel of the Hodge Laplacian $A^*A + BB^*$ is equal to the intersection of the kernels of A and B^*.*

THEOREM 4.4 (Cohomology Group). *For a graph G, the k-th cohomology group $H^k(G)$ is defined as the quotient space $ker(\delta^k)/im(\delta^{k-1})$, where δ^k is the k-th coboundary operator.*

These foundational elements lay the groundwork for exploring the applications of Hodge theory to the study of graph Laplacians. They provide the necessary tools to dissect the complex structures that underpin our understanding of topological spaces.

5. Application. Traditionally, partial differential equations and their numerical solutions have been the cornerstones of computational mathematics, predicated on a detailed and accurate understanding of phenomena to construct and express models as PDEs. However, in contemporary data applications, we are often equipped merely with rough metrics for gauging similarities between data points and the distribution of datasets. It is here that topology finds its utility. Within computational mathematics, discrete partial differential equations typically arise from the numerical treatment of their continuous counterparts, laying the groundwork for cohomology theory in numerical analysis [12, 13, 14]. Our study mentions the versatile applications of cohomology and Hodge rank, foundational concepts in algebraic topology, across various domains, underlining the breadth and depth of topological methods in data analysis

with further details to follow in the section on Hodge rank.

One prominent application of cohomology and Hodge rank is in the field of ranking systems[15, 16, 17]. Ranking, an essential feature in numerous industries, benefits from these topological tools to order entities based on multi-faceted relationships and interactions. For instance, search engines utilize these concepts to rank web pages by interpreting the internet's complex hyperlinked structure as a high-dimensional topological space[15]. In academic databases, they help determine the relevance and impact of scholarly articles. By employing cohomology and Hodge rank, these systems can discern the underlying structure of data, assigning ranks that reflect more than just numerical counts or simple metrics, thus enabling a more nuanced and context-aware ordering that can adapt to the ever-evolving data landscapes.

Beyond ranking, cohomology has proven invaluable in understanding and visualizing complex networks, such as those found in brain connectivity studies. By applying Hodge theory, researchers can analyze neural patterns and interactions[18], potentially unlocking new insights into cognitive processes and neurological disorders. This approach also extends to sensor networks[19], where cohomology can help optimize coverage and ensure efficient communication between nodes.

In the vast expanse of data science, cohomology and Hodge rank aid in tasks like image processing and object synchronization[20, 21], by allowing the extraction of high-level features that remain consistent under various transformations. These methods contribute to the robustness of algorithms in deep learning and dimension reduction, facilitating the distillation of essential information from large datasets. Such applications underscore the transformative impact of topological techniques, bringing a powerful abstract mathematical framework to bear on concrete, real-world problems.

6. Hodge Rank.

6.1. Background and Motivation.

6.1.1. Introduction to Hodge Rank in Cohomology. Hodge Rank, initially conceived for statistical ranking in varied domains such as sports and web search, has proven instrumental in cohomological studies due to its robust mathematical framework. The algorithm leverages Hodge theory to derive global rankings from subjective and often incomplete data sets, making it ideal for analyzing complex cohomological data where similar challenges of data incompleteness and voter bias exist.

6.1.2. Challenges in Cohomological Analysis. Cohomological analysis often grapples with data that is not only incomplete but also biased and imbalanced. Traditional methods struggle to handle these issues efficiently, leading to prolonged computation times and potentially inaccurate analyses. The growing scale and complexity of data in cohomological studies further compound these challenges, necessitating the development of more efficient computational methods.

6.1.3. Motivation for Adopting Hodge Rank. The motivation for integrating Hodge Rank into cohomology stems from its unique ability to process incomplete and biased data through its graph-based least squares approach. Here, we employ a grouping method that reduces computational complexity by first sorting within individual subgroups and then aggregating the rankings from all subgroups into a unified result. This approach effectively segments the data, allowing for more manageable and efficient processing of each subset before combining them, which not only simplifies the computational task but also enhances the overall accuracy and reliability of

the rankings. This method has demonstrated considerable success in reducing computational complexity from $O(n^3)$ to $O(\frac{n^3}{k^2})$ where k is the number of groups, by employing grouping methods that segment data into manageable subsets. This not only accelerates the computation but also maintains the integrity and accuracy of the rankings, as evidenced by its application to datasets like NBA team ranking.

This section establishes the rationale for utilizing Hodge Rank in cohomology, emphasizing the algorithm's suitability for handling the specific challenges of cohomological data analysis and the recent advancements that enhance its computational efficiency.

6.2. Methodology. This methodology section elucidates the application of the Hodge Rank algorithm to datasets characterized by incompleteness and bias. Utilizing graph theory and least squares optimization, the algorithm provides a statistically sound ranking of elements. Here we outline the approach and detail the computational steps involved in each of the three examples provided, referencing the Python code available in the Appendix.

6.2.1. General Approach. The Hodge Rank algorithm constructs a graph where nodes represent elements to be ranked, and edges symbolize the pairwise comparisons. The algorithm's objective is to minimize the discrepancies between these comparisons to determine a global ranking through a least squares problem. This is mathematically represented as minimizing the objective function $\|W^{1/2}(f - \delta^T r)\|^2$, where W is a diagonal matrix of edge weights, f is a vector of edge flows, δ denotes the graph's incidence matrix, and r is the sought global ranking vector.

6.2.2. Notations. Let V be a finite set. Denote $|V|$ as the cardinality of set V. The following notation is from combinatorics:

$$\binom{V}{k} := \text{the set of all } k\text{-element subset of } V.$$

Also suppose we have a set of m voters, denoted by $\Lambda = \{1, 2, ..., m\}$, and a set of n alternatives to be rated, $V = \{1, 2, ..., n\}$. Now, consider one single voter $\alpha \in \Lambda$, we will investigate how to build a graph for one voter first, and then for all voters, combining m graphs together. Also, for a matrix A, both $A(i, j)$ and A_{ij} denote the entry on the i-th row and j-th column of A.

6.2.3. For Single Voter. For voter $\alpha \in \Lambda$, let $V_\alpha \subseteq V$ be the set of alternatives that this voter has voted on. Define pairwise comparison function $f^\alpha \in \mathbb{R}^{|V_\alpha| \times |V_\alpha|}$, where

$$f^\alpha(i, j) := \text{voter } \alpha\text{'s preference between object } i \text{ and object } j \text{ for } i, j \in V_\alpha$$

Particularly, given the nature of our data, we will measure voter's preference in the following way:

$$\text{Ratings Difference: } f^\alpha(i, j) = R(\alpha, i) - R(\alpha, j),$$

which is the subtraction between two cardinal scores.

If we use V_α as the set of vertices in the graph for voter α, denoted by G_α, then f^α can be regarded as the set of edges in the form of edge flows. Thus, we can create graph $G_\alpha = (V_\alpha, E_\alpha)$. Notice that this is a complete graph with $|V_\alpha|$ vertices and $\binom{|V_\alpha|}{2}$ edges.

6.2.4. For Multiple Voters. Given G_α, now we consider how to combine these graphs together to form $G = (V, E)$. Intuitively, V should include all the alternatives that each vote has voted on. In other words, $V = \bigcup_{\alpha\in\Lambda} V_\alpha$ and $E = \bigcup_{\alpha\in\Lambda} E_\alpha$. For edge flow f, we define

$$f(i,j) = \frac{1}{|\Lambda_{ij}|} \sum_{\alpha\in\Lambda_{ij}} f^\alpha(i,j) = \frac{1}{|\Lambda_{ij}|} \sum_{\alpha\in\Lambda} f^\alpha(i,j),$$

where Λ_{ij} is a subset of voters who have particularly voted between alternatives i and j.

In other words, f takes the average of all the votes between each pair of alternatives. However, it is easy to see that if we combine all the edges together in this way, these edges have different levels of importance and thus extra consideration is needed. For example, suppose $i, j \in V$ are popular alternatives such that a lot of the voters have voted on them, while only one voter has voted for $k, l \in V$. Thus, $e = (i, j) \in E$ is more important than $e' = (k, l) \in E$ because e represents more votes; we need to introduce a "weight" to the edges in order to account for this difference in popularity.

For all $e = (i, j) \in E$, define $\omega_e = |\Lambda_{ij}|$. Thus, if more voters have voted between i and j, that particular edge will become more important in our graph. Formally, we form the weighted graph $G = (V, E, \omega)$.

Upon the introduction of ω, it is evident that our data is unbalanced. There will inevitably be some unpopular alternatives that are seldom voted on and others that are more regularly voted on given any dataset of user-rated items (think of two Hollywood films and two low-budget films). We must make a weighted graph in order to account for this discrepancy.

6.2.5. COVID-19 Symptom Ranking Example. The ranking of COVID-19 symptoms based on severity, derived from biased and incomplete patient data, demonstrates the Hodge Rank algorithm's adaptability.

Data Collection. Symptom severity ratings, such as fever, cough, and nausea, were collated from patients, with the dataset exhibiting instances of missing values.

Graph Construction. Symptoms were represented as nodes in a graph, with edges indicating the differential severity ratings assigned by patients.

	Fever	Sore Throat	Cough	Nausea
Patient 1	3	2	2	5
Patient 2	7	8	9	X
Patient 3	2	2	1	3

Table 1: Multiple Voters Example on Covid-19 Symptoms

There is some terminology needed to understand the method, which we detail here. Following the notation used in Jiang et al. [17] and Colley et al. [22], we define Λ to be the set of voters and V to be the set of elements that are voted on. For $\alpha \in \Lambda$, we denote V_α to be the set of elements rated by voter α. Similarly, we let Λ_{ij} denote the set of voters who rated both elements i and j. In our example, the following is the case:

$$\Lambda = \{\text{Patient 1, Patient 2, Patient 3}\}$$

$$V = \{\text{Fever, Sore Throat, Cough, Nausea}\}$$

$$V_{\text{Patient 2}} = \{\text{Fever, Sore Throat, Cough}\}$$

Also, we define the rankings as $R : \Lambda \times V \to \mathbb{R}$. For example, if voter α gave element i a score of 5, we would say $R(\alpha, i) = 5$.

Using the elements in V_α as nodes, we can form a complete graph for every voter where each node represents an element. Note in the example in Figure 6, we replace the name of each symptom (fever, sore throat, cough, nausea) with letters (A, B, C, D), respectively.

Let E denote the set of all edges. For every edge, we define an orientation by indiscriminately designating one node to be the sink node and the other to be the source node. To keep things simple, we let nodes which are indexed earlier be the source nodes.

The relationship between pairs of nodes is described with the pairwise comparison function in Figure 7, $f^\alpha(i, j)$ where α is a voter.

$$f^\alpha(i, j) = R(\alpha, j) - R(\alpha, i)$$

We can now define one graph, G, pertaining to all voter's data. G is a complete graph containing every alternative in V, so long as it's been voted on, as well as $\binom{|V|}{2}$ edges.

We should take into higher consideration pairs of elements which were voted on by many people. With this in mind, we define the weights for each edge as the number of voters who rated both alternatives:

$$\omega_{ij} = |\Lambda_{ij}|$$

where Λ_{ij} is the set of voters who rated both i and j. The edge flow of the entire group's graph, $f : V \times V \to$, represents the average pairwise difference of each edge:

$$f(i, j) = \frac{1}{|\Lambda_{ij}|} \sum_{\alpha \in \Lambda_{ij}} f^\alpha(i, j)$$

We can now define one graph, G, pertaining to all voter's data. G is a complete graph containing every alternative in V, so long as it's been voted on, as well as $\binom{|V|}{2}$ edges.

We should take into higher consideration pairs of elements which were voted on by many people in Figure 7. With this in mind, we define the weights for each edge as the number of voters who rated both alternatives:

$$\omega_{ij} = |\Lambda_{ij}|$$

where Λ_{ij} is the set of voters who rated both i and j. The edge flow of the entire group's graph, $f : V \times V \to$, represents the average pairwise difference of each edge:

$$f(i, j) = \frac{1}{|\Lambda_{ij}|} \sum_{\alpha \in \Lambda_{ij}} f^\alpha(i, j)$$

6.2.6. Least Squares Problem. Our goal is to find a universal rating $r : V \to \mathbb{R}$ that maps every element to its relative rating. By comparing r to the data processed

in our graph, we can judge the efficacy of our rating. A good choice for r should agree highly with our edge flow, so we must minimize:

$$f(i,j) - (r(j) - r(i)) \tag{6.1}$$

While minimizing Equation 6.1 , we should also take into account the number of voters who evaluated both i and j. If a lot of voters evaluated both elements, we should take that edge into greater consideration than if not many evaluated it. This handles the imbalanced nature of our data.

With this in mind we arrive at the function we use to judge the efficacy of any r:

$$\sum_{i,j \in V} \omega_{ij}(f(i,j) - (r(j) - r(i)))^2 \tag{6.2}$$

Notably, multiplying by ω_{ij} does not boost any element's rating. Instead it places more importance (or lack thereof) on the balance of $f(i,j)$ and the difference of the universal ratings.

6.2.7. Hodge Decomposition. In this section, we demonstrate how Hodge decomposition is used to show that it is possible to find a ranking $\tilde{r} \in \mathbb{R}^{|V|}$ (where $\tilde{f}$ is the vectorized form of r indexed by V) and local consistency $\tilde{c} \in \mathbb{R}^{|E|}$ for any $f \in \mathbb{R}^{|E|}$.

First, we must define the boundary operators of the graph:

- The negative divergence, denoted by ∂, $\mathbb{R}^{|E|} \to \mathbb{R}^{|V|}$, is defined as:

$$(\partial)_j = \begin{cases} -1 & \text{if } u_j \text{ is the source node in } e_j, \\ 1 & \text{if } u_j \text{ is the sink node in } e_j, \\ 0 & \text{else.} \end{cases}$$

  Note the divergence is simply ∂^T.
- The curl, δ, $\mathbb{R}^{|E|} \to \mathbb{R}^{|E|}$, is defined as:

$$(\delta)_j = \begin{cases} 1 & \text{if } e_j \text{ is in } T_j \text{ with same orientation as } T_j, \\ -1 & \text{if } e_j \text{ is in } T_j \text{ with same orientation as } T_j, \\ 0 & \text{else.} \end{cases}$$

The 1-Hodge Laplacian is $L_1 = \delta^T\partial + \partial\delta$. Due to Hodge decomposition [2; 17; 11], we can show the following:

$$\mathbb{R}^{|E|} = \text{im}(\delta^T) \oplus \ker(L_1) \oplus \text{im}(\partial)$$

Therefore for any $f \in \mathbb{R}^{|E|}$ we can find $\tilde{r} \in \mathbb{R}^{|V|}$, $\tilde{c} \in \mathbb{R}^{|E|}$, and $\tilde{h} \in \ker(L_1)$ such that:

$$\tilde{f} = \delta^T\tilde{c} + \partial\tilde{h}$$

This shows that for any f' we'd be able to find a ranking $\tilde{r}$ and a local consistency $\tilde{c}$. An extensive explanation of Hodge decomposition can be found in [2], with implementations demonstrated in [17, 22].

6.2.8. Solving with Linear Algebra. First, let $\tilde{u}$ be the vectorized version of u indexed by E. Using the negative divergence of the system, the minimization becomes:

$$\min_{\tilde{r}\in\mathbb{R}^{|E|}} \|\tilde{f} - \partial^T \tilde{r}\|_W^2$$

where W is the diagonal matrix whose entries are $\tilde{u}$.

Using some basic calculus, the minimization reduces to the following:

$$\partial W \partial^T \tilde{r} = \partial W \tilde{f}$$

The only unknown in this equation is $\tilde{r}$ so this is an $A\tilde{x} = \tilde{b}$ problem. The matrix $\partial W \partial^T$ is a well-studied matrix called the graph Laplacian, which has no inverse, so when solving for $\tilde{r}$ the pseudo-inverse must be taken. The time complexity of this operation is $O(n^3)$ where $n = |V|$.

Without detailing the derivation, the solution of $\tilde{c}$ follows the same pattern. The minimization problem reduces to:

$$\min_{\tilde{c}\in\mathbb{R}^{|E|}} \|\tilde{f} - \partial\tilde{c}\|_W^2$$

Similarly, this reduces to:

$$\partial^T W \partial \tilde{c} = \partial^T W \tilde{f}$$

This equation has one unknown, $\tilde{c}$, which can be solved for with complexity $O(n^3)$, where $n = |E|$.

6.2.9. Grouping Method. The first step in the grouping method is to obtain a naive ranking of elements, which we denote $r_0 : V \to \mathbb{R}$. There are various choices you could make for this first step.

1. **Arithmetic mean of rating**: In this ranking, elements are ordered by their average rating. This is similar to sorting search results by "top rated".

$$r_0(i) = \frac{\sum_{\alpha\in A_i} R(\alpha, i)}{|A_i|}$$

2. **Arithmetic mean of edge flow**: Here r_0 is the result of averaging the edge flow between one node and every other node. The edge flow refers to f which we defined in Equation 3.

$$r_0(i) = \frac{1}{|V|}\sum_{j\in V} f(j, i)$$

Let V be the set of elements and $\{V_1, V_2, \ldots, V_k\}$ be the set of subgroups like in Figure 9. Then we let W_n, for n in $1, 2, \ldots, k$, denote the set of nodes that HodgeRank will run on such that $W_n = \{v_n | v \in V, m \neq n\}$. We modify the definitions of edge flow and edge weight to suit the introduction of subgroups as pseudo-nodes:

For the edge flow $f(i, j)$, we define it as:

$$f(i,j) = \begin{cases} \frac{1}{|A_{ij}|}\sum\limits_{\alpha\in A_{ij}} f^\alpha(i,j) & i \text{ and } j \text{ are single nodes} \\ \frac{1}{\sum\limits_{u\in V_i}|A_{uj}|}\sum\limits_{u\in V_i}\sum\limits_{\alpha\in A_{uj}} f^\alpha(u,j) & i \text{ is a subgroup and not } j \\ \frac{1}{\sum\limits_{u\in V_i}\sum\limits_{v\in V_j}|A_{uv}|}\sum\limits_{u\in V_i}\sum\limits_{v\in V_j}\sum\limits_{\alpha\in A_{uv}} f^\alpha(u,v) & i \text{ and } j \text{ are subgroups} \end{cases} \tag{6.3}$$

And for the edge weight $w(i,j)$, it is given by:

$$w(i,j) = \begin{cases} |A_{ij}| & \text{if } i \text{ and } j \text{ represent single nodes} \\ \sum_{u \in V_i} |A_{uj}| & \text{if } i \text{ represents a subgroup and not } j \\ \sum_{u \in V_i} \sum_{v \in V_j} |A_{uv}| & \text{if } i \text{ and } j \text{ represent subgroups} \end{cases}$$

Finally, we can run HodgeRank on each W_n, which will give us k rankings. We achieve our final ranking by stacking the groupings' rankings on top of each other according to their order from the naive rank.

6.3. Application in Covid-19 Dataset.

Dataset Introduction. The information, which was gathered from 5700 respondents, describes characteristics of health complaints that are grouped according to their severity. The information contains the following details and makes a distinction between non-severe and severe symptoms:

- There are 14 symptoms monitored.
- The symptoms are classified into two tiers of severity: non-severe and severe.
- Out of the 5700 respondents, 1376 reported severe symptoms, while 4324 reported non-severe symptoms.
- The data is non-individualized, meaning it aggregates the number of reports per symptom rather than tracking individual patient data.
- The information was collected in March 2020.

The following table lists the symptoms along with the proportion of respondents who reported them for each severity category:

Symptom	Non-Severe	Severe
Fever	1216	3520
Cough	978	2841
Fatigue	830	1911
Dyspnea	608	246
Sputum	517	1211
Shortness	491	553
Myalgia	358	566
Chill	358	471
Dizziness	222	523
Headache	155	584
Sore	107	419
Nausea	81	246
Diarrhea	78	251
Congestion	39	221

Table 2: Symptom Prevalence by Severity

This table 2 provides a count of how many respondents reported each symptom and allows for analysis of symptom prevalence as related to the severity of their condition. The data could be instrumental in understanding symptom patterns and their association with disease severity, especially within the context of the time when the data was collected, which coincides with the onset of the COVID-19 pandemic.

Algorithm Application. The algorithm processed these ratings to solve for the universal ranking vector $\mathbf{r}$, effectively handling incomplete data. This process is captured in the Python code provided in Appendix section B.

Results and Conclusion. In comparison, the Naive rating column shows generally higher values across the board. Notably, symptoms like chills, shortness of Breath, and myalgia have Naive ratings highlighted in red, indicating significant divergence from the HodgeRank rating, potentially suggesting that the Naive approach overestimates their relative importance.

Symptom	HodgeRank rating ($\tilde{r}$)	Naive rating
Fever	1.53	3.52
Cough	1.28	2.84
Fatigue	0.875	2.18
Sputum Production	0.5	1.36
Dyspnea	0.25	0.967
Chills	0.219	0.744
Shortness of Breath	0.156	0.959
Myalgia	0.125	0.780
Dizziness	0.0313	0.577
Headache	0	0.522
Nausea	-0.031	0.250
Sore Throat	-0.094	0.372
Diarrhea	-0.156	0.241
Congestion	-0.156	0.170

Table 3: Comparison of HodgeRank and Naive Ratings for Symptoms

6.4. NBA Team Scoring. The algorithm's application to NBA team scoring further highlights its utility in sports analytics.

Data Collection. The data included scores from NBA games, translating into a graph representation where each team is a node and edges correspond to score differences.

GAME_DATE_EST	HOME_TEAM_ID	VISITOR_TEAM_ID	SEASON	PTS_home	PTS_away
2020/3/12	1610612748	1610612750	2019	104	113
2020/3/12	1610612741	1610612739	2019	101	91
2020/3/12	1610612759	1610612754	2019	108	119
2020/3/12	1610612744	1610612749	2019	122	109
2020/3/12	1610612743	1610612761	2019	115	127
2020/3/12	1610612762	1610612758	2019	134	125
2020/3/12	1610612757	1610612764	2019	127	118
2020/3/11	1610612753	1610612750	2019	118	110
2020/3/11	1610612737	1610612746	2019	112	106

Table 4: Game Scores by Date

Graph Construction. A complete graph was constructed, with edge weights reflecting the point differences in games between teams.

TEAM_ID	NICKNAME
1610612737	Hawks
1610612738	Celtics
1610612740	Pelicans
1610612741	Bulls
1610612742	Mavericks
1610612743	Nuggets
1610612745	Rockets
1610612746	Clippers
1610612747	Lakers

Table 5: NBA Teams and Their Nicknames

Algorithm Application. Using the least squares method, the global ranking vector $\mathbf{r}$ was computed, resulting in a season-long performance rating of the teams. The Appendix section B contains the Python code that describes the computational stages involved in this approach. The graph Laplacian is also used in the mathematical formulation of the NBA scoring issue, and the same least squares goal is optimized to determine the ranking vector $\mathbf{r}$.

Results and Conclusion. The Hodge Rank algorithm's analysis, presented in Table 6 and illustrated in Figure 10, provides a succinct yet powerful depiction of NBA team standings over the season. The rankings reveal not just the outcomes of matches, but a deeper insight into the teams' performance levels.

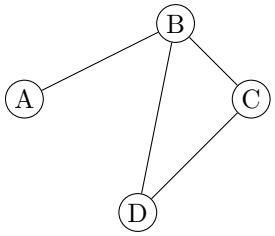
Essentially, the algorithm reduces the intricacy of the competitive dynamics of the season to a solitary, logical ranking system. It highlights a more nuanced assessment of team strength by taking into account the margins of success and defeat, providing a viewpoint that standard win-loss tallies cannot.

In summary, the Hodge Rank algorithm provides a fresh perspective on sports analytics and demonstrates its value in supporting professional basketball players' strategic decision-making.

Rank	Team	HodgeRank	Overall Rank	Overall Record
1	Bucks	27.832563	1	56-17
2	Celtics	20.189534	5	48-24
3	Lakers	19.658284	3	52-19
4	Raptors	16.670283	2	53-19
5	Clippers	16.107803	4	49-23
6	Mavericks	11.251159	-	-
7	Heat	10.236409	-	-
8	Rockets	8.767659	8	44-28
9	76ers	5.663692	-	-
10	Nuggets	5.115338	6	46-27
11	Jazz	4.687130	10	44-28
-	Pacers	-	7	45-28
-	Thunder	-	9	44-28

Table 6: Comparison of NBA Teams HodgeRank and Overall Ranking

Appendix A. Figures.



$$V = \{A, B, C, D\}\,,$$
$$E = \{\{A, B\}, \{B, C\}, \{C, D\}, \{D, B\}\}\,.$$
$$T = \{\{B, C, D\}\}$$

Fig. 1: Example of a simple graph with four vertices, showing the set of vertices V, edges E, and 3-cliques T.

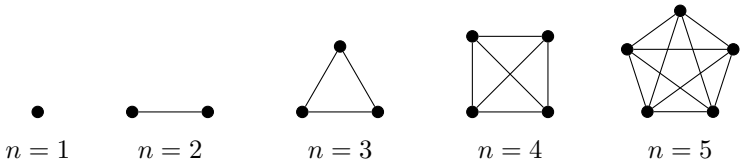


Fig. 2: Complete graphs up to $n = 5$ vertices.

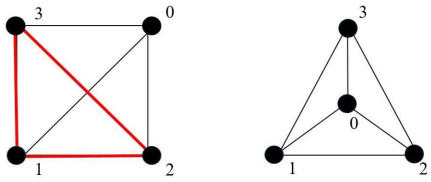


Fig. 3: Planar and non-planar representation of the same graph. The additional face is highlighted in red.

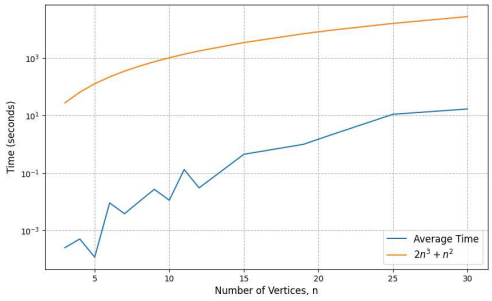


Fig. 4: Plot of the computation time required to compute graph cohomology groups for an adjacency matrix of dimension n.

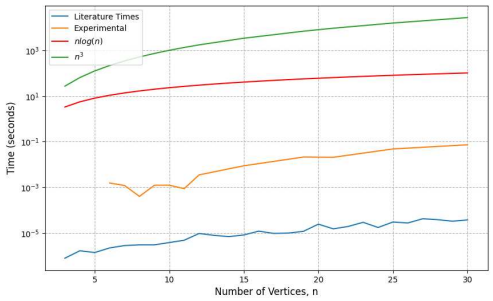


Fig. 5: Plot of the computation time in computing faces for literature code and experimental code developed in this report.

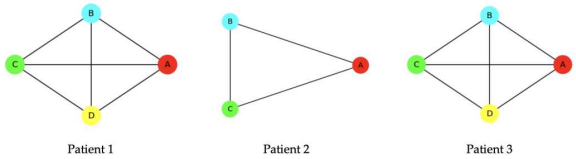


Fig. 6: Simple Graph for Example

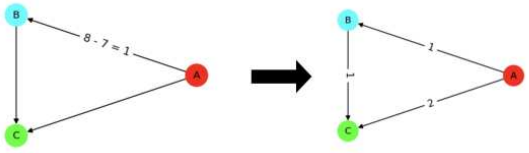


Fig. 7: Patient 2's graph with Pairwise Comparisons

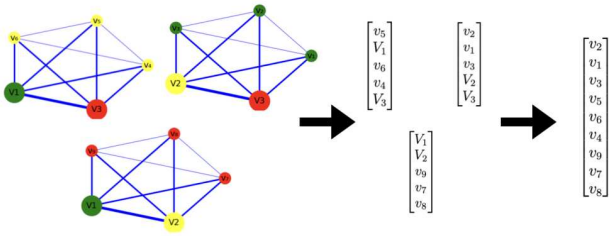


Fig. 8: Patient 2's graph with Pairwise Comparisons

$$V = [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9] \Longrightarrow \begin{aligned} V_1 &= \{v_1, v_2, v_3\} \\ V_2 &= \{v_4, v_5, v_6\} \\ V_3 &= \{v_7, v_8, v_9\} \end{aligned}$$

Fig. 9: Example: Split into 3 Subgroups by Naive Ranking

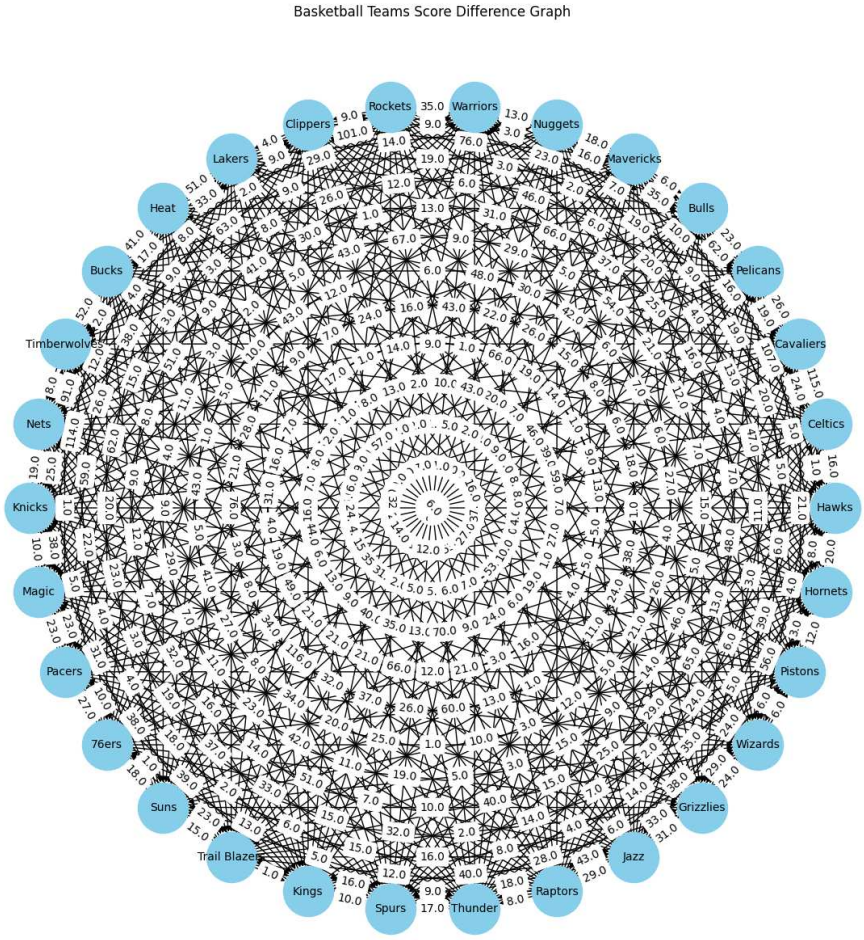


Fig. 10: NBA Teams Graph

Appendix B. Code Snippets. Hodge Rank Tools Python Code

```
import numpy as np
import pandas as pd
import math

# returns a list of all edges in the form of tuples
#ex: get_edges(['a', 'b', 'c']) => [('a', 'b'), ('a', 'c'), ('b','c')]
def get_edges(nodes):
    edges = []
    for i in range(len(nodes)):
        for j in range(i + 1, len(nodes)):
            edges.append((nodes[i],nodes[j]))
    return edges

# Converts a voter df into a list of dictionaries, where each dictionary
     represents the
# opinions of one voter
def df_to_dict_list(df):
    data = []
    nodes = df.columns
        #row_nodes = [node_index for node_index in range(len(nodes)) if
    not np.isnan(row[node_index])]
    for index, row in df.iterrows():
    # get group of nodes that this voter has scored
        row_nodes = [node for node in nodes if not np.isnan(row[node])]
        voter_dict = dict()
        for node in row_nodes:
            voter_dict[node] = row[node]
        data.append(voter_dict)
    return data

# gets all nodes from voter data structures as a list of dictionaries
def get_nodes(data):
    nodes = set()
    for voter in data:
        nodes.update(voter.keys())
    nodes = list(nodes)
    nodes.sort()
    return nodes

# returns:
# - f is a vector representing the edge flows between the nodes. It is
    indexed by
# the edges from the get_edges function
#  - W is a diagonal matrix containing the weights of each edge
# where df is a pandas dataframe such that the columns represent the
    nodes to be ranked and each
# row gives a voter's ratings
def get_f_W(data, nodes):

    edges = get_edges(nodes)
    f = np.zeros((len(edges)))
    W = np.zeros((len(edges), len(edges)))
    # iterate through each voter's ratings
    for voter in data:

        for edge in get_edges(list(voter.keys())):
            # for each edge in this voter's graph, add pairwise
    difference to f and add 1 to W
            f[edges.index(edge)] += voter[edge[1]] - voter[edge[0]]
            W[edges.index(edge), edges.index(edge)] += 1
    # weighting f by W
```

```
    for i in range(len(edges)):
        if W[i, i] != 0:
            f[i] = f[i]*1/W[i,i]
        else:
            f[i] = 0
    return (f, W)

#TODO: revise
# returns the error of this ranking according to some ranking r, which
    is a list containing
# the overall (numerical) rating of each node, indexed by the nodes in
    nodes
def get_error(f, W, r, nodes):
    edges = get_edges(nodes)
    sum = 0
    for i in range(len(edges)):
        to_add = (f[i] + (r[nodes.index(edges[i][0])] - r[nodes.index(
    edges[i][1])]))**2
        sum += to_add
    return sum

# returns the negative divergence matrix, structured as a numpy array,
    where df is a
# pandas dataframe such that the columns represent the nodes to be
    ranked and each
# row gives a voter's ratings
def get_neg_divergence(nodes):
    edges = get_edges(nodes)
    neg_divergence = np.zeros((len(edges), len(nodes)))
    for i, edge in enumerate(edges):
        for j, node in enumerate(nodes):
            if edge[0] == node:
                neg_divergence[i,j] = -1
            elif edge[1] == node:
                neg_divergence[i,j] = 1
    return neg_divergence

# ranks the nodes in the df according to HodgeRank, where df is a
# pandas dataframe such that the columns represent the nodes to be
    ranked and each
# row gives a voter's ratings
# returns:
# - pandas data frame representing the overall rating of each node, with
     two columns, node and
# r, which is the numerical score
# - error of this ranking
def rank(data):
    nodes = get_nodes(data)
    # get edges, negative divergence, f, and W
    edges = get_edges(nodes)
    neg_divergence = get_neg_divergence(nodes)
    (f, W) = get_f_W(data, nodes)
    # solve for r
    right_side = np.matmul(np.transpose(neg_divergence), np.matmul(W, f)
    )
    left_side = np.matmul(np.matmul(np.transpose(neg_divergence), W),
    neg_divergence)
    r = np.matmul(np.linalg.pinv(left_side), right_side)
    # put r into df and sort by score
    rank_df = pd.DataFrame({'node': nodes, 'r': r})
    rank_df = rank_df.sort_values(by =['r'],  ascending = False)
    rank_df = rank_df.reset_index(drop = True)
```

```
    #rank_df.to_csv('data/hodge_ranking.csv') # uncomment this line if
    you want to save the ranking
    return(rank_df, get_error(f, W, r, nodes))

# Ranks the nodes in the df by calculating the average score given to
    them. The input and output
# are structured the same as rank()
def naive_rank_0(df):
    nodes = list(df.columns)
    if len(list(set(nodes))) != len(nodes):
        raise Exception("All columns must have different names")
    naive_r = [0]*len(nodes)
    for i, node in enumerate(nodes):
        naive_r[i] = df[node].mean()
    naive_rank_df = pd.DataFrame({'node': nodes, 'r': naive_r})
    naive_rank_df = naive_rank_df.sort_values(by =['r'],  ascending =
    False)
    naive_rank_df = naive_rank_df.reset_index(drop = True)
    return (naive_rank_df, get_error(get_f_W(df)[0], get_f_W(df)[1],
    naive_r, nodes))

# Ranks the nodes in the df by calculating the mean pairwise difference
    of each node
# The input and output are structured the same as rank()
# TODO: revise this <3
def naive_rank(df):
    nodes = list(df.columns)
    if len(list(set(nodes))) != len(nodes):
        raise Exception("All columns must have different names")
    naive_r = [0]*len(nodes)
    node_weights = [0]*len(nodes)
    for index, row in df.iterrows():
        row_nodes = [node for node in nodes if not np.isnan(row[node])]
        for edge in get_edges(row_nodes):
            naive_r[nodes.index(edge[0])] += row[edge[0]] - row[edge[1]]
            naive_r[nodes.index(edge[1])] += row[edge[1]] - row[edge[0]]
            node_weights[nodes.index(edge[0])] += 1
            node_weights[nodes.index(edge[1])] += 1
    for i in range(len(naive_r)):
        if node_weights[i] != 0:
            naive_r[i] = naive_r[i]/node_weights[i]
        else:
            naive_r[i] = - 100000
    #naive_r = [naive_r[i]/node_weights[i] for i in range(len(naive_r))]
    naive_rank_df = pd.DataFrame({'node': nodes,'r': naive_r})
    naive_rank_df = naive_rank_df.sort_values(by =['r'],  ascending =
    False)
    naive_rank_df = naive_rank_df.reset_index(drop = True)
    return (naive_rank_df, get_error(get_f_W(df)[0], get_f_W(df)[1],
    naive_r, nodes))

# where k is the number of groups, returns a list of lengths of how big
    each
# group of nodes should be
# ex: get_group_lengths({df with 10 nodes/ cols}, 3) => [4, 3, 3]
def get_group_lengths(df, k):
    num_nodes = len(list(df.columns))
    group_size = math.floor(num_nodes/k)
    group_lengths = [group_size]*k
    for n in range(num_nodes - group_size * len(group_lengths)):
        group_lengths[n % len(group_lengths)] += 1
    return(group_lengths)
```

```
# where k is the number of groups, returns a list of groupings of the
    nodes in df
# ex: group_similar_scoring({df with cols 'a', 'b', 'c'}, 2) => [['a', '
    b'], ['c']]
def group_similar_scoring(df, k):
    naive_r = naive_rank(df)[0]
    # sort by score
    scores_df = naive_r[['r', 'node']]
    scores_df = scores_df.sort_values(by =['r'],  ascending = False)
    scores_df = scores_df.reset_index(drop = True)
    ranked_teams = list(scores_df["node"])
    # fill list with groups of teams, sorted by score
    groupings = []
    for group_length in get_group_lengths(df, k):
        groupings.append(ranked_teams[0:group_length])
        ranked_teams = ranked_teams[group_length:]
    return(groupings)

# Ranks the nodes in df by creating rankings on k different groups (
    organized by naive_rank())
# and stacking the final scores on top of eachother
def simple_group_rank(df, k):
    groupings = group_similar_scoring(df, k)
    r_groups = pd.DataFrame(columns = ['node', 'r'])
    error_groups = 0
    # create ranking for each grouping
    for grouping in groupings:
        small_game_df = df[grouping]
        small_game_df = small_game_df.dropna(how='all')
        (group_rank, group_error) = rank(small_game_df)
        r_groups = pd.concat([r_groups,group_rank])
        error_groups += group_error
    r_groups = r_groups.reset_index(drop = True)
    return(r_groups, error_groups)

# a specialized method that returns the sum across the first axis of a
    list of lists,
# treating nan's as 0 unless the entire row is nans
# ex: nansum([[5, np.nan, 1],       =>    [15, np.nan, 2]
#             [5, np.nan, np.nan],
#             [5, np.nan, 1]])
def nansum(to_sum):
    sum = [np.nan]*len(to_sum[0])
    for j in range(len(sum)):
        for i in range(len(to_sum)):
            if not np.isnan(to_sum[i][j]):
                sum[j] = np.nansum([to_sum[i][j], sum[j]])
    return sum


def group_rank(df, k):
    groupings = group_similar_scoring(df, k)
    r_groups = pd.DataFrame(columns = ['node', 'r'])
    fake_teams = set()
    error = 0
    # create ranking for each grouping
    for grouping in groupings:
        new_grouping = grouping.copy()
        small_game_df = df.copy()
        for i, other_grouping in enumerate(groupings):
            if other_grouping == grouping:
```

```
                continue
            fake_team_col = [[np.nan]*len(small_game_df.index)]
            fake_team_name = 'OTHER_TEAM' + str(i)
            fake_teams.add(fake_team_name)
            for group in other_grouping:
                fake_team_col.append( list(small_game_df[group]))
                small_game_df = small_game_df.drop(columns = [group])
            fake_team_col = nansum(fake_team_col)
            small_game_df[fake_team_name] = fake_team_col
        (group_rank, group_error) = rank(small_game_df)
        r_groups = pd.concat([r_groups,group_rank])
        error += group_error
    r_groups = r_groups[~r_groups['node'].isin(fake_teams)]
    r_groups = r_groups.reset_index(drop = True)
    return(r_groups, error)
```

COVID-19 Symptom Ranking Python Code

```
# COVID-19 Symptom Ranking Python Code
import csv
import pandas as pd
import numpy as np
import random
from hodgerank_tools import *

k_sev=1376
k_non_sev=4324
sev_p=[0.884,0.711,0.603,0.442,0.376,0.357,0.26,0.26,0.161,0.113,0.078,
       0.059,0.057,0.028]
non_sev_p
    =[0.814,0.657,0.442,0.057,0.28,0.128,0.131,0.109,0.121,0.135,0.097,
           0.057,0.058,0.051]
sev_n=[round(k_sev*i) for i in sev_p]
non_sev_n=[round(k_non_sev*i) for i in non_sev_p]
naive_r = [(sev_n[i]+non_sev_n[i])/5700 for i in range(len(non_sev_p))]
print(naive_r)
data = [sev_n, non_sev_n]
symptoms = ["Fever", "Cough", "Fatigue", "Dyspnea", "Sputum", "Shortness
     of Breath", "Myalgia", "Chill", "Dizziness", "Headache", "Sore
    Throat", "Nausea", "Diarhea", "Congestion"]
covid_df = pd.DataFrame(data, columns = symptoms)

print(covid_df)

# multiply all values by scalar, with sd, around normal dist
def norm(center, sd):
    rating = np.abs(np.random.normal(center, sd))
    if rating > 10: rating = 10
    return rating

    big_sev_list = []
big_non_sev_list = []

for symptom in symptoms:
    ones = [norm(8, 2) for i in range(covid_df[symptom][0])]
    zeroes = [0 for i in range(k_sev - covid_df[symptom][0])]
    new_col = ones + zeroes
    random.shuffle(new_col)
    big_sev_list.append(new_col)

for symptom in symptoms:
    ones = [norm(3,1) for i in range(covid_df[symptom][1])]
    zeroes = [0 for i in range(k_non_sev - covid_df[symptom][1])]
```

```
    new_col = ones + zeroes
    random.shuffle(new_col)
    big_non_sev_list.append(new_col)

big_sev_array = np.array(big_sev_list)
big_non_sev_array = np.array(big_non_sev_list)

big_sev_array = np.transpose(big_sev_array)
big_non_sev_array = np.transpose(big_non_sev_array)

print(big_sev_array.shape)
print(big_non_sev_array.shape)
big_data = np.concatenate((big_sev_array, big_non_sev_array), axis=0)
print(big_data.shape)
covid_df = pd.DataFrame(big_data, columns = symptoms)
print(covid_df.head)

#save final version
#covid_df.to_csv('data/covid/covid_df.csv')

covid_df = pd.read_csv("data/covid/covid_df.csv")[symptoms]
print(covid_df.head())

(covid_rank, covid_rank_error) = rank(covid_df)
print(covid_rank)
print("error: ", covid_rank_error)

#save final version
covid_rank.to_csv('data/covid/covid_rank.csv')
```

NBA Team Scoring Python coding

```
# NBA Team Scoring Python coding
import csv
import pandas as pd
import numpy as np
import math
from scipy import stats
from hodgerank_tools import *

big_game_df = pd.read_csv("data/nba/games.csv")
team_df = pd.read_csv("data/nba/teams.csv")
og_game_df = big_game_df[['GAME_ID','SEASON', 'HOME_TEAM_ID', '
    VISITOR_TEAM_ID', 'PTS_home', 'PTS_away']].copy()
og_game_df = og_game_df[og_game_df['SEASON'] == 2021]
og_game_df.dropna( inplace = True)

teams = set(og_game_df['HOME_TEAM_ID'].tolist())
teams.update(set(og_game_df['VISITOR_TEAM_ID'].tolist()))
teams = list(teams)

team_names = []
for team in teams:
    team_names.append(str(team_df[team_df['TEAM_ID'] == team].iloc[0]['
    NICKNAME']))

data = np.empty((len(teams), len(teams)))*np.nan
for i, game in og_game_df.iterrows():
    visiting_i = teams.index(game['VISITOR_TEAM_ID'])
    home_i = teams.index(game['HOME_TEAM_ID'])
    data[visiting_i, home_i] = np.nansum([data[visiting_i][home_i], int(
    game['PTS_home']) - int(game['PTS_away'])])
    data[home_i, visiting_i] = np.nansum([data[home_i][visiting_i],  int
```

```
    (game['PTS_away']) - int(game['PTS_home'])])
game_df = pd.DataFrame(data = data, columns = team_names)
print( og_game_df)
print(game_df.head())

(naive_r0, naive_r0_error) = naive_rank_0(game_df)
print(naive_r0)
print("error: ", naive_r0_error)

(naive_r, naive_r_error) = naive_rank(game_df)
print(naive_r)
print("error: ", naive_r_error)

(r_regular, error_regular) = rank(game_df)
print(r_regular)
print("error: ", error_regular)

print(stats.kendalltau(naive_r['node'], r_regular['node']))
print(stats.kendalltau(list(naive_r['node'])[:10], list(r_regular['node'
    ])[:10]))

# Runs on 30/3 = 10 nodes at a time
k = 3
(r_groups_1, error_groups_1) = simple_group_rank(game_df, k)
print(r_groups_1)
print("error: ", error_groups_1)

print(stats.kendalltau(r_groups_1['node'], r_regular['node']))
print(stats.kendalltau(list(r_groups_1['node'])[:10], list(r_regular['
    node'])[:10]))

# Improved group method
# Runs on 30/3 + 2 = 12 nodes at a time
(r_groups_2, error_groups_2) = group_rank(game_df, k)
print(r_groups_2)
print("error: ", error_groups_2)

# Kendall's Tau
print(stats.kendalltau(r_groups_2['node'], r_regular['node']))

# Kendall's Tau of first n teams
n = 10
print(stats.kendalltau(list(r_groups_2['node'])[:n], list(r_regular['
    node'])[:n]))
```

REFERENCES

[1] A. Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, UK, 1996.

[2] Lek-Heng Lim. Hodge Laplacians on Graphs. *SIAM Review*, 62(3):685–715, January 2020.

[3] Lei. Fu. *Etale cohomology theory [edited by] Lei Fu.* World Scientific, 2011.

[4] Jonathan L. Gross and Thomas W. Tucker. *Topics in Topological Graph Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2009.

[5] John M. Lee. *Introduction to Smooth Manifolds*, volume 218 of *Graduate Texts in Mathematics*. Springer, 2003.

[6] Russell Merris. *Graph theory / Russell Merris.* John Wiley, 2001.

[7] M. Pellikka, S. Suuriniemi, L. Kettunen, and C. Geuzaine. Homology and cohomology computation in finite element modeling. *SIAM Journal on Scientific Computing*, 35(5):B1195–B1214, 2013.

[8] Solomon Lefschetz. *Applications of algebraic topology : graphs and networks : the Picard-Lefschetz theory and Feynman integrals / S. Lefschetz.* Applied mathematical sciences ; v.16. Springer-Verlag, New York, 1975.

[9] Levin and Taylor. Computing planarity in computable planar graphs. *Graphs and Combinatorics*, 32:2525–2539, 2016.

[10] CP-Algorithms. Number of faces of a planar graph. https://cp-algorithms.com/geometry/planar.html#number-of-faces-of-a-planar-graph. Accessed: 2024-04-18.

[11] Steve Heller. *Optimizing C++*. Prentice Hall PTR, 2000.

[12] Douglas Arnold, Richard Falk, and Ragnar Winther. Finite element exterior calculus: from Hodge theory to numerical stability. *Bulletin of the American Mathematical Society*, 47(2):281–354, January 2010.

[13] Douglas N. Arnold. *Finite element exterior calculus*. CBMS-NSF regional conference series in applied mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 2018.

[14] Qiang Du, Max Gunzburger, R. B. Lehoucq, and Kun Zhou. A Nonloca Vector Calculus, Nonlocal Volume-Constrained Problems, and Nonlocal Balance Laws. *Mathematical Models and Methods in Applied Sciences*, 23(03):493–540, March 2013.

[15] D. Austin. Who's number 1? hodge theory will tell us. AMS Feature Column, December 2012.

[16] Anil N. Hirani, Kaushik Kalyanaraman, and Seth Watts. Least squares ranking on graphs, 2011.

[17] Xiaoye Jiang, Lek-Heng Lim, Yuan Yao, and Yinyu Ye. Statistical ranking and combinatorial Hodge theory. *Mathematical Programming*, 127(1):203–244, March 2011.

[18] H. Lee et al. Harmonic holes as the submodules of brain network and network dissimilarity. In R. Marfil et al., editors, *Computational Topology in Image Context*, volume 11382 of *Lecture Notes in Computer Science*, pages 110–122. Springer, Cham, 2019.

[19] Mengyi Zhang, Alban Goupil, Anas Hanaf, and Tian Wang. Distributed Harmonic Form Computation. *IEEE Signal Processing Letters*, 25(8):1241–1245, August 2018.

[20] Vin De Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson. Persistent Cohomology and Circular Coordinates. *Discrete & Computational Geometry*, 45(4):737–759, June 2011.

[21] Yiying Tong, Santiago Lombeyda, Anil N. Hirani, and Mathieu Desbrun. Discrete multiscale vector field decomposition. *ACM Transactions on Graphics*, 22(3):445–452, July 2003.

[22] Charles Colley, Junyuan Lin, Xiaozhe Hu, and Shuchin Aeron. Algebraic Multigrid for Least Squares Problems on Graphs with Applications to HodgeRank. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 627–636, Orlando / Buena Vista, FL, USA, May 2017. IEEE.