Dynamic Nonlinear Gaussian Model for Inferring a Graph Structure on Time Series

A Thesis Presented

by

Abhinuv Uppal

To the Keck Science Department

of

Claremont McKenna, Scripps, and Pitzer Colleges

In Partial Fulfillment of

The Degree of Bachelor of Arts

Senior Thesis in Physics

April 25th, 2022

Abstract

In many applications of graph analytics, the optimal graph construction is not always straightforward. I propose a novel algorithm to dynamically infer a graph structure on multiple time series by first imposing a state evolution equation on the graph and deriving the necessary equations to convert it into a maximum likelihood optimization problem. The state evolution equation guarantees that edge weights contain predictive power by construction. After running experiments on simulated data, it appears the required optimization is likely non-convex and does not generally produce results significantly better than randomly tweaking parameters, so it is not feasible to use in its current state. However, I discuss potential improvements and suggestions as to how the algorithm could become feasible in the future.

1 Introduction

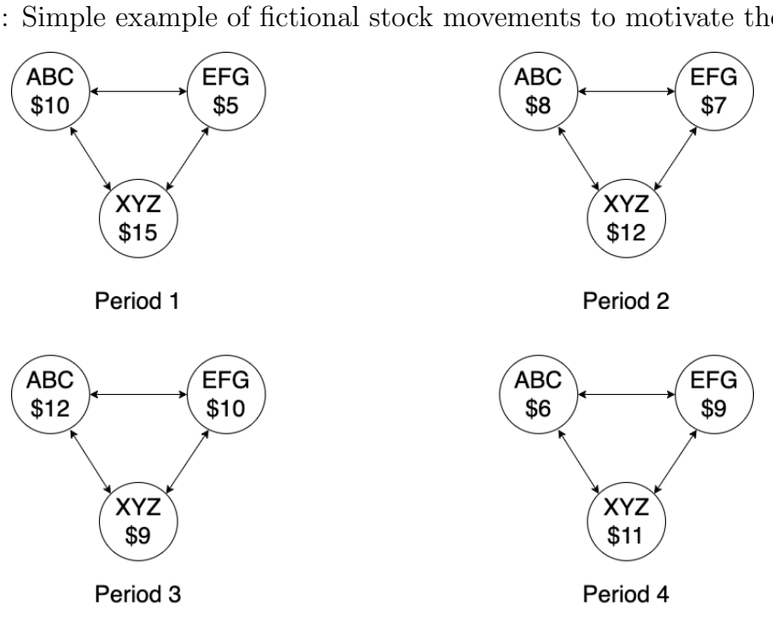
Graph theory is a flexible mathematical language for modeling the relationship between different entities. While it is a vague description, it draws its power from this – providing a framework for analyzing problems ranging all the way from social network analysis [25] to modelling natural language [27].

Once one has a graph structure, one can apply a plethora of algorithms to understand the statistics of the graph – which entities are the big players, how strongly connected are each of the entities, how many hops would it take to get from one end of the graph to the other, and many more. Said statistics can be fed into graph machine learning pipelines to create incredibly powerful predictive and generative models.

However, as with all data science, data analysis on graphs largely adheres to the tried and tested principle of "garbage in, garbage out." Choosing exactly how to structure your graph involves a plethora of design choices: what are the nodes intended to represent? What information do the edges encode? Should the edges be weighted? Should the graph be sparse or dense? State-of-the-art graph machine learning methods cannot fix a poorly constructed graph.

As a motivating example, consider the fictional stock movements presented in Figure 1. Companies ABC, EFG, and XYZ all have stock values that change over time. Presumably, they have some sort of influence on each other. Each node in the graph represents a time series of that particular company's stock price. The question that motivates this study is to determine whether it is possible to learn weights on the edges connecting the nodes using said time series data. These weights should have some tangible interpretation on how the graph evolves in time – how it moves from Period 1 to Period 2, from Period 2 to Period 3, and so forth. That is, the edge weights explicitly contain predictive power. As such, a construction of this form would be incredibly valuable for anyone working with potentially connected time series data.

Figure 1: Simple example of fictional stock movements to motivate the study.



As will be discussed in section 2, current methods in this field either rely either on graph neural network (GNN) methods or static methods where statistics are computed over a sliding window. This paper attempts to provide a new lens into this field with its dynamic model, applicable to any field in which multiple time series are involved. This alternative construction provides a new design choice for imposing graph structures on time series, where the edges explicitly encode information on how the nodes evolve at each point in time.

In Section 2, I provide a high-level overview of graphs, as well as some mathematical definitions of graphs and some node features. I then discuss some of the applications of graph theory before pivoting into a discussion of the problem at hand in this project: how to optimally infer edge weights for a graph whose nodes represent different time series, with the goal of modeling the relationships between the time series by considering how they evolve together in time. Within this discussion, I discuss a possible application of this model in the analysis of financial markets.

After the background, I present the novel methodology – a nonlinear Gaussian noise model that learns through stochastic gradient descent a graph and a Gaussian covariance matrix *dynamically* based on how different series move together in time. Following the methodology, I present the results of the algorithm and conclusions to be drawn from it.

Past the conclusions section is the Appendix, where all of the mathematical work is done. In the Appendix, I assume some familiarity with matrix calculus. I invite the reader to consult the Matrix Cookbook [22] as a reference.

The interested reader may find the code used for this paper available on my GitHub profile[1].

2 Background and Literature Review

Graph theory as a field, while widely applicable, is seldom discussed outside of its direct applications. As such, it is customary in many papers to give an overview of the field in the interest of being self-contained. This section first takes a pedagogical approach to familiarizing the reader with graphs, and then discusses some of the past work in the field of inferring graph structures from data.

Readers familiar with graph theory may skip to Section 2.3. For those familiar with graph machine learning and its applications, the content most relevant to this study begins at section 2.3.3.

2.1 A High-Level Overview of Graphs

A modern understanding of the world, whether quantitative or qualitative, seldom permits the ability to understand phenomena in isolation. One of the most prominent ways this is seen is in the humanities, where scholars will attempt to understand issues as the complex combination of multiple causes. A popular example is Mark Juergensmeyer's *Terror in the Mind of God* [8], where he attempts to understand global violence and terrorism through the intersection of politics and religion – two somewhat disparate topics with a sizable overlap. Looking at each factor alone, while informative, cannot provide the full picture. Rather, it is the *relationship between* factors that can often provide informative signals towards the truth.

So, how does this method of inquiry manifest itself in quantitative terms? The idea is rooted in an application of *graph theory*, one of the most flexible fields of mathematics. We represent the objects we wish to understand as agents, or nodes, in a network (henceforth referred to as a *graph*). These nodes are connected to each other via edges that categorize and/or quantify the relationship between them. For example, in a graph describing things we might see in a suburban neighborhood, we might have a node for "person" and "car". How might we write the relationship between them? We might write,

$$\text{person} \xrightarrow{\text{DRIVE}} \text{car},$$

where "DRIVE" is a type of edge connecting the "person" and "car" nodes, denoting that "DRIVE" is an action that "person" can take with "car". Similarly, we might have another edge

[1] https://github.com/AbhiUppal/seniorthesis

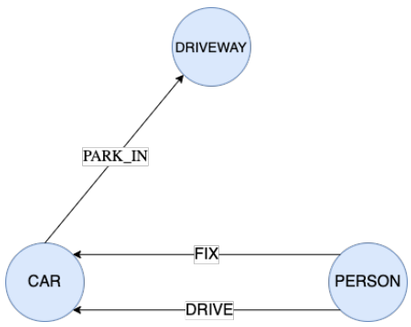
$$\text{person} \xrightarrow{\text{FIX}} \text{car},$$

denoting that "person" can also fix the object represented by "car". As a final example, we might have

$$\text{car} \xrightarrow{\text{PARK\_IN}} \text{driveway},$$

denoting that "PARK_IN" is also an action that the object represented by "car" can take with the object represented by "driveway". Following this process to construct more nodes and edges, we may end up with a graph that describes pretty well how the components of a suburban neighborhood might act with each other, giving us a clear picture of how they might operate on a day-to-day basis.

Figure 2: Simple Heterograph Toy Example



In this toy example, we easily understand what each of these nodes and edges represent. It may seem pointless to go through this exercise, but the importance lies in how we abstract from this model. What if we didn't aptly name the nodes, but created a set of nodes $V = \{1, 2, 3, \ldots, n\}$, but we kept the same relationships? So, $1 \xrightarrow{\text{DRIVE}} 2$, $1 \xrightarrow{\text{FIX}} 2$, and $2 \xrightarrow{\text{PARK\_IN}} 3$. From a mathematical perspective, assigning the word "car" is just as arbitrary as assigning the number 1 to a node. In a similar sense, the edge names "DRIVE", "FIX", "PARK_IN" are also completely arbitrary. They make sense to us, given our learned experiences of the world, but I easily could have written them in Cyrillic, Hebrew, or a niche dialect of Hindi. The point is, the exact label has no real meaning from a mathematical perspective – they are just labels that keep everything organized. Regardless of what the labels are, they stay consistent – and *that's* what encodes the information.

The big picture, now, is that we have a bunch of objects, which we refer to as **nodes** or **vertices**, all interacting with each other. These interactions are encoded by **edges**. The collection of objects (nodes) and interactions (edges) form a **system** (graph).

Graph theory provides a mathematical framework for working with structures like these. A graph with multiple edge (and/or node) types, as we saw in the above example, is called a **heterogeneous graph**, or **heterograph**. Since the relationships all run in one direction, we call it a **directed graph**, or **digraph** (that is, person can drive car but car cannot drive person). If a graph is not directed, we creatively call it an **undirected graph**.

Edges need not always encode a qualitative relationship. If we let the nodes of a graph be random variables $V = \{X_1, X_2, ..., X_N\}$, then we can define edges between them such as:

$$e_{ij} = \begin{cases} 1 & \text{if } X_i \text{ and } X_j \text{ are drawn from the same distribution} \\ 0 & \text{if not} \end{cases}$$

This creates a graph with binary edges – 0 if the nodes are not connected, and 1 if they are connected. One could alternatively define *weights* on these edges. We could, say, define

$$e_{ij} = \frac{\text{cov}(x, y)}{\text{var}(x)\text{var(y)}},$$

which would produce values in the interval $e_{ij} \in [-1, 1]$. These edges are said to be **weighted**, in which case we have a **weighted graph**. At this point, it is worth noting that there is no one correct way to construct a graph between a set of nodes. One construction may be useful for some applications, and another may be useful for others. This is a design choice that has to be made by the person using the graph to solve a specific problem.

2.2 Concrete Definitions

This section will define many of the terms used throughout the paper

Definition 2.1 (Graph)**.** A graph $\mathcal{G} = (V, E)$ is an ordered pair consisting of a set of nodes (vertices) V and a set of potentially ordered pairs of nodes (called edges) E. An ordered pair (i, j) represents an edge from node i to node j. Order does not matter in an undirected graph.

It is often useful (and it will be in this paper) to construct a representation of a graph as an $n \times n$ matrix, where n is the number of nodes:

Definition 2.2 (Adjacency Matrix)**.** An adjacency matrix is a representation of a graph as an $n \times n$ matrix, where each entry A_{ij} denotes the existence (or lack thereof) an edge between from node i to node j.

Definition 2.3 (Neighborhood)**.** The k-hop neighborhood of a node u of a graph $\mathcal{G} = (V, E)$ is the set of all nodes within k hops of the node u. A 1-hop neighborhood of u are all of the nodes u is directly connected to. The 2-hop neighborhood contains the 1-hop neighborhood as well as all direct connections of each of those nodes, and so on. For a 1-hop neightborhood of a node u, I denote this as $\mathcal{N}(u)$. For a k-hop neighborhood, I denote this as $\mathcal{N}^k(u)$.

Note that the adjacency matrix can encode relationships for both directed and weighted graphs. If $\mathcal{G}$ is a directed graph, then A need not be symmetric (there can be an edge from node $i \to j$ but not from node $j \to i$). If $\mathcal{G}$ is a weighted graph, then each entry A_{ij} in the matrix will not be binary, but rather the edge weight between nodes i and j. For unweighted graphs, self connections (nodes connecting to themselves; elements on the diagonal of the adjacency matrix) are typically counted as 2 rather than 1.

Any given row i or column j describes the 1-hop neighborhood (direct neighbors) of node i (or j). That is, all nonzero entries in row i correspond to nodes within one hop of node i. If we want to examine a 2-hop neighborhood of a node i, we can similarly look at row i, but not in A – rather, we look in A^2. For a k-hop neighborhood, we can look at row i of A^k. A proof is provided in Chapter 2 of [5].

Naturally, once one has a graph, one may be interested in some of the descriptive features of the nodes in the graph (as well as the graph as a whole. I provide a quick overview of some of the common definitions of node features, but refer the reader to [17] for a more detailed explanation of graph feature generation.

Definition 2.4 (Node Degree)**.** The degree of the a node in a graph $\mathcal{G} = (V, E)$ is the sum of the outgoing edge weights:

$$d_u = \sum_{v \in \mathcal{N}(u)} A_{uv}.$$

For a directed graph, one must make the distinction between the in-degree and the out-degree of a node, since $A_{uv} \neq A_{vu}$ in general. Performing this sum for A_{uv} (edges going from node u to node v) corresponds to the out-degree, and the sum for A_{vu} corresponds to the in-degree (edges going from node v to node u).

Node degree is often used when considering the importance of a node in a graph – that is, more important nodes tend to be more densely connected. However, alternative measures exist to measure node importance – particularly centrality measures such as betweenness centrality, closeness centrality, and eigenvector centrality.

2.3 Why Graphs? Applications of Graphs

Graphs are a general language for describing and analyzing disparate entities with interactions and potentially nontrivial relationships. They are simple as a data structure, but have a wide variety of applications that stretches nearly as far as the imagination can stretch [5]. Part of the power in graph formalisms is that the data structure inherently models relationships *between* different points, as opposed to just the properties of individual points.

Modern methods in graph machine learning such as Graph Convolutional Networks (GCN) [11] and GraphSAGE [6] allow one to consider both the graph structure *and* features

of the individual points in machine learning pipelines – effectively increasing the potential predictive and explanatory power of machine learning models for any datasets that can be seen as a graph.

This same formalism can be used, for example, to represent molecules in biological research and machine learning, social networks, interactions between drugs and proteins, and relationships between internet pages. In some previous work in the literature, as well as some of my own work, graph structures are used on financial markets where nodes represent individual time series and edges represent some sort of quantititative relationship between the two time series.

2.3.1 Classical Algorithms and Statistics

One of the most famous examples is Zachary's Karate Club network [32], in which Wayne Zachary studied a graph where nodes represented members of a karate club, and binary edges represented whether or not two individuals socialized outside of the club. Zachary attempted to split the club into two factions and predict which nodes would fall into each given the graph structure. This is a classic example of a task known as **community detection**, which attempts to identify communities in graphs (densely connected clusters of nodes) and which nodes belong to said communities.

A classical algorithm that can be run on graphs is known as **PageRank** [20], which was initially developed by Google. As surprising as this may sound, this algorithm was designed to rank pages. Specifically, it assigns a score to each node based on how likely a random walker strolling through the graph is to end up at that node. Applying this algorithm with some slight modifications is how Google knows how to rank pages when showing the results of a search query. The first result is the node with the highest PageRank score, the second is the node with the second highest PageRank score, etc.

The beauty of this algorithm is twofold in that it can be computed fairly efficiently, and it can also be easily applied to graph datasets that represent completely different things (we see the power of using a common data structure now)! Given a vector of initial guesses for PageRank scores $\vec{r}_0$ and a stochastic transition matrix G that we can easily compute with the adjacency matrix, we can update our guess for the PageRank value as

$$\vec{r}_{n+1} = G\vec{r}_n,$$

and the PageRank score itself is simply the stationary distribution of the transition matrix. That is, the $\vec{r}$ such that

$$\vec{r} = G\vec{r}.$$

So, in order to calculate the PageRank scores of each of the nodes in a graph, all one has to do is trivially compute a transition matrix G and apply this to some initial guess $\vec{r}_0$ repeatedly until the solution converges – that is, $\|r_{n+1} - r_n\| < \varepsilon$ for some small $\varepsilon > 0$. This process is known as **power iteration**, which depends on repeated matrix multiplication – a relatively efficient process in modern computing. One may also recognize this as an eigenvalue problem, where $\vec{r}$ is the eigenvector that corresponds to the eigenvalue 1 of the matrix G.

The key point of this exposition into PageRank is to show that, with a simple algorithm that can be computed fairly efficiently, we now have a system of ranking the most important nodes in *any* graph, *regardless* of what it may represent! Of course, modifications have to be made for more complicated graph structures such as heterographs, but the basic idea remains. Oftentimes, one of the first things that should be done when analyzing a new graph is to first look at the distribution of node degrees, and then look at PageRank scores.

Other work, such as [2], applies graph theory to epidemics. Specifically, it uses an SIR model for epidemic spreads [9] on graphs, assuming that the graph's evolution in time accords with the SIR model. With this in hand, they are able to show that the model can "turn back the clock" and infer patient zero in an outbreak given a snapshot of the graph at an unknown point in time after the beginning of the epidemic.

2.3.2 Graph Machine Learning

So, classical algorithms are already powerful, but the real beauty of having graph representations is the realm of **graph machine learning**. These are incredibly powerful models that

utilize graph structures, and potentially features associated with nodes, to perform a classification or regression task. There are many tasks possible for machine learning algorithms, but the three most common tasks are

Node classification: Classify each node as a member of a pre-defined group. An example of this would be to determine whether or not each person in a social network owns a car.

Edge prediction: Given an incomplete graph (a subset of edges), predict what edges should be added to complete the graph. An example of this would be to determine which users should become friends in a social network based on their profile characteristics and who they are already connected to.

Graph classification: Assign a classification label to the graph as a whole. If each graph in your dataset represents a molecule used to treat a disease, the classification task could be used to determine whether or not that molecule is toxic for the human body.

There are additional complications that come with using graph data (e.g., how does one determine how to perform a train/test/validation split for a graph dataset?). For a review of graph machine learning methods, the interested reader should look at [31]. For an in-depth dive into graph neural networks, see [33].

2.3.3 Learning Graph Structures

To this point, we have seen some of the incredibly useful features of graphs, including but nowhere near limited to being able to describe relationships between entities, incorporate said relationships into machine learning pipelines, and providing a common data structure for complex relationships.

While these structures are evidently important, the question then becomes how we can obtain a graph structure in the first place? In some cases this may be trivial – if we want an unweighted, undirected graph of friends in a Facebook network, then we can simply let an edge between two nodes (people) be 1 if they are friends and 0 if not.

But what if we wanted to quantify the strength of their friendship in an interval $[0, 1]$, where 0 represents complete strangers (unconnected) and 1 represents best friends? Quantifying this relationship may become slightly more difficult and would largely depend on the available data. Maybe you could quantify edge strength based on messaging frequency. Perhaps by profile views, or maybe a combination of the two. The key takeaway here is that there is not necessarily one correct way to construct a graph. Even when a graph is constructed, it is often subject to some sort of measurement error that requires reconstruction methods based on generative models to correct [21].

We then turn to the key question: how can one define and learn optimal edge weights from the data? The anticlimactic answer is: it depends. The long answer is that the use case generally dictates what we mean by "optimal." If we are discussing measuring friendship strength in a social network, then modeling friendship strength with messages and interactions might be a great way to do this. But in the case of a network of moving objects such as particles, one may want to instead define edge weights to be stronger if the particles move together closer in time. For example, [10] describes a Neural Relational Inference model for interacting systems using graph neural networks, which is applied to data on particle trajectories as well as player trajectories in pick-and-roll basketball plays. From this, one can observe the strength in graph inference algorithms in action – once a solid algorithm is developed, it can be applied to a variety of seemingly disparate tasks.

In this paper, I devlop a framework that focuses particularly on use cases based on studying how nodes move together in time. I aim to construct graphs in which the nodes represent objects that have time series data associated with them, where the edges represent a similarity measure detailing how they evolve together in time.

This type of work, while relatively new, is not completely novel. Graph Signal Processing (GSP) is a broad field that deals with processing data associated with nodes on a graph. One subfield of this is graph learning, which deals with how to infer a graph structure from data when a graph structure may not be inherently obvious. An overview of this, as well as GSP as a whole, can be found in [19]. An example of graph learning is [16], which attempts to learn a weighted, directed graph encoding causality from unstructured time series data.

Alternative approaches exist, such as [28]. In it, Stepaniants et al. note that the problem of graph inference is ill-posed in that there are multiple viable graph structures that can be used to describe the behavior of the same dynamical system. To remedy this, they

describe a graph learning algorithm that works by considering how the behavior of the nodes change after a perturbation at one of the other nodes. Modeling how the nodes react to perturbations, the authors claim, helps to disambiguate and clarify what the true graph structure behind the data.

2.4 Financial Market Graphs

A possible application of these methods is learning a graph structure on financial markets (particularly a subset of the U.S. stock market).

The study of graph structures on stock markets is popular among many disciplines – publications of which stem from journals that cover machine learning, information geometry, econophysics, statistical physics, econometrics, and behavioral finance [23]. However, most of the interest in this subject stems from Mantegna's 1998 seminal paper [13] *Hierarchical Structure in Financial Markets*. In this paper, he introduces the idea of learning a meaningful topological structure on stock returns in financial markets, and chooses to represent said topological structure via a graph. Specifically, he chooses to represent them via a minimum spanning tree (MST):

Definition 2.5 (Tree)**.** A tree is a graph with no cycles in which there is exactly one path between any two points.

Definition 2.6 (Minimum Spanning Tree)**.** A minimum spanning tree is a tree-structured subgraph T of a graph $\mathcal{G} = (V, E)$,with edges selected such that their weights are as small as possible.

Mantegna begins with a selection of stock series he wishes to model the relationship between, represented as time series of prices P_i, where i represents a particular stock. To ensure stationarity, he transforms each series i to be the difference in logged prices between the current and previous period:

$$R_i(t) = \log P_i(t) - \log P_i(t-1).$$

He hopes to capture relationships in the graph by considering how these log returns move together over a sliding window. To achieve this, he uses the Pearson correlation, specialized for capturing linear correlations. This is defined by

$$\rho(i,j) = \frac{\text{cov}(i,j)}{\text{var}(i)\text{var}(j)},$$

where cov represents the covariance between series i and j, and var represents the variance of a series. This coefficient can range from -1 (two series are completely anti-correlated) to 1 (two series are perfectly correlated), with 0 representing no correlation between the two series. Since this is not a Euclidean metric ($-1 \leq \rho(i,j) \leq 1$), the correlation coefficient can be transformed into angular distances:

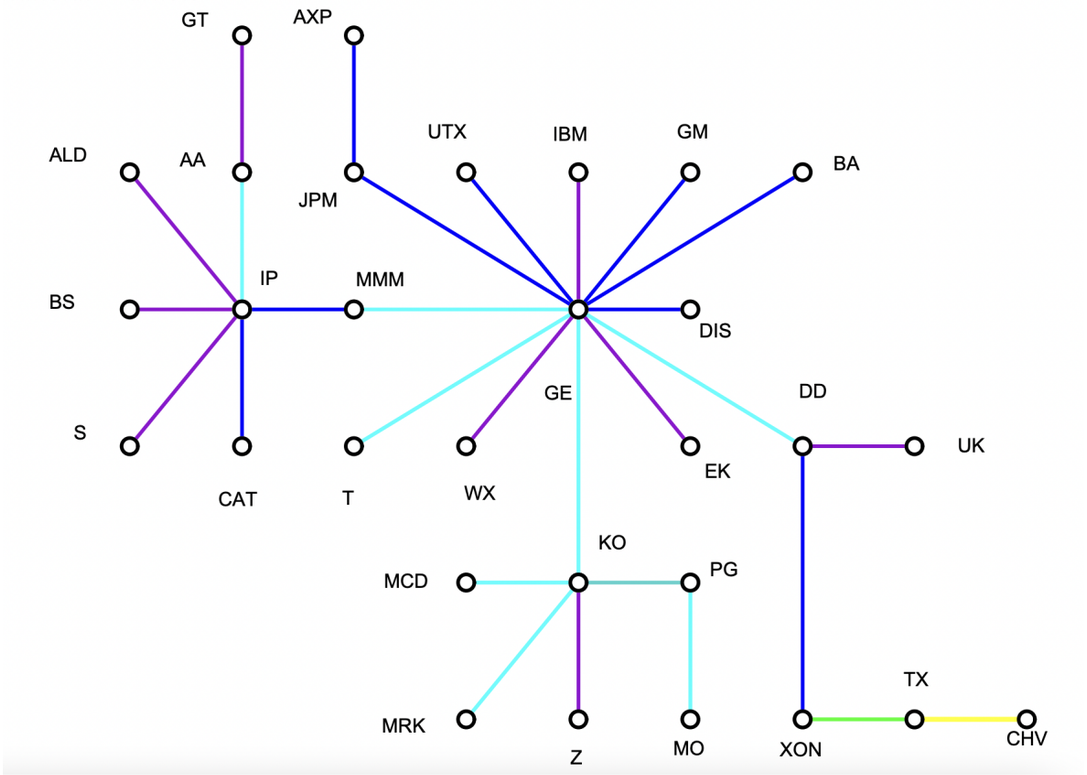
$$d(i,j) = \sqrt{2(1-\rho(i,j)}.$$

Once these distances are computed, we can compute an MST using one of many algorithms, such as Kruskal's Algorithm [12] [23]. See Figure 3 for an example of one of the MSTs constructed in Mantegna in his experiments. Once one has a market graph, there are many applications. For example, one may want to compute graph features and integrate them into a machine learning pipeline. However, there are other applciations beyond machine learning.

Mantegna motivates the usage of an MST to model financial markets by attempting to discover a hierarchical structure through which they implicitly operate. In such a structure, parts of the MST tend to be strongly connected, which tend to be stocks that belong to the same industry (financial services, electrical utility companies, automobile manufacturers, etc.). These stocks, which tend to be at the leaves of the trees, are typically affected by economic factors relevant to their particular industries. More central nodes, on the other hand, tend to be influenced by more factors, and are therefore usually more unpredictable and risky investments. However, none of these results are shown mathematically – they are derived from empirical data collected from 1989 to 1995.

In [18], Onnela et al. show that the optimal Markowitz (risk-minimizing) portfolio almost always lies on the leaves of the tree (nodes with only one incoming edge). In [24], Sandhu et

Figure 3: Example MST computed by Mantegna in his paper *Hierarchical Structures in Financial Markets*.



al. generalize the differential geometric idea of Ricci curvature to graphs and show that the average Ricci curvature of a market graph (with stocks as nodes) increases drastically during financial crises. They also briefly explore the relationship between minimum risk Markowitz portfolios and the curvature of the graph.

However, the methodology presented above does not come without issues. While both the algorithm and the metric are relatively quick to compute, they tend to be statistically weak. The MST is known to be unstable, which may be in part due to the fact that the Pearson correlation is brittle to outliers and may not be preferable to other metrics such as a distance derived from the Spearman correlation or distances designed for working on i.i.d. random processes [15][14]. These are some of a whole host of different potential issues with the methodology. I invite the reader to look at [23] for a comprehensive review of the problems and potential solutions.

Among the solutions mentioned in the above papers, one may opt to instead use a metric that better captures nonlinear relationships such as the Brownian distance correlation [30], or a directed graph based on Granger causality [3]. However, while they are certainly informative, nonlinear metrics have a drawback in that they take much longer to compute than their linear counterparts.

In the case of the algorithm, several alternatives have been suggested to remedy Mantegna's original algorithm. An example of such is a dynamic spanning tree (DST), designed to tackle the stability problem of the MSTs by capturing the fact that correlations vary over time within the algorithm itself [26]. There also exist maximum likelihood methods [4] that define clustering non-parametrically based on simple 1-factor models that do not need to specify the number of clusters *a priori*.

All to say, graphs and statistics related to graphs can have useful and interpretable implications for someone hoping to understand and/or predict the behavior of a financial market, or simply make risk-minimizing portfolio management decisions given recent market data. The novel approach to graph learning provided in this paper provides a potential basis for a new exposition into understanding financial markets through learning graphs where edges encode information about how the graph evolves as a whole.

2.5 The Goal and Novel Introductions

The goal of the project is to represent different time series as nodes in a graph, and using the data of the time series to learn weighted, undirected edges between these nodes. The key here is to not use some heuristic metric (e.g., using something like angular distance), but rather learn the edge weights through a process that explicitly considers how each node's value changes at each time step with respect to each other. So, instead of just computing the correlation over a window and calling that an edge weight, I use a noisy model to determine how each node evolves both by itself (noise) and in relation to other nodes (edges). This means that the inferred edges of in this type of graph encode predictive information on what values nodes will take in the future. As noted in Section 2.1, there is no one correct way to create a graph. It is a design choice. This sort of dynamic graph generation process is, to the best of my knowledge, a novel introduction to the field of graph learning.

The rest of this paper proceeds as follows: I first define a mathematical model describing how, under certain assumptions, it may be possible to infer a graph structure on a collection of time series. Next, I describe an experiment setup on simulated data to test if the method works. Finally, I present and interpret the results, and offer some concluding remarks to the viability of this algorithm and other modeling concerns.

3 Methodology

As in the models discussed earlier, I start with a dataset of n time series with a defined frequency, with the discrete time index taking values in $t \in \{0, 1, 2, \ldots, T\}$. Index the nodes with $i \in \{1, 2, \ldots, n\}$. Then, at each point in time t, each node i has value $x_i(t)$. Each $x_i(t)$ can be collected into an $n \times 1$ vector $\vec{x}_t$, where the i-th element in $\vec{x}_t$ is $x_i(t)$. The collection of all nodes will be referred to as the *system*, while the vector $\vec{x}_t$ will be referred to as the *state vector* at time t.

The approach taken in this paper assumes that these nodes are connected via an unknown graph that determines the time evolution of the entire system. I represent this graph as an adjacency matrix A. Each element in A, A_{ij}, represents the relative influence of series j on the value of series i. This is counterintuitive relative to what one would expect from a directed edge, but this result is simply derived from an interpretation of the influence of A_{ij} in the product $A\vec{x}_t$.

Additionally, let $\mu : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ be some differentiable function acting on a vector (for example, element-wise tanh). The purpose of this function is to add some additional (potentially nonlinear) effects to expand the flexibility of the model. μ acts on the vector product $A\vec{x}_t$ and determines the final step of deterministic time evolution of the system.

Some constraints must be placed on A, μ, or both in order to prevent the state vector from blowing up. In the case of modeling stationary series, one may be able to set A to be a contraction mapping (defined below), especially if μ is just the identity map. Another alternative would be to set μ's range to some finite subset of the real numbers. It may be possible to learn μ from data, which may require a neural network due to its potentially nonlinear nature, but this is beyond the scope of this paper. For the purpose of this study, I set μ to be an element-wise tanh function, which has range $[-1, 1]$, to add in some nonlinearity to the state evolution.

Definition 3.1 (Contraction Mapping)**.** Let d be a metric (distance function) on a set M. That is, $d(x, y)$ represents a distance between points x and y for $x, y \in M$. Then a function $A : M \to M$ is a contraction mapping if there exists a constant $k \leq 1$ such that $d(Ax, Ay) \leq kd(x, y)$

On top of the effect of the adjacency matrix A on the time evolution of the system, I also introduce a noise factor. Let $\vec{\eta}_t$ be an $n \times 1$ vector of Gaussian noise at time t with covariance matrix $C = I_n$, the n-dimensional identity matrix. However, the noise in one node may affect the noise in others through a *second* network (on the same nodes), represented by another adjacency matrix B. Similar to A, an element of this matrix B_{ij} indicates the extent to which the value of noise in node j influences the value of noise in node i. So, independent noise is generated for each node through an i.i.d Gaussian process, and the noise is then mixed together through the network represented by B.

The main goal of this paper, then, is to develop a mathematical and computational framework for simultaneously inferring A and B from the time series data associated with

each of the nodes. The key assumption is that the state vector of the system evolves according to the following state evolution equation:

$$\vec{x}_{t+1} = \mu(A\vec{x}_t) + B\vec{\eta}_t \tag{1}$$

Another way to interpret the adjacency matrices with Equation 1 in mind is to note that A is an adjacency matrix that holds information on deterministic state evolution. The matrix B, on the other hand, is an adjacency matrix that encodes information about correlated noise and "explainable randomness" in the system.

Since $\vec{\eta}$ is drawn from a multivariate Gaussian distribution, this state evolution equation is an affine transformation on said distribution. That is,

$$\vec{x}_{t+1} = \mu(A\vec{x}_t) + B\vec{\eta}_t \sim \mathcal{N}(\mu(A\vec{x}_t), BB^\top). \tag{2}$$

Writing this out in its functional form gives

$$P(\vec{x}_{t+1}|\vec{x}_t, A, B) = \frac{1}{(2\pi)^{\frac{n}{2}}} \frac{1}{\sqrt{|BB^\top|}} \exp\left[-\frac{1}{2}(\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top (BB^\top)^{-1}(\vec{x}_{t+1} - \mu(A\vec{x}_t))\right] \tag{3}$$

In order to prevent issues associated with floating-point precision in later computations, a log-likelihood function can be derived by taking the logarithm of both sides:

$$\begin{aligned} \log y &= \log P(\vec{x}_{t+1}|\vec{x}_t, A, B) \\ &= \log \prod_{t=0}^{T-1} p(\vec{x}_{t+1}|\vec{x}_t, A, B) \\ &= \sum_{t=0}^{T-1} \log p(\vec{x}_{t+1}|\vec{x}_t, A, B) \end{aligned} \tag{4}$$

Which then expands to

$$\begin{aligned} \log y &= \sum_{t=0}^{T-1} \log((2\pi)^{-\frac{n}{2})} + \log(|BB^\top|^{-\frac{1}{2}}) - \frac{1}{2}(\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top (BB^\top)^{-1}(\vec{x}_{t+1} - \mu(A\vec{x}_t)) \\ &= -\frac{1}{2}\sum_{t=0}^{T-1} n\log(2\pi) + \log(|BB^\top|) + (\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top (BB^\top)^{-1}(\vec{x}_{t+1} - \mu(A\vec{x}_t)) \end{aligned} \tag{5}$$

This allows the treatment of extremely small probabilities without having to worry about floating-point precision in computer systems. For example, a log-likelihood of -100 corresponds to a probability of $e^{-100} \approx 3.72 \times 10^{-42}$, which is incredibly small. Computers are notorious for making errors in floating point calculations even after 12 or 15 decimal places, making it absolutely essential to perform this transformation.

I then take gradients of the likelihood function in Equation 5 with respect to both A and B, and then optimize their parameters via gradient descent. See Appendix sections A.3 and A.6 for details on the derivations.

Taking the derivative with respect to the (i, j) element of A gives

$$\nabla_A \log y = \frac{\partial \log y}{\partial A_{ij}} = -\sum_{t=0}^{T-1} \vec{x}_{t,i}[\mu'(A\vec{x}_t)]_j[(BB^\top)^{-1}(\vec{x}_{t+1} - \mu(A\vec{x}_t))]_j, \tag{6}$$

Taking the derivative with respect to the (i, j) element of B gives

$$\begin{aligned} \nabla_B \log y = \frac{\partial \log y}{\partial B_{ij}} = -TB_{ij}B_{jj} - \\ \frac{1}{2}\sum_{t=0}^{T-1}\sum_{k=1}^{n}\sum_{l=1}^{n}(\vec{x}_{t+1} - \mu(A\vec{x}_t))_k(\vec{x}_{t+1} - \mu(A\vec{x}_t))_l\left[(BB^\top)^{-1}_{ki}[B^{-1}]_{jl} + (BB^\top)^{-1}_{il}[(B^\top)^{-1}]_{kj}\right]. \end{aligned} \tag{7}$$

With these gradients in hand, the likelihood function can be optimized with respect to the matrices A and B, and use the result to obtain an optimal graph representation of the time series data.

4 Time Complexity

A particularly astute reader may notice that the required computations for even a single step of gradient descent are incredibly intensive.

Note that each matrix, A and B, are of dimension $n \times n$. Square matrix multiplication and inversion, via the standard algorithms (textbook matrix multiplication and Gauss-Jordan elimination) each have a computational complexity of $O(n^3)$. Algorithms exist that optimize each of these to around $O(n^{2.4})$ [1], but as I cannot be sure without inspection into source code the method that `numpy` uses, I assume each operation is maximally complex, at $O(n^3)$. Additionally, some of these improved algorithms are *galactic algorithms*, where the input size of the problem needs to be so astronomically large to see asymptotic gains that the algorithm cannot be implemented in practice. Even in the case of algorithms that are closer to $O(n^{2.8})$ such as Strassen's Algorithm [29], they often are only valid for algorithms where n approaches large numbers, such as 100. It will become apparent in the rest of this section that requiring n to reach 100 is simply not feasible for this project.

Generally, in a degrees of freedom argument, one must have at least as many data points as parameters in order to optimize correctly. This puts a lower bound on T as well: between the two matrices A and B, each with n^2 elements, this means $T \geq 2n^2$.

The likelihood computation itself requires matrix multiplications, the worst of which being $O(n^3)$ in the final term in the sum. This is summed over all times T, so the overall complexity of just the likelihood calculation is $O(Tn^3)$. With the lower bound of T being $\Omega(n^2)$, the likelihood computation is at least $O(n^5)$. Small optimizations can be made by precomputing the values of $\log|BB^\top|$ and $(BB^\top)^{-1}$ at each optimization step to prevent having to compute the same value more than once, which is employed throughout the entire codebase along with some other values, but that does not change the asymptotic scaling.

The time complexity of computing single element of the gradient with respect to A is $O(n^3+Tn^2)$ for the matrix multiplications and the sum over t, assuming a precomputation of $(BB^\top)^{-1}$. However, since this needs to be computed for all n^2 elements of A, the algorithm is actually $O(Tn^4)$ (the n^3 term drops since it is only computed once). With the limit of $T \geq 2n^2$, this becomes, at best, $O(n^6)$.

The time complexity of computing a single element of the gradient with respect to B follows a similar pattern. The matrix multiplications lead to an overhead of $O(n^3)$, but they can be precomputed so they do not have to be done at each time step. Computing $(x_{t+1} - \mu(Ax_t))$, is an $O(n^2)$ operation which needs to be computed for each time step, giving $O(Tn^2)$. For each of those $O(Tn^2)$ operations, there are n^2 terms to compute and sum over. This leads to an overall time complexity of $O(n^3 + Tn^4) \implies O(n^6)$ for *each* element of B. So, the total complexity for computing the gradient with respect to B is $O(n^8)$.

Each of these time complexities are valid for a *single* optimization step. The worst computation is the gradient with respect to B, which is $O(n^8)$. So, for k optimization steps this gives an overall complexity of $O(kn^8)$ for the matrix inference algorithm.

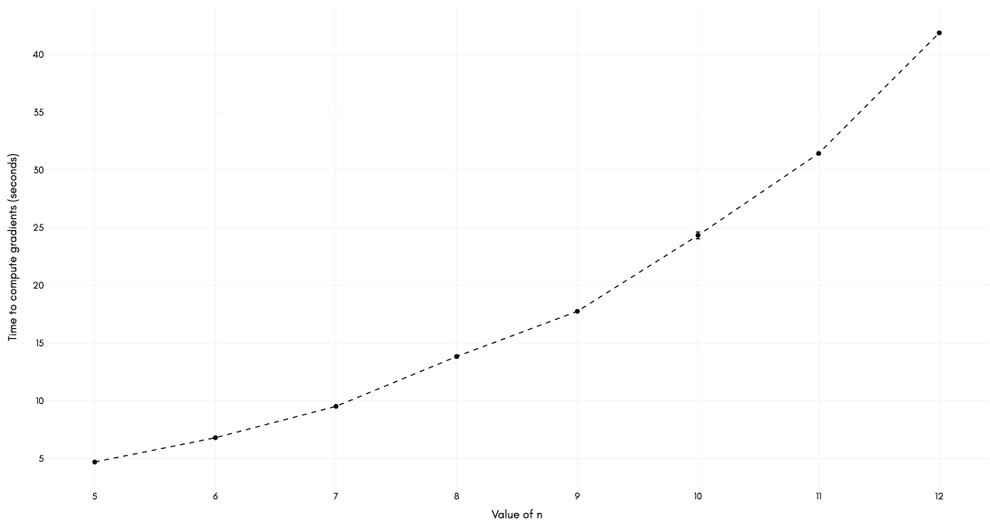
5 Technical Limitations and Experiment Setup

$O(kn^8)$ is an incredibly large time complexity. For example, if the optimization with $n = 10$ takes 1 second, then the optimization with $n = 100$ will take 10^8 seconds, or about 3.2 *years*. That's optimistic as well, since computing each of the gradients on simulated data for $n = 10$ takes about 30 seconds when parallelized across 6 CPU cores on an Apple M1 chip running at 3.2 GHz. For 100 optimization steps, this takes about 3000 seconds, or 50 minutes. Performing the optimization for $n = 100$, then, would take about $3000 \cdot 10^8$ seconds ≈ 4500 **years**.

To illustrate, consider Figure 4, detailing the average time in seconds to compute the gradients with respect to A and B for $n = 5, 6, \ldots, 12$. Note that it takes almost 10 times as long to compute the gradients for $n = 12$ than $n = 5$. These computations were performed on a Windows PC with a 3.9 GHz Intel i9 16-core processor, with computations being parallelized across 14 of the 16 available cores.

In the interest of not turning my laptop into an iToaster, I employ a simplification. Rather than letting B be a general $n \times n$ matrix, I let it be a constant c times the identity matrix I. This removes some of the complications associated with correlated noise, and allows for much simpler computation.

Figure 4: Average runtimes for computing the gradients with respect to A and B in seconds for $n = 5, 6, \ldots, 12$



.

To motivate this assumption, this can be interpreted as each node experiencing i.i.d. noise. That is, the noise in one node has no explicit effect on the noise in another, barring the indirect effect on state evolution for the next step. For example, noise for node 1 at time t will affect the state value for node 1 at time $t + 1$, but none others. Then, at time $t + 2$, node 2 may have its value affected by node 1, which contains the noise from time t. While the main goal of this assumption is to reduce computational complexity to make an experiment on simulated data feasible given my limited computing power, the results of said experiment may still be interpreted

Under this condition, the key equations reduce to

$$\log y = -\frac{1}{2}\sum_{t=0}^{T-1} n\log(2\pi) + 2n\log(c) + \frac{1}{c^2}(\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top(\vec{x}_{t+1} - \mu(A\vec{x}_t)) \tag{8}$$

$$\frac{\partial \log y}{\partial A_{ij}} = \frac{1}{c^2}\sum_{t=0}^{T-1} x_{t,i}[\mu'(A\vec{x}_t)]_j(\vec{x}_{t+1} - \mu(A\vec{x}_t))_j \tag{9}$$

$$\frac{\partial \log y}{\partial c} = -\frac{1}{2}\sum_{t=0}^{T-1}\frac{2}{c}n - \frac{2}{c^3}(\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top(\vec{x}_{t+1} - \mu(A\vec{x}_t)). \tag{10}$$

Equation 10 can be solved analytically for the optimal value of c by setting the derivative to 0 and solving for c:

$$c = \sqrt{\frac{1}{2nT}\sum_{t=0}^{T-1}(\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top(\vec{x}_{t+1} - \mu(A\vec{x}_t))}. \tag{11}$$

However, in the interest of simplicity (since c depends on A), I set it to a known constant, rather than attempting to infer it. This will allow straightforward measurement of the error incurred in inferring A as a function of the strength of the noise c.

The general experiment workflow that occurs in this paper is as follows:

- Generate simulated data according to Equation 1
- Store the true value of A and c.
- Generate an initial guess of A as a random matrix taking values in $[0, 1]$.
- Compute the gradient of the likelihood function with respect to this matrix. Call this $\mathcal{L}$.
- Update the value of A with $A \leftarrow A + \nu\nabla_A(\log y)$. The value ν is called the **learning rate**.

- Recompute the likelihood function. Call this $\mathcal{L}'$. For some small threshold $\varepsilon > 0$, repeat the gradient computation and update if $\mathcal{L}' - \mathcal{L} > \varepsilon$. If not, the current value of A is the maximally likely value (in a neighborhood around the current value).
- If the optimization reaches some threshold number of maximum steps N, then stop the optimization where it is.
- Since the data are simulated, I can compare the inferred values to the true value and see the degree of error in each parameter.

In my experiments, I set $\eta = 10^{-4}$, $\varepsilon = 0.1$ and $N = 30$. I carry out 2 major experiments.

5.1 Experiment 1: Global Optimum

The goal of Experiment 1 is to see if multiple optimizations run on the same dataset result in the same estimate of A when they start at different points. I generate a dataset with $c = 1$ and a known value of A with parameter values in $[0, 1]$. I then initialize 10 random matrices, also with values in $[0, 1]$, and attempt to optimize them and store the results. With these results, I can determine two things:

1. How consistent each of the estimates are with each other
2. How far off the average estimate is from the true value.

If all of the estimates are close together, I can infer that they are all initial guesses are converging to the same local maximum. If the average estimate is perfect for each parameter in A, I can further conclude that the optimal value from the optimization is a global maximum.

If each initial value converges to a different value, then it can be reasonably assumed that the optimization problem is non-convex and finding a global solution is difficult to do efficiently using numerical methods.

5.2 Experiment 2: Estimate Errors

The goal of this experiment is to estimate the average error in each value of A after the optimization in order to gauge the effectiveness of the optimization algorithm.

Denote $\hat{A}$ to be the estimated value of A after optimization, and A_0 to be the initial guess. I then take $\hat{E} = \hat{A} - A$ to be the post-optimization error matrix (how far off each element is from the true value) and $E_0 = A_0 - A$ to be the pre-optimization error matrix.

Then, I compare the values of $|\hat{E} - E_0|$. If an element of $|\hat{E} - E_0|$ is negative, then that element moved closer to the true value after the optimization (since its error decreased). If it is positive, the element after the optimization is further from the true value than before. This allows the ability to gauge whether or not the optimization is *actually* doing anything in terms of getting closer to the true value of A.

I also quantify the value of error by estimating 95% confidence intervals for the average error produced in an optimization. For this, I perform 10 separate optimizations, each with its own generated dataset and initial guess, for each value of c. Then, for each value of c, I estimate a 95% confidence interval for the average error in the inference.

6 Results

6.1 Experiment 1 Results

In order to visualize the consistency of the estimates, I create a heatmap in which each cell represents the standard deviation of the estimated values for all 10 estimates performed on the data, as shown in Figure 5. The top left cell corresponds to $A_{1,1}$, with the cell to the right representing $A_{1,2}$, etc.

Figure 5: Heatmap of the standard deviation of matrix values across 10 estimates of A, with $c = 1$.

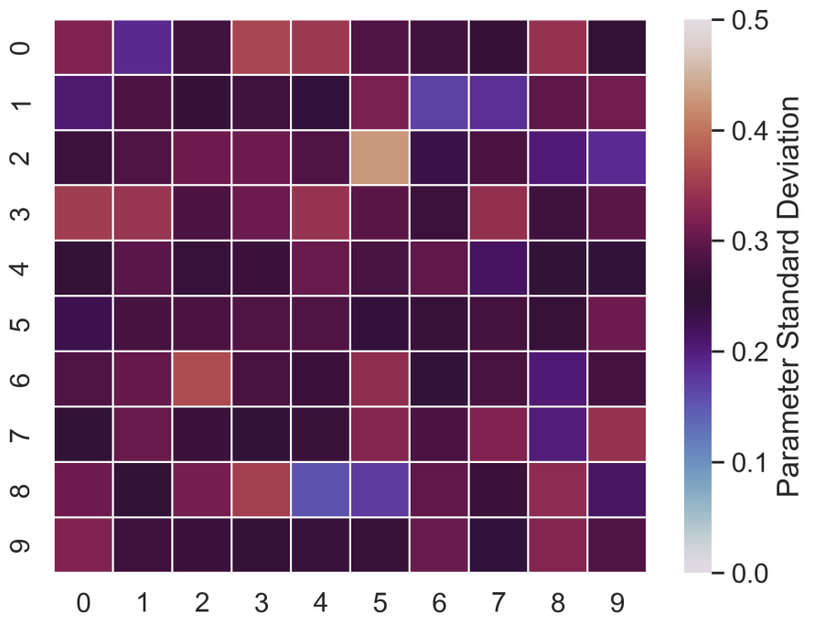
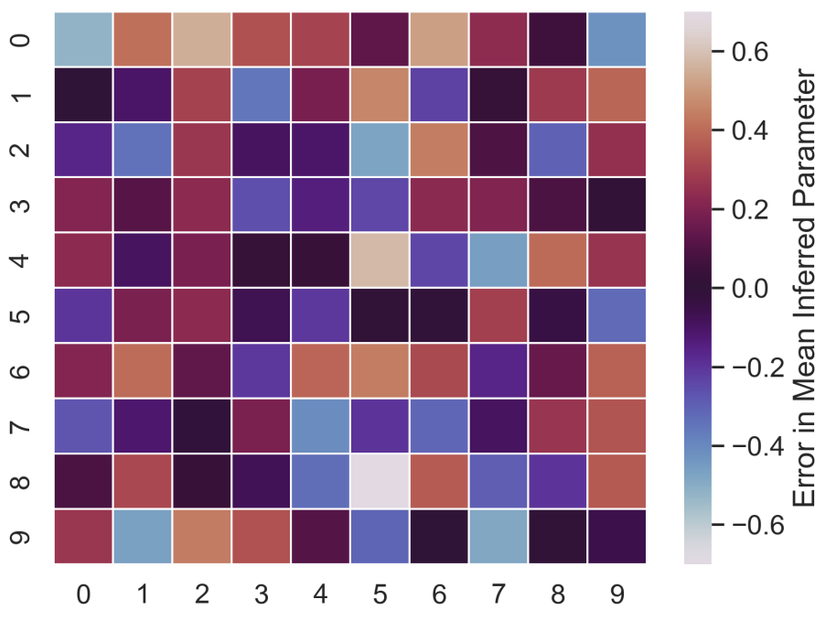


Figure 6: Heatmap of the difference in the average estimate of A and the true value of A, $\mathbb{E}[\hat{A}] - A$.



This figure shows a somewhat large degree of variation between the values, with optimized values tending to differ in magnitude by about 0.2 to 0.3 from the mean estimate. As such, it can be inferred that the estimates are not the same. Each trial seems to be converging to some local maximum of the likelihood function, differing from the local maxima that initial guesses converge to.

The result in Figure 5 begs the question: given the large variation of estimates, does the average estimate for each parameter well approximate the true value used to generate the data? That is, are these seemingly disparate estimates at least centered around the true value? Figure 6 attempts to answer this question by plotting a heatmap of $\mathbb{E}[\hat{A}] - A$, the average post-optimization estimate minus the true value of A. Results show varying amounts of success (values near 0 are accurate), but the general result is that the average estimate does not well approximate the true value of A.

6.2 Experiment 2 Results

Given the results of Section 6.1, it may be helpful to quantify the degree of error incurred in estimation.

Firstly, I plot a heatmap for an experiment with $c = 1$, where values indicate $|\hat{E} - E_0|$, the difference in error after and before the optimization. This is shown in Figure 7. Values that take a purple or blue color represent edges that improved post-optimization, whereas red-shaded values represent nodes that incurred more error post-optimization. Many values are near 0, and the rest appear to be relatively evenly distributed between decreasing error and increasing error.

Moving beyond single-optimization results and towards the statistics of all of the experiments, Figure 8 displays the average number of post-optimization parameter improvements for various values of c, each computed using 10 trials each.

Figure 7: Heatmap of the difference in the error after and before the optimization, $|\hat{E} - E_0|$.

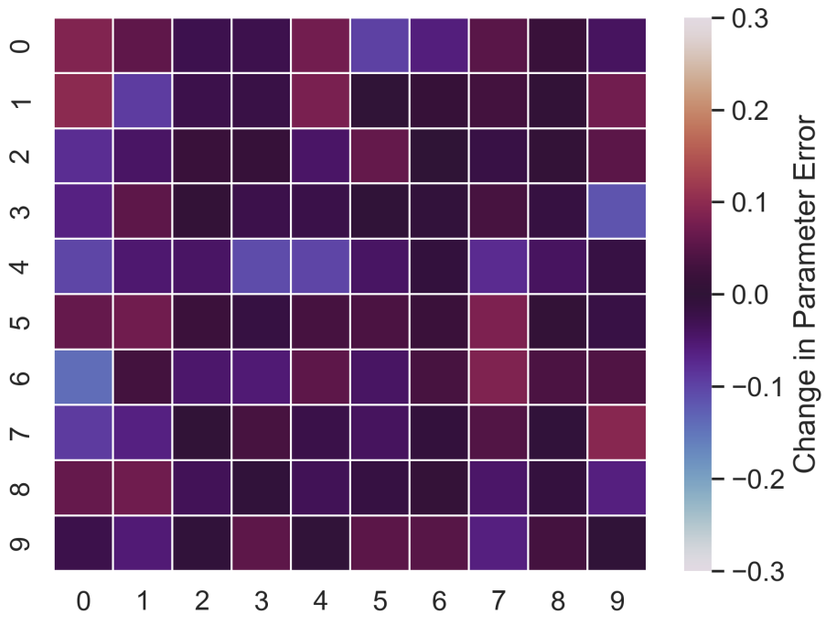
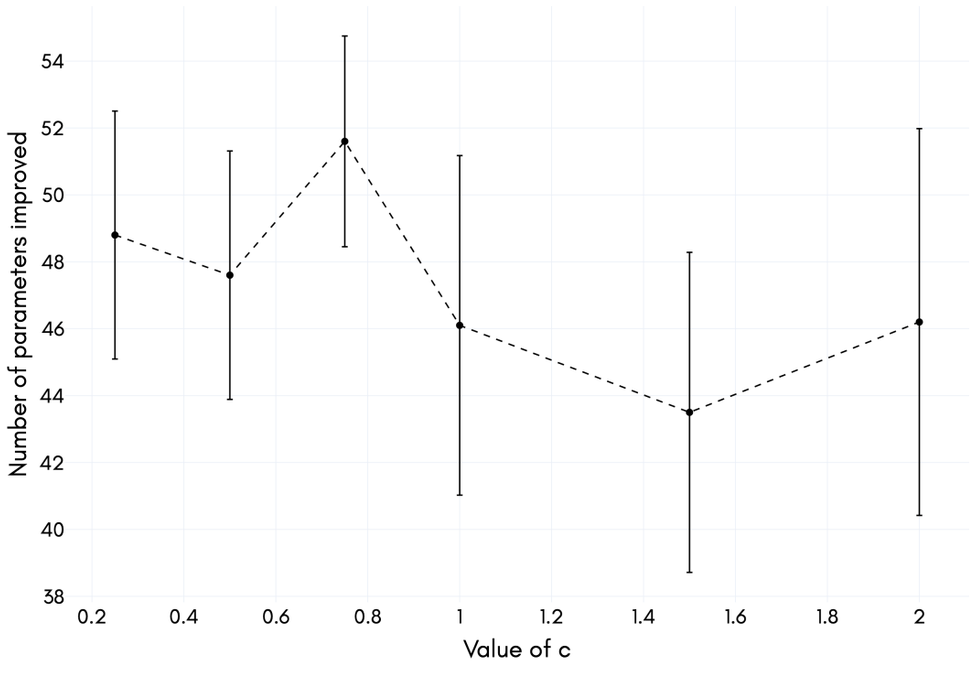


Figure 8: Distribution of 95% confidence intervals for the number of parameters that improved over 10 trials for each value of c

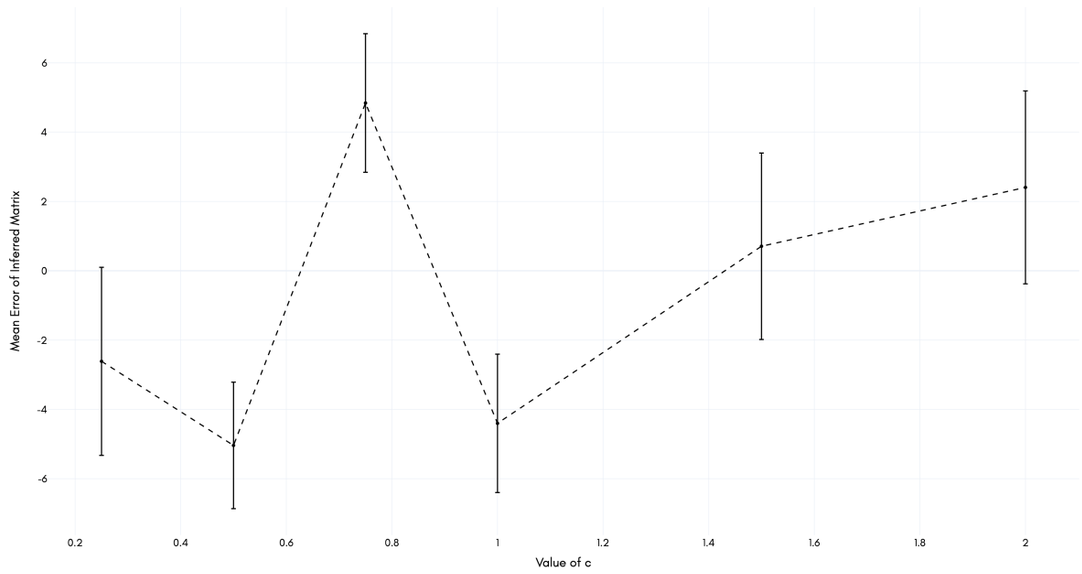


The trials were run for values of $c \in \{0.25, 0.5, 0.75, 1, 1.5, 2\}$. It should be noted that

the value to compare to is 50 – the expected number of improvements if every parameter of a random guess is incremented or decremented by a constant. For all values of c except $c = 0.75$, the sample average is below 50, indicating that, on average, the parameter inference does not perform well. Additionally, notice for $c = 1.5$, the entire confidence interval is below 50, indicating that the probability that the optimization actually brings parameters closer to their true values is less than 2.5%. In general, this means it cannot be concluded with high probability that the true average parameter improvement is better than random.

The number of improved parameters, however, does not necessarily quantify the exact amount of improvement. For that, I examine the average error value for each value of c. In particular, for each value of c, the average error value across all 10 trials was computed, and the 95% confidence intervals for the true average error for each value of c are shown in Figure 9.

Figure 9: Distribution of 95% confidence intervals for the average post-optimization error value.



Unfortunately, the results of the mean error analysis are not incredibly promising. Error values seem to be incredibly large, given that all true values lie in the interval $[0, 1]$. However, given the non-convexity of the problem, this is largely to be expected. Most estimates latch onto local minima in their neighborhoods, which may be far from optimal on the global scale.

7 Analysis and Conclusion

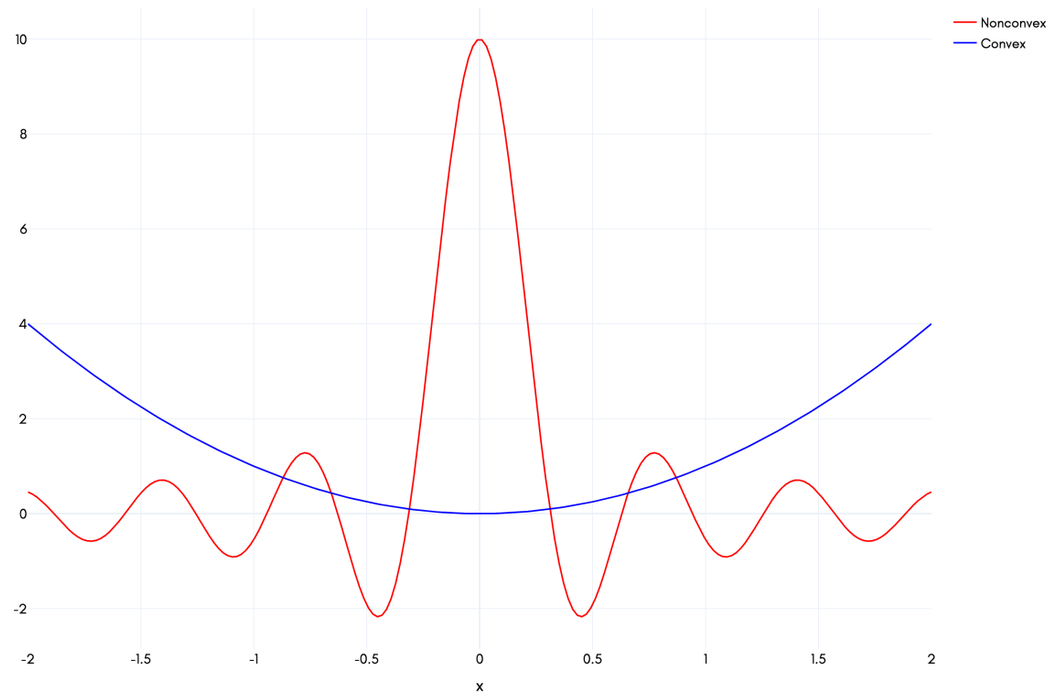
The results of the experiments on simulated data seem to point the same way: the optimization algorithm, in its current state, is not suitable to be used in data analysis.

The mathematical results present an incredibly computationally intensive problem that is not feasible for many applications. For example, if one wanted to apply a graph learning algorithm to financial data for the purpose of gaining trading advantages, spening an hour to compute a graph on 10 tickers would be largely useless for actual applications. This is largely why classical methods, which can be efficiently vectorized and parallelized with libraries such as `numpy` and `dask` in Python, are typically preferable to dynamic methods. Although they do not share the strength of the dynamic model–namely, having edges that theoretically guarantee predictive power, they can often be computed in under a second, allowing for more frequent use.

Furthermore, the unpredictable relationship between the error in the inferred value of A and the strength of the noise c indicates an unstable optimization that is largely dependent on initial conditions. The likelihood function is dotted with local maxima all over its parameter space, making efficient optimization incredibly difficult. For problems such as these, oftentimes the only way to effectively optimize the function is to perform a grid search. However, with 100 parameters governing the evolution of 10 nodes, this is also incredibly computationally intensive, and even less feasible to use in an applied setting.

The issue of the mean number of parameters improving post-optimization being less than random is a natural consequence of the likelihood function being nonconvex. Consider the example shown in Figure 10. If I attempt to minimize the convex function (detailed in blue), the starting location does not matter–following the path of steepest descent will lead to the global minimum regardless of what initial guess is chosen. However, for the nonconvex function, consider what happens if the initial guess x_0 is chosen to be $x_0 = 1$. In this case, a gradient descent algorithm would choose to follow the path of steepest descent and would land at a local minimum near $x = 1.2$. While this certainly does result in a more optimal value for the objective function, the inferred value of x ends up deviating further from the value of x that would produce the global minimum, near $x = 0.5$ and $x = -0.5$. In analogy with the results of this paper, the likelihood function was reaching a local maximum after optimizations were completed, but the parameters that generate this maximum may have gone in a separate direction from the values that would have generated the global maximum, but in a 100-dimensional parameter space instead of the 1-dimensional parameter space detailed in this example.

Figure 10: Example of a convex and nonconvex function.



Further work in this field must first address the issue of computational complexity. Oftentimes, computationally expensive operations such as matrix multiplications can be sped up with vectorization, as it allows for many for computations per CPU clock cycle than element-wise operations. A large bottleneck of the computation of gradients is the nested sums, especially when computing the gradient with respect to B. As such, if one could find a way to reduce the term in the sum of equation 7 to a matrix operation, one could represent each term in the sum as a 2-dimensional slice of a 3-tensor with dimensions (i, j, t). One could then leverage parallelization and vectorization to quickly compute the terms of the sum and sum the tensor along the time axis. In general, the more these operations can be written in terms of matrix multiplications and tensor operations, the faster the code can run on most machines.

However, no efficiency improvements can address the problem of nonconvexity. Nonconvex optimization is at least NP-hard, meaning any problem in NP can be reduced to this problem in polynomial time. That means there is no way to achieve truly optimal results in polynomial time in the input size. Rather, only the solution can be checked in polynomial time. However, in order to completely prove that this problem is NP-hard, a concrete proof must be presented that the optimization problem is truly nonconvex. As of right now, the results of Experiment 1 point to this fact, but a mathematical proof does not exist.

Nonconvex optimization methods do exist in some capacity, with the most popular

method being the solution generally employed in training deep neural networks: throwing a massive amount of computational power at the problem. In general, techniques exist to try and solve nonconvex optimization problems *better* than a grid search, though any theoretical guarantees on convergence to the optimal value are weak, if they even exist. An overview of nonconvex optimization for the interested reader can be found in [7]. Given that the approach applied in this paper is simple gradient ascent, a method tailored specifically for nonconvex problems may produce more promising optimization results. Balancing computational complexity and accuracy is a difficult goal, but if nonconvex methods can be used in conjunction with mathematical sleight-of-hand in rewriting equations 6 and 7, the optimization may become feasible with enough computing power.

Beyond the issue of computational complexity, from a modeling perspective it may be detrimental to assume every edge in the graph has explanatory power, as this method of modeling may lend itself to overfitting. I could use 100 parameters to model the evolution of 10 series in the experiment of this paper since the data were actually generated from this type of process, but this may not be true for all processes. It may prove more helpful to think about only the most important factors in determining the graph's evolution.

Oftentimes, when needing to eliminate parameters from models, machine learning practitioners will turn to L^1 regularization. This involves adding an extra term to the objective function that penalizes larger model parameters using an L^1 norm, defined on a vector as the sum of the absolute value of individual elements:

$$\|\vec{a}\|_1 = \sum_{i=1}^{N} |a_i| \tag{12}$$

So, as a modeling prospect, one could modify the likelihood function to penalize larger values of these parameters. This could be achieved by subtracting the term $\lambda_A \|\vec{a}\|_1 + \lambda_B \left\|\vec{b}\right\|_1$ from the log-likelihood function, where λ_A and λ_B are real-valued parameters that define the amount of artificial penalty incurred by increasing a parameter's value. The vectors $\vec{a}$ and $\vec{b}$ represent the entries in A and B turned into an $\mathbb{R}^{n^2}$ vector. The exact method of doing this conversion is irrelevant, since norm computations are agnostic to the position of each element. This method, of course, adds additional hyperparameters that may need to be tuned, but could be worth including because L^1 regularization is specially designed to encourage model sparsity – that is, models that only have a few non-zero parameters.

While the allure of truly dynamic graph inference is certainly enticing, it is unfortunately not realistic yet. The feasibility of this method is likely to grow in tandem with developments in nonconvex optimization. For now, classical graph generation methods may reign supreme in striking the balance between utility and computing time, but this is likely to change in the future.

A Derivation of Gradients

A.1 Notation

For the derivations that follow, I will use the following notation:

- J^{ij} denotes a square **single-entry matrix** whose elements are all zero except for a 1 at element (i, j).
- Diag(A) for a square matrix A represents a copy of the matrix A where all non-diagonal elements are set to 0.
- δ_{ij} denotes the **Kronecker delta**, which equates to 1 if $i = j$ and 0 if $i \neq j$.

A.2 Derivative of $(\vec{x}_{t+1} - \mu(A\vec{x}_t))$ with respect to A

I tackle this derivative element-wise. Define a function $f : \mathbb{R}^n \to \mathbb{R}^n$ defined by $f = (x_{t+1} - \mu(Ax_t))$. The k-th element of f can be computed as

$$f_k = ((x_{t+1})_k + \mu(\sum_{\nu=1}^{n} x_\nu A_{k\nu}).$$

Taking the derivative of this expression with respect to A_{ij} gives

$$\frac{\partial f_k}{\partial A_{ij}} = \frac{\partial}{\partial A_{ij}} \mu(\sum_{\nu=1}^{n} x_\nu A_{k\nu}).$$

Apply the chain rule and linearity of the differentiation operator[2]:

$$\begin{aligned}
\frac{\partial f_k}{\partial A_{ij}} &= \mu'(A\vec{x}_t)_k \sum_{\nu=1}^{n} x_\nu \frac{\partial A_{\nu k}}{\partial A_{ij}} \\
&= \mu'(A\vec{x}_t)_k \sum_{\nu=1}^{n} x_\nu \delta_{\nu i}\delta_{kj} \\
&= \mu'(A\vec{x}_t)_k x_i \delta_{kj},
\end{aligned}$$

which is a rank-3 tensor.

A.3 Derivation of $\nabla_A \log y$

Recall that

$$\log y = -\frac{1}{2}\sum_{t=0}^{T-1} nlog(2\pi) + \log(|BB^\top|) + (\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top (BB^\top)^{-1}(\vec{x}_{t+1} - \mu(A\vec{x}_t)).$$

Taking the derivative of this expression with respect to A_{ij} gives us

$$\frac{\partial \log y}{\partial A_{ij}} = -\frac{1}{2}\sum_{t=0}^{T-1} \frac{\partial}{\partial A_{ij}} (\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top (BB^\top)^{-1}(\vec{x}_{t+1} - \mu(A\vec{x}_t)). \tag{13}$$

Focusing on the term in the sum, note that letting $z = (\vec{x}_{t+1} - \mu(A\vec{x}_t))$ and $Q = (BB^\top)^{-1}$, this is a scalar quantity. Specifically, it is a real quadratic form $q(z; Q) = z^\top Qz$. This can be written out element-wise as

$$q(z; Q) = \sum_{k=1}^{n}\sum_{l=1}^{n} Q_{kl} z_k z_l$$

In Appendix section A.2, I discussed how to take the derivative of each of the z_k and z_l terms with respect to A_{ij}, and Q is a constant with respect to A since it only depends on

[2]Note to self: Review real analysis notes at some point to make this statement more accurate

B. Also, since Q is the inverse of a symmetric matrix $BB^\top$, it is also symmetric – that is, $Q = Q^\top$. Taking the derivative of this expression with respect to A_{ij},

$$\begin{aligned}
\frac{\partial q(z;Q)}{\partial A_{ij}} &= \sum_{k=1}^{n}\sum_{l=1}^{n} Q_{kl}\frac{\partial}{\partial A_{ij}}[z_k z_l] \\
&= \sum_{k=1}^{n}\sum_{l=1}^{n} Q_{kl}\frac{\partial}{\partial A_{ij}}[z_k]z_l + \sum_{k=1}^{n}\sum_{l=1}^{n} Q_{kl}\frac{\partial}{\partial A_{ij}}[z_l]z_k \\
&= \sum_{k=1}^{n}\sum_{l=1}^{n} Q_{kl}[\mu'(A\vec{x}_t)]_k x_i \delta_{kj} z_l + \sum_{k=1}^{n}\sum_{l=1}^{n} Q_{kl}[\mu'(A\vec{x}_t)]_l x_i \delta_{lj} z_k \\
&= \sum_{l=1}^{n} Q_{jl}[\mu'(A\vec{x}_t)]_j x_i z_l + \sum_{k=1}^{n} Q_{kj}[\mu'(A\vec{x}_t)]_j x_i z_k \qquad (14)\\
&= [\mu'(A\vec{x}_t)]_j x_i \big[\sum_{l=1}^{n} Q_{jl} z_l + \sum_{k=1}^{n} Q_{kj} z_k\big] \\
&= [\mu'(A\vec{x}_t)]_j x_i \big[(Qz)_j + \sum_{k=1}^{n} Q^\top_{jk} z_k\big] \\
&= [\mu'(A\vec{x}_t)]_j x_i \big[(Qz)_j + (Q^\top z)_j\big] \\
&= 2[\mu'(A\vec{x}_t)]_j x_i (Qz)_j.
\end{aligned}$$

Plugging this into the likelihood function 13 and substituting the values of z and Q gives

$$\frac{\partial \log y}{\partial A_{ij}} = -\sum_{t=0}^{T-1} x_{t,i}[\mu'(A\vec{x}_t)]_j[(BB^\top)^{-1}(\vec{x}_{t+1} - \mu(A\vec{x}_t))]_j, \qquad (15)$$

and the gradient (Jacobian matrix) of y with respect to A can be computed as the collection of all elements (i, j) for this term:

$$\nabla_A \log y = \left[\frac{\partial \log y}{\partial A_{ij}}\right]_{ij} \qquad (16)$$

A.4 Directional Derivative of $\log\det(XX^\top)$

First, I list the identities used to complete this proof. The first is **Jacobi's Identity**, which reformulates the log determinant of a matrix A in terms of the logarithm of its trace:

$$\log|A| = \operatorname{Tr}\log(A) \qquad (17)$$

The logarithm of a matrix is defined such that applying the matrix exponential $\exp(A) = \sum_{k=0}^{\infty}\frac{1}{k!}X^k$ to $\log(A)$ will produce the identity matrix. The matrix logarithm can be shown to be

$$\log(A) = \sum_{k=1}^{\infty}(-1)^{k+1}\frac{(A-I)^k}{k} = (A-I) - \frac{(A-I)^2}{2} + \frac{(A-I)^3}{3} + \dots. \qquad (18)$$

Note that multiplying a single entry matrix by its transpose gives us

$$J^{ij}(J^{ij})^\top = J^{ii}.$$

For some matrix X, note that

$$XJ^{ij} = [X_1\ X_2\ \dots\ X_n]J^{ij} = [0\ 0\ \dots\ 0\ X_j\ 0\ \dots\ 0]$$

Finally, note that the trace is a linear operator. So,

$$\begin{aligned}
\operatorname{Tr}(A+B) &= \operatorname{Tr}(A) + \operatorname{Tr}(B) \\
cTr(A) &= Tr(cA),\ c \in \mathbb{R}
\end{aligned}$$

Now, on to the proof. First let $f(X) = \log|XX^\top|$. Assume X has full rank, such that $XX^\top$ is invertible. Now, to compute the directional derivative with respect to the element (i, j) of some matrix B, I use the limit definition of a derivative:

$$\begin{aligned}
\frac{\partial f(X)}{\partial B_{ij}} &= \lim_{h\to 0}\frac{1}{h}\Big(\log|(X+hb_{ij}J^{ij})(X+hb_{ij}J^{ij})^\top| - \log(|XX^\top|\Big) \\
&= \lim_{h\to 0}\frac{1}{h}\Big(\log|XX^\top + hb_{ij}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})| - \log|XX^\top|\Big) \\
&= \lim_{h\to 0}\frac{1}{h}\Big(\log|(XX^\top)[I + hb_{ij}(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})]| - \log|XX^\top|\Big) \\
&= \lim_{h\to 0}\frac{1}{h}\Big(\log|(XX^\top)||I + hb_{ij}(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})| - \log|XX^\top|\Big) \\
&= \lim_{h\to 0}\frac{1}{h}\Big(\log|(XX^\top)| - \log|(XX^\top)| + \log|I + hb_{ij}(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})|\Big) \\
&= \lim_{h\to 0}\frac{1}{h}\Big(\log|I + hb_{ij}(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})|\Big) \\
&= \lim_{h\to 0}\frac{1}{h}\Big(\operatorname{Tr}(\log[I + hb_{ij}(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})])\Big) \\
&= \lim_{h\to 0}\frac{1}{h}\Big(\operatorname{Tr}(hb_{ij}(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii}) \\
&+ \frac{1}{2}h^2b_{ij}^2(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})^2) + O(h^3)\Big) \\
&= \lim_{h\to 0}\Big(\operatorname{Tr}[b_{ij}(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})] \\
&+ \frac{1}{2}hb_{ij}^2(XX^\top)^{-1}(XJ^{ji} + J^{ij}X^\top + hJ^{ii})^2) + O(h^2)\Big) \\
&= \operatorname{Tr}\left(b_{ij}[XJ^{ji} + J^{ij}X^\top]\right) \\
&= b_{ij}\left[\operatorname{Tr}(XJ^{ji}) + \operatorname{Tr}(J^{ij}X^\top)\right] \\
&= b_{ij}\left[\operatorname{Tr}(XJ^{ji}) + \operatorname{Tr}((XJ^{ji})^\top)\right] \\
&= 2b_{ij}\operatorname{Tr}(XJ^{ij}) \\
&= 2b_{ij}X_{jj}
\end{aligned} \tag{19}$$

In matrix form, if X and B are both $n \times n$ square matrices, the gradient in can be represented in matrix form as

$$\frac{\partial \log|(XX)^\top|}{\partial B} = 2B\operatorname{Diag}(X), \tag{20}$$

where each element represents

$$\frac{\partial \log|(XX^\top)|}{\partial B_{ij}}$$

.

A.5 Derivative of $(BB^\top)^{-1}$

First note that, by the definition of an inverse,

$$(BB^\top)(BB^\top)^{-1} = I.$$

Taking the derivative of both sides with respect to B_{ij} gives

$$\begin{aligned}
0 &= \frac{\partial}{\partial B_{ij}}\left[(BB^\top)(BB^\top)^{-1}\right] \\
&= \frac{\partial(BB^\top)}{\partial B_{ij}}(BB^\top)^{-1} + \frac{\partial(BB^\top)^{-1}}{\partial B_{ij}}(BB^\top),
\end{aligned}$$

which can then be used to solve for $\frac{\partial (BB^\top)^{-1}}{\partial B_{ij}}$:

$$\frac{\partial (BB^\top)^{-1}}{\partial B_{ij}} = -(BB^\top)^{-1} \frac{\partial (BB^\top)}{\partial B_{ij}} (BB)^{-1}. \tag{21}$$

This is most easily expanded element-wise. To do so, the right-hand side of Equation (21) needs to be explicitly evaluated in terms of the indices i and j. Letting $Q = (BB^\top)^{-1}$, the matrix multiplication can be expanded as

$$\frac{\partial (BB^\top)^{-1}_{kl}}{\partial B_{ij}} = \sum_{r=1}^{n} Q_{kr} \sum_{q=1}^{n} \frac{\partial (BB^\top)_{rq}}{\partial B_{ij}} Q_{ql}. \tag{22}$$

The derivative in the middle evaluates into a 4-tensor:

$$\begin{aligned}
\frac{\partial (BB^\top)_{rq}}{\partial B_{ij}} &= \frac{\partial}{\partial B_{ij}} \sum_{w=1}^{n} B_{rw} B^\top_{wq} \\
&= \sum_{w=1}^{n} \frac{\partial B_{rw}}{\partial B_{ij}} B^\top_{wq} + B_{rw} \frac{\partial B^\top_{wq}}{\partial B_{ij}} \\
&= \sum_{w=1}^{n} \frac{\partial B_{rw}}{\partial B_{ij}} B_{qw} + B_{rw} \frac{\partial B_{qw}}{\partial B_{ij}} \\
&= \sum_{w=1}^{n} \delta_{ri} \delta_{wj} B_{qw} + B_{rw} \delta_{qi} \delta_{wj} \\
&= \delta_{ri} B_{qj} + \delta_{qi} B_{rj}
\end{aligned} \tag{23}$$

This can be used to compute the gradient derivative in Equation 21, assuming that B is invertible so $(BB^\top)^{-1} = (B^\top)^{-1} B^{-1}$:

$$\begin{aligned}
\frac{\partial (BB^\top)^{-1}_{kl}}{\partial B_{ij}} &= \sum_{r=1}^{n} Q_{kr} \sum_{q=1}^{n} \frac{\partial (BB^\top)_{rq}}{\partial B_{ij}} Q_{ql} \\
&= \sum_{r=1}^{n} Q_{kr} \sum_{q=1}^{n} (\delta_{ri} B_{qj} + \delta_{qi} B_{rj}) Q_{ql} \\
&= \sum_{r=1}^{n} Q_{kr} \sum_{q=1}^{n} \delta_{ri} B_{qj} Q_{ql} + \sum_{r=1}^{n} Q_{kr} \sum_{q=1}^{n} \delta_{qi} B_{rj} Q_{ql} \\
&= Q_{ki} \sum_{q=1}^{n} B_{qj} Q_{ql} + Q_{il} \sum_{r=1}^{n} Q_{kr} B_{rj} \\
&= Q_{ki} \sum_{q=1}^{n} B^\top_{jq} Q_{ql} + Q_{il} \sum_{r=1}^{n} Q_{kr} B_{rj} \\
&= Q_{ki} [B^\top Q]_{jl} + Q_{il} [QB]_{kj} \\
&= (BB^\top)^{-1}_{ki} [B^\top (BB^\top)^{-1}]_{jl} + (BB^\top)^{-1}_{il} [(BB^\top)^{-1} B]_{kj} \\
&= (BB^\top)^{-1}_{ki} [B^\top (B^\top)^{-1} B^{-1}]_{jl} + (BB^\top)^{-1}_{il} [(B^\top)^{-1} B^{-1} B]_{kj} \\
&= (BB^\top)^{-1}_{ki} [B^{-1}]_{jl} + (BB^\top)^{-1}_{il} [(B^\top)^{-1}]_{kj}.
\end{aligned} \tag{24}$$

A.6 Derivation of $\nabla_B \log y$

Recall that

$$\log y = -\frac{1}{2} \sum_{t=0}^{T-1} nlog(2\pi) + \log(|BB^\top|) + (\vec{x}_{t+1} - \mu(A\vec{x}_t))^\top (BB^\top)^{-1} (\vec{x}_{t+1} - \mu(A\vec{x}_t)).$$

Note that the second term in the sum is a real quadratic form. Let $z = (\vec{x}_{t+1} - \mu(A\vec{x}_t))$ and $Q = (BB^\top)^{-1}$. Taking the derivative of this expression with respect to B_{ij} gives

$$\nabla_B \log y = \frac{\partial \log y}{\partial B_{ij}} = -\frac{1}{2}\sum_{t=0}^{T-1} \frac{\partial \log |BB^\top|}{\partial B_{ij}} + \frac{\partial}{\partial B_{ij}}[z^\top Q z].$$

The derivative of the term $\log |BB^\top|$ is covered in section A.4. For the real quadratic form, it can be expanded element-wise as

$$\begin{aligned}
\frac{\partial}{\partial B_{ij}}[z^\top Q z] &= \sum_{k=1}^{n}\sum_{l=1}^{n} z_k z_l \frac{\partial}{\partial B_{ij}}[Q_{kl}] \\
&= \sum_{k=1}^{n}\sum_{l=1}^{n} z_k z_l \frac{\partial}{\partial B_{ij}}[(BB^\top)^{-1}_{kl}] \\
&= \sum_{k=1}^{n}\sum_{l=1}^{n} z_k z_l \Big[(BB^\top)^{-1}_{ki}[B^{-1}]_{jl} + (BB^\top)^{-1}_{il}[(B^\top)^{-1}]_{kj}\Big].
\end{aligned} \tag{25}$$

Putting these two together:

$$\begin{aligned}
\frac{\partial \log y}{\partial B_{ij}} &= -\frac{1}{2}\sum_{t=0}^{T-1} 2B_{ij}B_{jj} + \sum_{k=1}^{n}\sum_{l=1}^{n} z_k z_l \Big[(BB^\top)^{-1}_{ki}[B^{-1}]_{jl} + (BB^\top)^{-1}_{il}[(B^\top)^{-1}]_{kj}\Big] \\
&= -TB_{ij}B_{jj} - \\
&\frac{1}{2}\sum_{t=0}^{T-1}\sum_{k=1}^{n}\sum_{l=1}^{n}(\vec{x}_{t+1} - \mu(A\vec{x}_t))_k(\vec{x}_{t+1} - \mu(A\vec{x}_t))_l \Big[(BB^\top)^{-1}_{ki}[B^{-1}]_{jl} + (BB^\top)^{-1}_{il}[(B^\top)^{-1}]_{kj}\Big].
\end{aligned} \tag{26}$$

And finally, these values can be put into a single matrix

$$\nabla_B \log y = \left[\frac{\partial \log y}{\partial B_{ij}}\right]_{ij} \tag{27}$$

B Acknowledgements

First and foremost, I want to thank Professor Sarah Marzen of the Keck Science Department. Her guidance in developing the mathematical framework of this project to formalize my idea was absolutely essential. Beyond this thesis, she has supported me in other research and has always pushed me to never take the easy route in anything I do–without her guidance, I would certainly not have been able to produce much of the work I have over the last 4 years.

Furthermore, I thank Professor Julio Garín of Claremont McKenna College and Professor Scot Gould of the Keck Science department for assisting me in drafting and reading the paper, despite unfamiliarity with the subject matter. I also thank Professor Adam Landsberg of the Keck Science Department for lending his expertise and essential feedback to this project.

To my parents, Krantee Uppal and Vinesh Uppal, I thank you for supporting me and helping me form the work ethic and habits I needed to excel at CMC.

To my incredible friends at CMC, I thank you for supporting me through thick and thin. Specifically, I want to thank Kartikeya Agarwal, Lauren Kula, Lauren Cashdan, Bettina Benitez, Cindy Lay, Aryaman Gulati, and Edward Wu for being the greatest friends anyone could ask for. All of your support was essential to me throughout this process, and I could not be more grateful.

Last, but certainly not least, I want to thank the incredible faculty that have guided and supported me through the last 4 years–the lessons I learned are unforgettable.

References

[1] Josh Alman and Virginia Vassilevska Williams. *A Refined Laser Method and Faster Matrix Multiplication*. 2020. DOI: 10.48550/ARXIV.2010.05846. URL: https://arxiv.org/abs/2010.05846.

[2] Fabrizio Altarelli et al. "Bayesian Inference of Epidemics on Networks via Belief Propagation". In: *Phys. Rev. Lett.* 112 (11 Mar. 2014), p. 118701. DOI: 10.1103/PhysRevLett.112.118701. URL: https://link.aps.org/doi/10.1103/PhysRevLett.112.118701.

[3] Monica Billio et al. "Econometric measures of connectedness and systemic risk in the finance and insurance sectors". In: *Journal of Financial Economics* 104.3 (2012). Market Institutions, Financial Market Risks and Financial Crisis, pp. 535–559. ISSN: 0304-405X. DOI: https://doi.org/10.1016/j.jfineco.2011.12.010. URL: https://www.sciencedirect.com/science/article/pii/S0304405X11002868.

[4] Lorenzo Giada and Matteo Marsili. "Algorithms of maximum likelihood data clustering with applications". In: *Physica A: Statistical Mechanics and its Applications* 315.3-4 (Dec. 2002), pp. 650–664. ISSN: 0378-4371. DOI: 10.1016/s0378-4371(02)00974-3. URL: http://dx.doi.org/10.1016/S0378-4371(02)00974-3.

[5] William L. Hamilton. "Graph Representation Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.3 (), pp. 1–159.

[6] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2018. arXiv: 1706.02216 [cs.SI].

[7] Prateek Jain and Purushottam Kar. "Non-convex Optimization for Machine Learning". In: *Foundations and Trends® in Machine Learning* 10.3-4 (2017), pp. 142–336. DOI: 10.1561/2200000058. URL: https://doi.org/10.1561%2F2200000058.

[8] Mark Juergensmeyer. *Terror in the Mind of God: the Global Rise of Religious Violence*. University of California Press, 2017.

[9] W. O. Kermack and A. G. McKendrick. "A Contribution to the Mathematical Theory of Epidemics". In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 115.772 (1927), pp. 700–721. ISSN: 09501207. URL: http://www.jstor.org/stable/94815.

[10] Thomas Kipf et al. *Neural Relational Inference for Interacting Systems*. 2018. arXiv: 1802.04687 [stat.ML].

[11] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG].

[12] Joseph B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50. ISSN: 00029939, 10886826. URL: http://www.jstor.org/stable/2033241.

[13] R.N. Mantegna. "Hierarchical structure in financial markets". In: *The European Physical Journal B* 11.1 (Sept. 1999), pp. 193–197. ISSN: 1434-6028. DOI: 10.1007/s100510050929. URL: http://dx.doi.org/10.1007/s100510050929.

[14] Gautier Marti, Philippe Very, and Philippe Donnat. *Toward a generic representation of random variables for machine learning*. 2015. arXiv: 1506.00976 [cs.LG].

[15] Gautier Marti et al. *A proposal of a methodological framework with experimental guidelines to investigate clustering stability on financial time series*. 2015. arXiv: 1509.05475 [q-fin.ST].

[16] Jonathan Mei and Jose M. F. Moura. "Signal Processing on Graphs: Causal Modeling of ¡italic¿Un¡/italic¿structured Data". In: *IEEE Transactions on Signal Processing* 65.8 (Apr. 2017), pp. 2077–2092. ISSN: 1941-0476. DOI: 10.1109/tsp.2016.2634543. URL: http://dx.doi.org/10.1109/TSP.2016.2634543.

[17] Ece C. Mutlu et al. "Review on Learning and Extracting Graph Features for Link Prediction". In: *Machine Learning and Knowledge Extraction* 2.4 (2020), pp. 672–704. ISSN: 2504-4990. DOI: 10.3390/make2040036. URL: https://www.mdpi.com/2504-4990/2/4/36.

[18] J.-P. Onnela et al. “Dynamics of market correlations: Taxonomy and portfolio analysis”. In: *Physical Review E* 68.5 (Nov. 2003). ISSN: 1095-3787. DOI: 10.1103/physreve.68.056110. URL: http://dx.doi.org/10.1103/PhysRevE.68.056110.

[19] Antonio Ortega et al. “Graph Signal Processing: Overview, Challenges, and Applications”. In: *Proceedings of the IEEE* 106.5 (2018), pp. 808–828. DOI: 10.1109/JPROC.2018.2820126.

[20] Larry Page et al. *The PageRank Citation Ranking: Bringing Order to the Web*. 1998.

[21] Tiago P. Peixoto. “Reconstructing Networks with Unknown and Heterogeneous Errors”. In: *Phys. Rev. X* 8 (4 Oct. 2018), p. 041011. DOI: 10.1103/PhysRevX.8.041011. URL: https://link.aps.org/doi/10.1103/PhysRevX.8.041011.

[22] K. B. Petersen and M. S. Pedersen. *The Matrix Cookbook*. Version 20121115. Nov. 2012. URL: http://www2.compute.dtu.dk/pubdb/pubs/3274-full.html.

[23] “Progress in Information Geometry”. In: *Signals and Communication Technology* (2021). ISSN: 1860-4870. DOI: 10.1007/978-3-030-65459-7. URL: http://dx.doi.org/10.1007/978-3-030-65459-7.

[24] Romeil Sandhu, Tryphon Georgiou, and Allen Tannenbaum. *Market Fragility, Systemic Risk, and Ricci Curvature*. 2015. arXiv: 1505.05182 [q-fin.RM].

[25] John Scott. “Social Network Analysis”. In: *Sociology* 22.1 (1988), pp. 109–127. DOI: 10.1177/0038038588022001007. eprint: https://doi.org/10.1177/003803858802200 1007. URL: https://doi.org/10.1177/0038038588022001007.

[26] Ahmet Sensoy and Benjamin M. Tabak. *Dynamic spanning trees in stock market networks: The case of Asia-Pacific*. Working Papers Series 351. Central Bank of Brazil, Research Department, Mar. 2014. URL: https://ideas.repec.org/p/bcb/wpaper/351.html.

[27] Robyn Speer, Joshua Chin, and Catherine Havasi. *ConceptNet 5.5: An Open Multilingual Graph of General Knowledge*. 2018. arXiv: 1612.03975 [cs.CL].

[28] George Stepaniants, Bingni W. Brunton, and J. Nathan Kutz. “Inferring causal networks of dynamical systems through transient dynamics and perturbation”. In: *Phys. Rev. E* 102 (4 Oct. 2020), p. 042309. DOI: 10.1103/PhysRevE.102.042309. URL: https://link.aps.org/doi/10.1103/PhysRevE.102.042309.

[29] V. STRASSEN. “Gaussian Elimination is not Optimal.” In: *Numerische Mathematik* 13 (1969), pp. 354–356. URL: http://eudml.org/doc/131927.

[30] Gábor J. Székely and Maria L. Rizzo. “Brownian distance covariance”. In: *The Annals of Applied Statistics* 3.4 (Dec. 2009). ISSN: 1932-6157. DOI: 10.1214/09-aoas312. URL: http://dx.doi.org/10.1214/09-AOAS312.

[31] Feng Xia et al. “Graph Learning: A Survey”. In: *IEEE Transactions on Artificial Intelligence* 2.2 (Apr. 2021), pp. 109–127. ISSN: 2691-4581. DOI: 10.1109/tai.2021.3076021. URL: http://dx.doi.org/10.1109/TAI.2021.3076021.

[32] Wayne W. Zachary. “An Information Flow Model for Conflict and Fission in Small Groups”. In: *Journal of Anthropological Research* 33.4 (Dec. 1977), pp. 452–473. ISSN: 2153-3806. DOI: 10.1086/jar.33.4.3629752. URL: http://dx.doi.org/10.1086/jar.33.4.3629752.

[33] Jie Zhou et al. *Graph Neural Networks: A Review of Methods and Applications*. 2021. arXiv: 1812.08434 [cs.LG].