An Introduction to Data-Driven Modal Analysis

Rylie Foster

Reed College, class of 2022

To my parents Ken and Mindi, and to Ilana.
Your support has meant everything.
I love you.

Contents

Acknowledgements

I relied on a great deal of academic and emotional support to complete this thesis, and studied from many learning materials which were not directly cited in this work. My thesis advisors were instrumental in guiding me through the unknowns of self-study and exploratory work. Dr. Eitan Frachtenberg was greatly helpful in turning my scattered interests in matrix analysis into a concrete machine-learning question, and Dr. Alex Quijano was a great help in understanding my DDMA methods of interest, particularly the DMD, and in structuring my theory chapter. I also received significant help from many other members of the math department, and in particular Dr. Jonathan Wells, who frequently helped me understand various proofs and phenomena related to the POD and to analysis. I studied many textbooks, blog posts, and video lecture to understand the material and code of this thesis. In particular, the YouTube channels of Dr. Steve Brunton and Dr. Nathan Kutz were extremely useful for for the theory content of this thesis, and the YouTube channel sentdex was essential in the creation of my convolutional neural nets. I also could not have completed this thesis without the support of my parents, Ken and Mindi, of Ilana, and of my other friends in the Reed College community.

Abstract

Data-driven modal analysis algorithms are techniques in matrix analysis wherein a data matrix is decomposed into sets of modes, scalars, and dynamics which can be used to reconstruct it or to construct a lower-dimension approximation of it. These techniques are extremely useful in many areas of data science and mathematical modeling because they can reduce the dimension of data and extract important patterns and features from data. In this thesis, we compare the performances of three of these techniques, the Discrete Fourier Transform, the Proper Orthogonal Decomposition, and the Dynamic Mode Decomposition in identifying the features distinguishing a series of infrared scans of hurricanes and non-hurricane weather events. The "feature extraction" associated with each algorithm is measured as the ability of a convolutional neural net trained on lower-dimension modal reconstructions of these IR scans to correctly identify other modal reconstructions of different IR scans reconstructed with the same algorithm. We also discuss the underlying theory and relative applicability to various tasks of each algorithm. My hope for this thesis is that it will serve as a functional introductory resource for an interested person with some background in math or computer science to self-study some of the tools and theory of Data-Driven modal analysis.

Chapter 1

Introduction and Fundamentals

1.1 Systems and States

When scientists approach a new system to study, where a system is defined as some collection of objects, such as the cars on a freeway or the particles in the atmosphere, they must often find some means of characterizing its states. That is to say, they must find things which they can observe about the objects which, in turn, allow them to determine the state of the system. Since "system" and "state" are words which become less meaningful the more one tries to define them, it might be useful to consider some examples of the terms in action.

Example 1.1.1. Tic Tac Toe

Consider a Tic Tac Toe board. Let the "system", T, you consider, be made up of the nine tiles of a board, each of which can be a blank, an X, or an O. The "states" of T can be fully characterized by the states of each of its tiles. Since each tile on a Tic Tac Toe board can exist in one of three states, it makes sense to say that T can occupy one of $3^9 = 19683$ states. If, for whatever reason, you were unsatisfied with the presentation of T as a 3-by-3 board, you could instead establish a rule to list the characterizing information

about the state of T in a vector in $\mathbb{Z}_3^n$, as in figure 1.

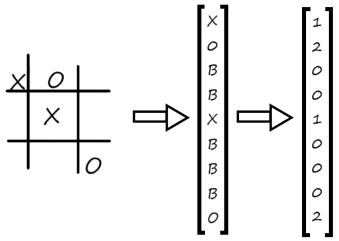

Figure 1.1: A worse way to describe a Tic Tac Toe board, read from left to right, with blanks replaced with 0's, X's with 1's, and O's with 2's.

In the example above, we gave meaning to the terms "system" and "state" in the context of a board game, and then created a convention for storing the state of our board in a vector. Were we interested in pursuing a computational framework for predicting Tic Tac Toe games, or, perhaps more likely, writing programs to play them, we might at this point begin a discussion of Q-learning, neural networks, and related tools, but our focus is instead on modeling physical processes. To this end, let us consider a thoroughly physical system, and carry foreword the effort of modeling it.

Example 1.1.2. A Mass on a Spring

A classic problem in physics is the mass on a spring. A rock of some given mass m is suspended from the ceiling by a spring. The restoring force exerted by the spring is balanced by gravity when the rock is some distance d from the ceiling. This distance d defines the point of equilibrium of the spring, at which no net force acts on the rock. Let x denote the distance between the height of the rock and d in meters, where x is positive when the rock is above the point of equilibrium and negative when the rock is below it. If the spring has stiffness k, then at any given moment, the rock will "feel" a net

force of $-k \cdot x$ Newtons, where, in keeping with our convention, a positive force indicates that the rock is being pulled towards the ceiling, and a negative force indicates that the rock is being pulled towards the floor.

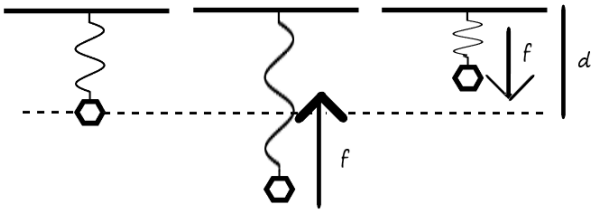

Figure 1.2: Our rock, suspended from a spring at three heights. The arrows indicate force and the dotted line indicates the point of equilibrium.

In this case, it seems reasonable to define our system as our rock, our spring, and the ceiling. Determining what should characterize our system's states, however, requires some more thinking. Assuming we already understand the details and physics of our situation, e.g. the stiffness of the spring, the mass of the rock, and the laws for using that data and the position of the rock to find the force acting on the rock, we can fully describe the state of our system by the position and velocity, v, of the rock. If we graph x on an x-axis and v on the y-axis of a Cartesian graph, we can represent the above images with three plots.

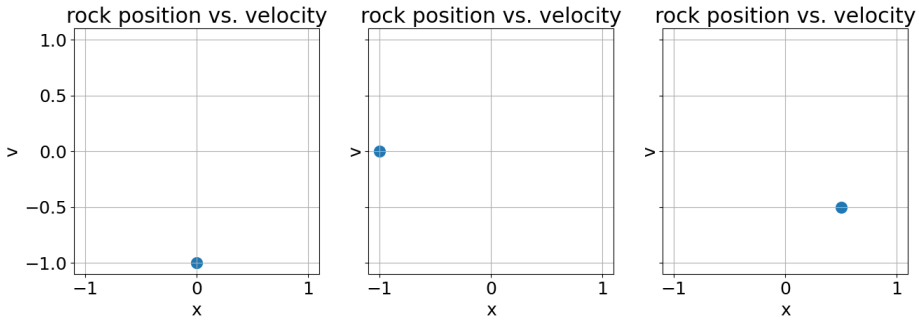

Figure 1.3: The three states of our system from figure 2 plotted on $x - v$ graphs.

With this rule established for encoding system's state in the $x - v$ plot, we can now describe the entire past, present, and future of this system as the *trajectory* of a point in the plane, a curve that we will be able to express parametrically as a function of time. Let us use a little math to find some possible trajectories.

Solving the Spring System

Since we are working in the $x - v$ coordinate system, we want to find and then solve a differential equation of the form $\dot{\mathbf{X}} = \Phi(\mathbf{X})$, where $\mathbf{X} = \left[\begin{smallmatrix} x \\ v \end{smallmatrix}\right]$ and Φ is some set of partial differential equations, which we can write as $\Phi(\mathbf{X}) = \left[\begin{smallmatrix} g(x,v) \\ h(x,v) \end{smallmatrix}\right]$. The first function, $g(x, v)$, is immediate. Since $v = \dot{x}$ by definition, $g(x, v) = v$. For the second function, $h(x, v)$, we look to Newton's second law, which gives us $\dot{v} = \frac{f}{m}$, for an object with mass m experience a force f. Since we've established our spring exerts a force of $-k \cdot x$ Newtons on our rock, we can define αm and write $\dot{v} = h(x, v) = -\alpha x$. With $g(x, v)$ and $h(x, v)$ in hand, we can now write $\dot{\mathbf{X}} = \left[\begin{smallmatrix} v \\ -\alpha x \end{smallmatrix}\right]$. In fact, since $f(x, v)$ and $g(x, v)$ are linear transformation, we can write out our partial differential equations as $\dot{\mathbf{X}} = A\mathbf{X}$, where $A = \left[\begin{smallmatrix} 0 & 1 \\ -\alpha & 0 \end{smallmatrix}\right]$, the matrix that carries out our partial differential equations transforming $\mathbf{X}$ into $\dot{\mathbf{X}}$.

So how has this helped? Why is this interesting? Likely, you the reader are already aware of ways of solving this system without needing to put in so much groundwork, so why bother establishing A as a linear transformation? In order to answer these questions, and to hopefully put an end to this offensively long example, we must take a detour through some crucial concepts from real analysis and linear algebra.

1.2 The Matrix Exponential

The previous example stopped after establishing a system of partial differential equations of the form $\dot{\mathbf{X}} = A\mathbf{X}$, where A is a linear operator. If $\mathbf{X}$ and A were just real numbers, say x and a, this equation might be immediately familiar. In that case, the solution $x(t) = ce^{at}$ might jump to mind, and we could immediately verify it with a derivative, writing $\frac{d}{dt}ce^{at} = \frac{d}{dt}[at]\frac{d}{d(at)}[ce^{at}] = ace^{at}$. Unfortunately, since our A is a matrix, the expression $e^{At}\mathbf{V}$ (with a capital $\mathbf{V}$ now to indicate that the starting value of $\mathbf{X}$ is a vector) is meaningless, at least in the sense that $e^{\frac{a}{b}} = \sqrt[b]{\prod_{i=1}^{a} e}$. We have, however, a more flexible way to calculate any e^x, namely $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$, as given by the Taylor expansion. Substituting our matrix A into the Taylor expansion of e^x is easier to justify, since integer powers of a matrix are perfectly well defined, but identifying the sum $\sum_{k=0}^{\infty} \frac{A^k}{k!}$ with e^A, while linguistically satisfying, does not in fact yield a solution to our system of equations. In the interest of confirming that our system of equations has solution Ae^{At}, we must first verify that $\frac{d}{dt}e^{At}\mathbf{V} = Ae^{At}\mathbf{V}$.

Theorem 1.2.1. Let $A \in \mathbb{R}^r$, $t \in \mathbb{R}$, $e^{At}\mathbf{V} := (\sum_{k=0}^{\infty} \frac{(At)^k}{k!})\mathbf{V}$. Then $\frac{d}{dt}\sum_{k=0}^{\infty} \frac{(At)^k}{k!} = (A\sum_{k=0}^{\infty} \frac{(At)^k}{k!})\mathbf{V}$.

Proof 1.2.1. Let $A \in \mathbb{R}^{n\times n}$, $t \in \mathbb{R}$, $e^{At}\mathbf{V} := (\sum_{k=0}^{\infty} \frac{(At)^k}{k!})\mathbf{V}$. We write

$$\begin{aligned}
\frac{d}{dt}e^{At}\mathbf{V} &:= \frac{d}{dt}((\sum_{k=0}^{\infty} \frac{(At)^k}{k!})\mathbf{V}) \\
&= \sum_{k=1}^{\infty} \frac{1}{k!}\frac{d}{dt}(At)^k\mathbf{V} \\
&= \sum_{k=1}^{\infty} \frac{1}{k!}k(At)^{k-1}\frac{d}{dt}(At)\mathbf{V} \\
&= A\sum_{k=0}^{\infty} \frac{1}{k!}(At)^k\mathbf{V} \\
&= (A\sum_{k=0}^{\infty} \frac{1}{k!}(At)^k)\mathbf{V} \\
&:= Ae^{At}\mathbf{V}.
\end{aligned}$$

The last line of this derivation is valid because any square matrix A commutes with itself.

With this result in hand, $\mathbf{Q} = e^{At}\mathbf{V}$ (with $C \in R^{n\times 1}$) now has every property needed in order to assert that a system of differential equations of the form $\dot{\mathbf{X}} = A\mathbf{X}$ has solution $\mathbf{X} = \mathbf{Q}$. We return to our spring system. We left the spring system example having derived $\mathbf{X} = \left[\begin{smallmatrix} 0 & 1 \\ -\alpha & 0 \end{smallmatrix}\right]\mathbf{X}$. We substitute our matrix into the expression above, yielding $\mathbf{X} = \sum_{k=0}^{\infty}[\frac{1}{k!}\left[\begin{smallmatrix} 0 & t \\ -\alpha t & 0 \end{smallmatrix}\right]^k]\mathbf{Q}$, where $\mathbf{Q}$ is the 2-by-1 matrix given by the initial conditions of our spring system. We can take this solution a little further by supposing that A can be diagonalized, yielding $A = \Phi\Lambda\Phi^{-1}$, with eigenvectors $\{v_1, v_2\}$ given by Φ and eigenvalues $\{\lambda_1, \lambda_2\}$ given by Λ. In that case,

we could write

$$\begin{aligned} A^k &= (\Phi\Lambda\Phi^{-1})^k \\ &= \Phi\Lambda\Phi^{-1}\Phi\Lambda\Phi^{-1}\Phi\Lambda\Phi^{-1}\Phi\Lambda\Phi^{-1} \\ &= \Phi\Lambda^k\Phi^{-1}. \end{aligned}$$

If we substitute this expression into out spring result, we have

$$\begin{aligned} \mathbf{X} &= \sum_{k=0}^{\infty}[\frac{t^k}{k!}\Phi\Lambda^k\Phi^{-1}]\mathbf{Q} \\ &= \Phi\sum_{k=0}^{\infty}[\frac{t^k}{k!}\Lambda^k]\Phi^{-1}\mathbf{Q} \\ &= \Phi e^{\Lambda t}\Phi^{-1}\mathbf{Q}. \end{aligned}$$

Written this way, we can observe that the matrix e^{At} has the same eigenvalues as A, with associated eigenvalues $e^{\lambda_1 t}$ and $e^{\lambda_2 t}$. Thus, if our spring system had initial position and velocity given by $\mathbf{V}$, we could write $\mathbf{V} = av_1 + bv_2$, and thus $\mathbf{X} = e^{At}(av_1 + bv_2) = e^{\lambda_1 t}av_1 + e^{\lambda_2 t}bv_2$. In general, if you have a linear system of differential equations $\mathbf{X} = AX$, and if you can diagonalize A, then for an initial condition $\mathbf{X} = \mathbf{V}$ at $t = 0$ where $\mathbf{V} = \sum_{v_k \in \text{eigs}(A)} a_k v_k$, the trajectory of the system is $\mathbf{X} = \sum_{v_k \in \text{eigs}(A)} e^{\lambda_k t} a_k v_k$. Since the actual eigenvalues of the spring system A are complex, we will conclude our discussion of the spring system here.

1.3 Hard Systems

The previous section defined systems and states and demonstrated a procedure for finding the behavior of systems governed by systems of differential equations. Let's now outline that procedure:

Step 1. Identify the system we are looking at and construct a vector X to encapsulate its states.

Step 2. Use our knowledge of the system to establish a system of differential equations of the form $\frac{d}{dt}X = f(X)$.

Step 3. Solve $\frac{d}{dt}X = f(X)$ for X, yielding the dynamics of our system and giving us complete knowledge of its past and future.

While techniques like matrix exponentiation are obviously tremendously powerful tools, you may rightly suspect that they doesn't do justice to the enormous complexity of the wider world beyond systems of boxes and springs. To illustrate the true measure of this task, and to enumerate the challenges that separate most "toy" systems, like the box-and-spring system, from systems that emerge in nature, let's consider one more example.

Example 1.3.1. The Atmosphere

According to newscientist.com, The Earth's atmosphere contains 1.04×10^{44} molecules of various composition. They are heterogeneous, their interactions are governed by quantum mechanics, a litany of external factors contribute or subtract energy from the system, and particles enter and leave the atmosphere constantly. Let's try out our systems modeling procedure and see how far we can get.

Step 1. In order to describe and predict the behavior of this system, we first must create a vector to represent its state. A perfect description of the atmosphere's state would at the very least need to capture the 3-dimensional

momentum information of each particle in the atmosphere, a task which alone would necessitate a vector $X \in \mathbb{R}^{3.14\times10^{44}}$. This is hopeless, so we must make our first compromise by creating a vector which in some way *approximates* the atmosphere's state. Some traditional relevant statistics for atmospheric modeling are air pressure, temperature, humidity, wind velocity, etc. For the purpose of this thought experiment, we will consider air pressure, temperature, and density as sampled by whatever instruments we manage to place around the world and inferences we can make from satellite data.

Step 2. If we had the luxury of knowing the air pressure, temperature, and density everywhere on the planet at a moment in time, then we could claim that the Navier-Stokes equations govern our system, i.e. $\dot{X} = f(X)$ with f from the 3-dimensional Navier-Stokes. We, however, only have the data which we can measure, a sample of those statistics distributed non-uniformly across the surface of the earth at airports, weather towers, and similar installations. As such, the dynamics f which relate the separate components of X to $\dot{X}$ are deeply unclear.

Step 3. Putting aside the issue of not knowing f, the dynamics of the atmosphere are *certainly* nonlinear, and are quite probably impossible to solve analytically. Even if we had the infinite data required to use the Navier-Stokes f, there is no good solution to those equations in the 3-dimensional case, and not for lack of effort; the Navier-Stokes equations are one of the Millennium Problems laid down at the beginning of the 21st century as a principle challenge of modern mathematics.

The atmospheric example illustrates issues with each step of our systems modeling procedure. Lets enumerate those, as well as some other, issues with that procedure:

Step 1. *Identify the system we are looking at and construct a vector X to encapsulate its states.*

- We can almost never measure the full information X describing a system.
- Even if we could measure X, its almost always going to be far to large to store in any kind of computer.

Step 2. *Use our knowledge of the system to establish a system of differential equations of the form* $\frac{d}{dt}X = f(X)$.

- Since we can't measure X directly, even finding the dynamics of a system with completely understood physics is extremely hard.
- Often times we don't know the dynamics of a system anyways. Can anybody really know an equation governing the behavior of every neuron in the brain? What about every price in the stock market? Would they tell us of they did?

Step 3. *Solve* $\frac{d}{dt}X = f(X)$ *for X, yielding the dynamics of our system and giving us complete knowledge of its past and future.*

- This is usually pretty impossible for any nonlinear system of differential equations.
- Even when solutions to nonlinear differential equations can be found, the nature of those equations leads to high degrees of *sensitivity to initial conditions*, so even the smallest degrees of error in an approximation will, in time, lead to more-or-less unpredictable amounts of error in forecasts of the system. This is why weather predictions made more than around ten days out are relatively useless.

So our dream of using differential equations to predict the future is dead. What *can* we do?

While there is no hope of perfectly modeling the dynamics of most systems, we can still use numerical methods to improve the performance of short-term approximate models and to extract insights about the dynamics of of systems we don't fully understand. We can attempt to handle the issue of an immeasurable or oversized state vector by identifying the measurements of a system which contribute the most information about its behavior. This is called model reduction or reduced-order modeling (ROM). We can pursue an understanding of the underlying dynamics of a system by using insights from physics and statistics to identify relationships between our measurements. This is arguably a statement of the aim of most scientific inquiry, but in this thesis specifically we will discuss how some concepts at the intersection of linear algebra, functional analysis, and statistics can provide insights into the dynamics of a system. Finally (in the scope of this thesis), we can seek the measurements of a system will provide linear approximations of the dynamics of a system. While not necessarily useful for addressing the issue of future-state prediction of nonlinear system, these techniques can still provide useful insights to the significant "modes" which describe the behavior of a system efficiently. These goals and more are the aim of data-driven modal analysis.

In order to discuss some techniques that work towards a better understanding of "hard systems" like the atmosphere, we will need an additional piece language to describe the tools of data-driven modal analysis.

1.4 Functions and Signals

The techniques of Data Driven Modal Analysis handle real-world, discrete information. They allow data scientists and other professionals to extract predictions, models, and even intuition regarding a system from necessarily finite measurements. The techniques of real, complex, and functional analysis, on the other hand, largely

handle infinite, continuous data. Integrals, derivatives, and tools like the Fourier series allow mathematicians to glean insights into the nature of a function that can often be summarized with just a handful of numbers, such as the solutions of a function's derivatives, or the largest Fourier coefficients. It is understandable, then, that computer scientists might pursue a way to bring techniques from real analysis into the world of data. Establishing such a connection requires, first and foremost, a stand-in for real (or complex) valued continuous functions in the world of linear algebra.

Definition 1.4.1. A *signal* is representation of the relationship between two variables. A signal can either be discrete or continuous, and usually takes the form of a Cartesian graph. Continuous signals are written like functions, using parentheses, while discrete signals are written like computer arrays, using brackets (W. Smith 2002).

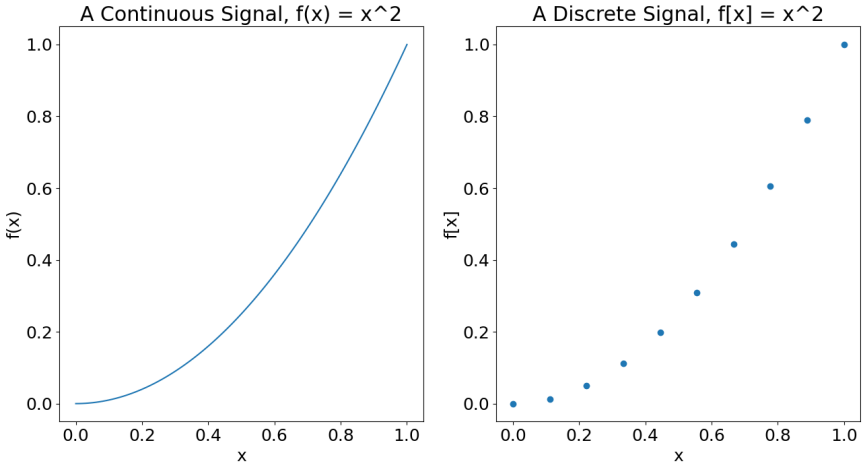

Figure 1.4

The dual-definitions of continuous and discrete signals immediately suggest a bridge between real analysis and linear algebra. If we assume the points comprising some discrete signal, call it $f[x]$ are uniformly spaced along the x-axis, then it seems

natural to write $f[x]$ down as the row vector $f = [f[x_1], f[x_2], \ldots, f[x_n]]$, where the x_i's are sorted in increasing order. With this analogy for a function in place, we are more or less free to pursue analogies for other tools from real analysis in the world of linear algebra, and vice versa .

We are finally ready to discuss the first of our three computational tools in data-driven modal analysis. For each algorithm we discuss, we will extend our intuition for the connections between linear algebra and calculus a little bit, and then leverage that intuition to better understand how linking these seemingly disparate fields of mathematics can grant us new ways to tackle the challenges of complex systems.

Chapter 2

Theory

2.1 The Discrete Fourier Transform

The natural development of the Discrete Fourier Transform (DFT) follows a circuitous intellectual track. It leverages an idea native to linear algebra, namely orthogonality, which was transposed into analysis to give us the Fourier series and Fourier transform, and which then returned to linear algebra (and computer science), to become the DFT. In order to best understand this progression, and by extension the DFT algorithm, it behooves us to refresh our understanding of the concept that underlies each step on this path, namely orthogonality.

In the vaguest, least rigorous terms, orthogonality is a concept that can exist whenever we can describe each element x of a set S as a combination of elements $e_1, e_2, \ldots$ of a subset $B \subset S$, which we might call a basis, and where the contribution of each element e_i of the subset B to x can be quantified, and where no elements of the basis can be created from the others. In similarly unrigorous language, the inner product of two elements of X quantifies the extent to which they are comprised of the same proportions of the same elements of B. In linear algebra, the dot product is generally used to determine when two vectors are orthogonal. It is usually defined $\langle , \rangle : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, $\langle X, Y \rangle = \sum_{k=1}^{n} x_k y_k$. Conceptually, the

dot product represents a measurement of the extent to which two vectors are either parallel or perpendicular, because the basis of a vector space generally represents the different directions in which vectors can point. If the inner product of two different basis vectors is 0, i.e. they are perpendicular, and if the inner product of any basis vector with itself is 1, i.e. B is normalized, then this inner product can be used to determine the exact contribution a_i of each element e_i of the basis to any vector v by $a_i = \langle e_i, v \rangle$. In this case, we can say $v = \sum_{e_i \in B} \langle e_i, v \rangle e_i$.

In order to take our first step towards the DFT, we must transpose this concept of orthogonality, and an inner product to measure it, into the world of real-valued functions. To do so, we will begin with our dot product and *stretch* it until it works for continuous data. Recall that at the end of the previous chapter, we established an analogy between vectors and real-valued functions. This analogy approaches reality as you sample your function at a higher resolution, so let us consider how the dot product of two discrete signals behaves as we increase their resolution.

Consider real-valued functions f and g defined on the interval $[a, b]$, represented by vectors f^n and g^n, constructed by sampling f and g at intervals of $\frac{b-a}{n}$, such that $f^n_k = f(k\frac{b-a}{n})$. In order to prevent our inner product exploding as we increase the resolution of our signal, we tweak the dot product by dividing its output by the dimension of its input vectors. Thus our new dot product, given our discrete signals, is $\langle f^n, g^n \rangle = \frac{1}{n} \sum_{k=1}^{n} f(k\frac{b-a}{n}) g(k\frac{b-a}{n})$. Seeing this, however, we may be reminded of the form of the right-handed Riemann integral, and thus tempted to adjust our inner product again by multiplying by a factor of $b - a$. Thus we arrive at the inner product $\langle f^n, g^n \rangle = \frac{b-a}{n} \sum_{k=1}^{n} f(k\frac{b-a}{n}) g(k\frac{b-a}{n})$, which has limiting behavior $\lim_{n\to\infty} \langle f^n, g^n \rangle = \lim_{n\to\infty} \frac{b-a}{n} \sum_{k=1}^{n} f(k\frac{b-a}{n}) g(k\frac{b-a}{n}) = \int_a^b f(x) g(x) dx$. This is the conventional inner product for real-valued functions defined on the interval $[a, b]$. It has the properties necessary for an inner product, not discussed here, and it serves the same function as the traditional vector space inner product. That is to say, if you can construct an orthogonal basis B of real

valued functions, this inner product will will measure the contribution of each element e of B to any real-valued function f in the span of B. The Fourier series and the discrete Fourier transform are just applications of a particular basis for real-valued functions.

2.1.1 Orthogonal Sinusoids

For the rest of this section, we will consider real-valued functions on the interval $[-\pi, \pi]$ and an inner product $\langle f, g \rangle = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x)g(x)dx$. Having established an inner product, and by extension a concept of orthogonality, we now need an orthonormal basis for real-valued functions. Since this basis is difficult to motivate, we will just introduce it.

Let $C = \{\cos(0), \cos(x), \cos(2x), , \cos(3x)...\}$, $S = \{\sin(x), \sin(2x), , \sin(3x)...\}$, $B = C \cup S$. This is the Fourier basis. As discussed in the preamble to this chapter, a good basis should be orthogonal and must have span equal to the set one hopes to describe with it, in this case real-valued functions f defined on the interval $[-\pi, \pi]$ (the actual restriction on f here is that it is 2π-periodic, but this distinction is not practically meaningful for the purposes of data analysis, and will be addressed shortly). While we will not be verifying the latter condition, we can show here and now that B is orthogonal, and in fact orthonormal (with some assistance from (Brunton & J. Kutz 2019)). Let $f, g \in B$. We must consider six cases given by two sets of conditions, those being that either the period of f period(f) = period(g) or period$(f) \neq$ period(g), and that either $f, g \in C$, $f, g \in S$, or $f \in C$, $g \in S$. Let's take these one at a time:

1. period(f) = period(g), $f, g \in C$. We write

$$\begin{aligned}
\frac{1}{\pi}\int_{-\pi}^{\pi} f(x)g(x)dx &= \frac{1}{\pi}\int_{-\pi}^{\pi} \cos(kx)\cos(kx)dx \\
&= \frac{1}{\pi}\int_{-\pi}^{\pi} \frac{1}{2}(\cos(2kx) + \cos(0))dx \\
&= \frac{1}{2\pi}\int_{-\pi}^{\pi} \cos(2kx)dx + 1 \\
&= \frac{1}{2\pi}\frac{1}{2k}\sin(2kx)|_{-\pi}^{\pi} + 1 \\
&= 1
\end{aligned}$$

2. period(f) = period(g), $f, g \in S$. This is almost identical to (1).

3. period(f) = period(g), $f \in C$, $g \in S$. We write

$$\begin{aligned}
\frac{1}{\pi}\int_{-\pi}^{\pi} f(x)g(x)dx &= \frac{1}{\pi}\int_{-\pi}^{\pi} \cos(kx)\sin(kx)dx \\
&= \frac{1}{\pi}\int_{-\pi}^{\pi} \frac{1}{2}(\sin(2kx) + \sin(0x))dx \\
&= \frac{1}{2\pi}\int_{-\pi}^{\pi} \sin(2kx)dx \\
&= -\frac{1}{4k\pi}\cos(2kx)|_{-\pi}^{\pi} \\
&= 0
\end{aligned}$$

4. period$(f) \neq$ period(g), $f, g \in C$. We write

$$\begin{aligned}\frac{1}{\pi}\int_{-\pi}^{\pi} f(x)g(x)dx &= \frac{1}{\pi}\int_{-\pi}^{\pi} \cos(kx)\cos(lx)dx \\ &= \frac{1}{\pi}\int_{-\pi}^{\pi} \frac{1}{2}(\cos((k+l)x) + \cos((k-l)x))dx \\ &= \frac{1}{2\pi}(\int_{-\pi}^{\pi} \cos((k+l)x) + \int_{-\pi}^{\pi} \cos((k-l)x)dx) \\ &= \frac{1}{2\pi}(\frac{1}{k+l}\sin((k+l)x)|_{-\pi}^{\pi} + \frac{1}{k-l}\sin((k-l)x))|_{-\pi}^{\pi} \\ &= 0\end{aligned}$$

5. period$(f) \neq$ period(g), $f, g \in S$. This is almost identical to (4).

6. period$(f) \neq$ period(g), $f \in C$, $g \in S$. We write

$$\begin{aligned}\frac{1}{\pi}\int_{-\pi}^{\pi} f(x)g(x)dx &= \frac{1}{\pi}\int_{-\pi}^{\pi} \cos(kx)\sin(lx)dx \\ &= \frac{1}{\pi}\int_{-\pi}^{\pi} \frac{1}{2}(\sin((k+l)x) - \sin((k-l)x))dx \\ &= \frac{1}{2\pi}(-\cos((k+l)x)|_{-\pi}^{\pi} + \cos(k-l)x)|_{-\pi}^{\pi}) \\ &= 0.\end{aligned}$$

Thus we see that B is an orthonormal* basis, and may move on to a discussion of the actual Fourier series.

2.1.2 The Fourier Series

The span of the Fourier basis is all real-valued, finitely discontinuous† functions which are periodic with period 2π. This last condition means that for f to be in the

*It's orthonormal with one exception, since we lied a little about $\cos(0) = 1$, where $\frac{1}{\pi}\int_{-\pi}^{\pi} = 2$. We deal with this by dividing the a_0 term in the Fourier series by 2.

†i.e. discontinuous at a finite number of points, at which it takes finite "jumps".

span of B, we must have $f(x+2\pi) = f(x)$. While this may seem like an extreme restriction on f, recall that we are only trying to decompose a function defined on some interval $[a, b]$. If we begin with a function f defined on some $[a, b]$, we can make the following adjustments to f:

1. Let $\hat{f}(x) = f(\frac{2\pi}{b-a}(x - \frac{b-a}{2}))$

2. Let $\hat{f}(x + 2\pi) = \hat{f}(x)$.

Our new function $\hat{f}$ is functionally identical to f, and is wholly within the span of B. It is outside the scope of this thesis to prove that B has the span that has been advertised, so we will now describe the Fourier series and the algorithm used to generate it.

Let f be a real-valued function in the span of B. Our goal is to find the set of coefficients $\{a_1, a_2, a_3, ...\} \cup \{b_1, b_2, b_3, ...\}$ such that $f(x) = \frac{a_0}{2} + \sum_{k=0}^{\infty} a_k \cos(kx) + b_k \sin(kx)$. These are the Fourier coefficients. In order to find them, we must use our inner product to measure the contribution of each basis element to f, which we may rightly suspect is accomplished by $a_k = \langle f(x), \cos(kx) \rangle$, $b_k = \langle f(x), \sin(kx) \rangle$. We can substantiate this intuition by substituting our Fourier series into this inner

product:

$$\begin{aligned}\langle \frac{a_0}{2} + \sum_{l=1}^{\infty} a_l\cos(lx) + b_l\sin(lx), \cos(kx)\rangle &= \frac{1}{\pi}\int_{-\pi}^{\pi}(\sum_{l=0}^{\infty} a_l\cos(lx) + b_l\sin(lx))\cos(kx)dx \\ &= \frac{1}{\pi}\sum_{l=0}^{\infty}\int_{-\pi}^{\pi}(a_l\cos(lx) + b_l\sin(lx))\cos(kx)dx \\ &\text{(Take this step with caution, we are being careless here in not justifying the exchange of an infinite sum and an integral.)} \\ &= \sum_{l=0}^{\infty}(a_l\frac{1}{\pi}\int_{-\pi}^{\pi}\cos(lx)\cos(kx)dx \\ &\quad + b_l\frac{1}{\pi}\int_{-\pi}^{\pi}\sin(lx)\cos(kx)dx) \\ &= a_k.\end{aligned}$$

A nearly identical derivation yields that $\langle \sum_{l=0}^{\infty} a_l\cos(lx) + b_l\sin(lx), \sin(kx)\rangle = b_k$. Thus, if we are given some real-valued f defined on $[a, b]$, we compute Fourier coefficients by the following procedure:

1. Calculate $\hat{f}$ by the procedure outlined above.
2. Let $a_0 = \frac{1}{2}\frac{1}{\pi}\int_{-\pi}^{\pi}\hat{f}(x)dx$.
3. For $\cos(kx) \in C$, $k > 0$, let $a_k = \langle \hat{f}, \cos(kx)\rangle$.
4. For $\sin(kx) \in S$, let $b_k = \langle \hat{f}, \sin(kx)\rangle$.
5. Our Fourier coefficients are $\{a_0\} \cup \{a_1, a_2, a_3, ...\} \cup \{b_1, b_2, b_3, ...\}$ and we have $\hat{f}(x) = \sum_{k=0}^{\infty} a_k\cos(kx) + b_k\sin(kx)$.

This is the Fourier series algorithm. Now that we have it, let's calculate an example.

Example 2.1.1. The Line

Let $f(x) = x$ on $[-\pi, \pi]$. We proceed according to our algorithm

1. Our $\hat{f}$ is just f, with the added condition that $\hat{f}(x + 2\pi) = \hat{f}(x)$.

2. We have

$$\begin{aligned} a_0 &= \frac{1}{2}\frac{1}{\pi}\int_{-\pi}^{\pi} x dx \\ &= \frac{1}{4\pi}x^2|_{-\pi}^{\pi} \\ &= 0. \end{aligned}$$

3. For $\cos(kx) \in C$, $k > 0$, we have

$$\begin{aligned} a_k &= \frac{1}{\pi}\int_{-\pi}^{\pi} x\cos(kx)dx \\ &= \frac{1}{\pi}(\frac{1}{k}x\sin(kx)|_{-\pi}^{\pi} - \frac{1}{k}\int_{-\pi}^{\pi}\sin(kx)dx) \\ &= \frac{1}{k^2\pi}\cos(kx)|_{-\pi}^{\pi} \\ &= 0. \end{aligned}$$

4. For $\sin(kx) \in S$, $k > 0$, we have

$$\begin{aligned} b_k &= \frac{1}{\pi}\int_{-\pi}^{\pi} x\sin(kx)dx \\ &= \frac{1}{\pi}(-\frac{1}{k}x\cos(kx)|_{-\pi}^{\pi} + \frac{1}{k}\int_{-\pi}^{\pi}\cos(kx)dx) \\ &= -\frac{1}{k\pi}x\cos(kx)|_{-\pi}^{\pi} \\ &= (-1)^{k+1}\frac{2}{k}. \end{aligned}$$

5. Our Fourier coefficients are $\{0, 0, 0, \ldots\} \cup \{2, -1, \frac{2}{3}, \ldots\}$ and we have $\hat{f}(x) = \sum_{k=0}^{\infty}(-1)^{k+1}\frac{2}{k}\sin(kx)$.

2.1.3 The Discrete Fourier Transform Algorithm

So far in this chapter, we have developed an intuitive sense of the form and purpose of the inner product, and we have used that intuition, and the conventional vector space inner product, to develop a reasonable inner product on real-valued functions defined on an interval $[a, b]$. This inner product led to the Fourier series algorithm, a procedure for decomposing these real-valued functions into sums of sines and cosines. While there are many important applications of the Fourier series to problems in calculus and analysis, we will now examine a computational tool that leverages Fourier series for data analysis.

Say you're a sound technician and you've been tasked with lowering the pitch of a music track. Since sound is communicated by the vibration of particles in the air, the track is given to you in the form of a discrete signal f representing air pressure vs time as recorded by a microphone. As you may know, a specific pitch corresponds to a specific frequency of vibration in the air, and as such will register as a sinusoid with that frequency on your signal. Due to a feature of the physics of sound called superposition, the pressure signal recorded when two sounds overlap is equal to the sum of the pressure signal recorded when each sound is played in isolation. Thus, if a sound is composed of several pitches played on top of one another, the resulting sound wave would be the sum of the sinusoids corresponding to each pitch. Were our signal f continuous, the way to complete this task would be as follows:

1. Given our continuous signal (function) f on the interval $[t_0, t_1]$, i.e. our time interval, compute $\hat{f}$ as in the Fourier procedure, and then find the Fourier coefficients $\{a_0, a_1, b_1, a_2, b_2, \ldots\}$.

2. Choose some rule by which to scale the coefficients such that "low" frequencies get bigger coefficients and "high" frequencies get smaller coefficients. The meaning of these terms and the extent to which the coefficients must be scaled

is subjective.

This procedure will, with an appropriate choice of re-scaling for your coefficients, achieve the desired result. Your data, however, is discrete, so we need a version of the Fourier transform that applies to discrete data.

In order to generate a continuous Fourier transform, we needed to turn our discrete inner product into a continuous inner product. In order to bring the continuous inner product into the world of discrete data, we must bring the sinusoids into the world of discrete data. This is accomplished by using discrete signals to represent sinusoids. Let us define those signals now for easy reference later

Definition 2.1.1. Discrete Cosine

Let $C_k^n \in \mathbb{R}^n$, $(C_k^n)_j = C_k^n[\frac{2\pi j}{n} - \pi] = \cos(\frac{2\pi kj}{n} - \pi)$ (C_k^n is a vector that is also being treated as a discrete signal, hence the use of bracket notation). Then C_k^n is the k-th discrete cosine in $\mathbb{R}^n$.

Definition 2.1.2. Discrete Sine

Let $S_k^n \in \mathbb{R}^n$, $(S_k^n)_j = S_k^n[\frac{2\pi j}{n} - \pi] = \sin(\frac{2\pi kj}{n} - \pi)$. Then S_k^n is the k-th discrete sine in $\mathbb{R}^n$.

Having these signals will allow us to recreate the Fourier series algorithm, but we first need to demonstrate that these vectors are orthonormal with respect to our inner product, and we need to determine which ones we need to recreate our data. We should note here that, if we succeed in demonstrating that we can make an orthonormal basis out of the discrete sinusoids, we will only need n of them to span $\mathbb{R}^n$, and thus perfectly recreate our data. Let us now demonstrate that our discrete sinusoids are orthogonal. We are now using the inner product $\langle u, v \rangle = \frac{2}{n} \sum_{k=1}^{n} u_n v_n$.

Theorem 2.1.1. Let n be an even integer, $C^n = \{C_0^n, C_1^n, C_2^n, ..., C_n^n\}$, $S^n = \{S_1^n, S_2^n, S_3^n, ..., S_{n-1}^n\}$, then the set $B^n = C^n \cup S^n$ is an orthonormal basis for $\mathbb{R}^n$.

In order to complete this proof, we will rely on a result from complex analysis, that $\cos(x) = \frac{e^{ix}+e^{-ix}}{2}$ and $\sin(x) = \frac{e^{ix}-e^{-ix}}{2i}$. This is discussed in the appendix of this thesis.

Proof 2.1.1. Let n be an even integer, $C^n = \{C_0^n, C_1^n, C_2^n, ..., C_n^n\}$, $S^n = \{S_1^n, S_2^n, S_3^n, ..., S_{n-1}^n\}$. We must check the same cases as in the real-valued function bases:

1. Let $V_1, V_2 \in C^n$, $V_1 = V_2$. We write

$$\begin{aligned}
\langle V_1, V_2 \rangle &= \frac{2}{n}\sum_{k=1}^{n} \cos(\frac{2\pi kj}{n} - \pi)^2 \\
&= \frac{2}{n}\sum_{k=1}^{n} (\frac{e^{i\frac{2\pi kj}{n}-\pi} + e^{-i\frac{2\pi kj}{n}+\pi}}{2})^2 \\
&= \frac{2}{n}\sum_{k=1}^{n} \frac{e^{2i\frac{2\pi kj}{n}-2\pi} + 2e^{i\frac{2\pi kj}{n}-\pi}e^{-i\frac{2\pi kj}{n}+\pi} + e^{-2i\frac{2\pi kj}{n}+2\pi}}{4} \\
&= \frac{2}{n}\frac{\sum_{k=1}^{n} e^{i\frac{4\pi kj}{n}} + \sum_{k=1}^{n} e^{-i\frac{4\pi kj}{n}}}{4} + 1 \\
&= 1.
\end{aligned}$$

   Note that this result relied on the sums $e^{i\frac{4\pi kj}{n}}$ and $e^{-i\frac{4\pi kj}{n}}$ each summing to 0. This is necessarily true because these are sums of n points that are uniformly spaced around a circle in $\mathbb{C}$ centered at 0. We will invoke essentially the same result throughout this proof without comment.

2. Let $V_1, V_2 \in S^n$, $V_1 = V_2$. This is mostly the same as (1).

3. Let $V_1, V_2 \in C^n$, $V_1 \neq V_2$. We write

$$\begin{aligned}
\langle V_1, V_2 \rangle &= \frac{2}{n}\sum_{k=1}^{n} \cos(\frac{2\pi kj}{n} - \pi)\cos(\frac{2\pi kl}{n} - \pi) \\
&= \frac{2}{n}\sum_{k=1}^{n} \frac{(e^{i\frac{2\pi kj}{n}-\pi} + e^{-i\frac{2\pi kj}{n}+\pi})(e^{i\frac{2\pi kl}{n}-\pi} + e^{-i\frac{2\pi kl}{n}+\pi})}{4} \\
&= \frac{2}{n}(\sum_{k=1}^{n} \frac{e^{i\frac{2\pi kj}{n}-\pi}e^{i\frac{2\pi kl}{n}-\pi} + e^{i\frac{2\pi kj}{n}-\pi}e^{-i\frac{2\pi kl}{n}+\pi}}{4} \\
&\qquad + \frac{e^{-i\frac{2\pi kj}{n}+\pi}e^{i\frac{2\pi kl}{n}-\pi} + e^{-i\frac{2\pi kj}{n}+\pi}e^{-i\frac{2\pi kl}{n}+\pi}}{4}) \\
&= \frac{2}{n}(\sum_{k=1}^{n} \frac{e^{i\frac{2\pi k(j+l)}{n}} + e^{-i\frac{2\pi k(j+l)}{n}} + e^{i\frac{2\pi k(j-l)}{n}} + e^{-i\frac{2\pi k(j-l)}{n}}}{4}) \\
&= 0.
\end{aligned}$$

4. Let $V_1, V_2 \in S^n$, $V_1 \neq V_2$. This is mostly the same as (3).
5. Let $V_1 = C_j^n$, $V_2 = S_l^n$, $j = l$. We write

$$\begin{aligned}
\langle V_1, V_2 \rangle &= \frac{2}{n}\sum_{k=1}^{n} \cos(\frac{2kj}{n} - \pi)\sin(\frac{2kj}{n} - \pi) \\
&= \frac{2}{n}\sum_{k=1}^{n} \frac{(e^{i\frac{2kj}{n}-\pi} + e^{-i\frac{2\pi kj}{n}-\pi})(e^{i\frac{2kj}{n}-\pi} - e^{-i\frac{2\pi kj}{n}-\pi})}{4i} \\
&= \frac{1}{n}\sum_{k=1}^{n} \frac{e^{i\frac{4kj}{n}} - e^{-i\frac{4kj}{n}}}{2i} \\
&= 0.
\end{aligned}$$

6. Let $V_1 = C_j^n$, $V_2 = S_l^n$, $j = l$. Exercise.

Thus the set $B^n = C^n \cup S^n = \{\frac{1}{\pi}C_0^n, \frac{1}{\pi}C_1^n, \frac{1}{\pi}C_2^n, ..., \frac{1}{\pi}C_n^n\} \cup \{\frac{1}{\pi}S_1^n, \frac{1}{\pi}S_2^n, \frac{1}{\pi}S_3^n, ..., \frac{1}{\pi}S_{n-1}^n\}$
With these results in hand, we may now analyze f through a discrete Fourier trans-

form:

1. Let $a_0 = \frac{1}{2}\frac{2}{\pi} f \cdot [1, 1, 1, ..., 1]$ Note that f is now being treated as a vector in $\mathbb{R}^n$ where $f_k = f[\frac{2\pi k}{n} - \pi]$. Also note that that we're using the inner product $\langle f, g \rangle = \frac{2\pi}{n} f \cdot g$.
2. For $C_k^n \in C^n$, let $a_k = \langle f, C_k^n \rangle$.
3. For $S_k^n \in S^n$, let $b_k = \langle f, S_k^n \rangle$.
4. Our Fourier coefficients are $\{a_0\} \cup \{a_1, a_2, a_3, ..., a_{\frac{n}{2}+1}\} \cup \{b_0, b_1, b_2, b_3, ..., b_{\frac{n}{2}+1}\}$ and we have $f = \sum_{k=0}^{n-1} a_k C_k^n + b_k S_k^n$.

As with the continuous case, we can now choose a re-balancing of these coefficients to achieve our desired result.

To conclude our discussion of Fourier series, and to highlight the method of analysis we wish to undergo using the other two methods, let's apply the DFT to some two-dimensional data.

Example 2.1.2. An IR scan of a hurricane

Suppose you are given a 160 × 160 IR scan of an image, which can be represented by a 160 × 160 matrix M of, roughly, temperatures, and you want the DFT recreation of the image. You may want this in order to or to identify some important structures in the image, or because you may be able to perform certain computations on the Fourier modes of the image which are only viable for sinusoids. In order to perform this decomposition, we proceed with our DFT algorithm:

1. For each column c_k of M, where $1 \leq k \leq 160$, do the following:
   (a) Let $a_{k0} = \frac{1}{2}\frac{2}{\pi} c_k \cdot [1, 1, 1, ..., 1]$ The vector c_k is now being treated as a vector in $\mathbb{R}^{160}$ where $c_{kl} = c_k[\frac{2\pi l}{160} - \pi]$.
   (b) For $C_l^n \in C^n$, let $a_{kl} = \langle c_k, C_l^n \rangle$.

(c) For $S_l^n \in S^n$, let $b_k = \langle c_k, S_l^n \rangle$.

(d) The Fourier coefficients a_jk, b_jk of c_k are $\{a_{0k}\} \cup \{a_{1k}, a_{2k}, a_{3k}, ..., a_{81k}\} \cup \{b_{0k}, b_{2k}, b_{3k}, ..., b_{81k}\}$ and we have $c_k = \sum_{j=0}^{81} a_{jk} C_j^n + b_{jk} S_j^n$.

Since we have Fourier coefficients $\{a_{0k}, b_{0k}, a_{1k}, b_{1k}, ..., a_{81}, b_{\frac{n}{2}+1}\}$ associated with each column of M, we know the contribution of each C_j^n or S_j^n to each column of M. While it might seem like I just said the same thing twice, associating each sinusoidal C_j^n or S_j^n v in the Fourier basis with a vector of its contributions $a_j = (a_{j1}, a_{j2}, ..., a_{j160})$ or $b_j = (b_{j1}, b_{j2}, ..., b_{j160})$ to each column of M (note that these are the same coefficients a_{jk} above), allows to complete the Fourier decomposition of M, which we do in step 2.

2. Write $M = \sum_{j=0}^{\frac{n}{2}+1} C_j^n a_j^t + S_j^n b_j^t$ or

$$M = \begin{bmatrix} | & | & | & | & & | & | \\ C_1^n & S_1^n & C_2^n & S_2^n & \cdots & C_{81}^n & S_{81}^n \\ | & | & | & | & & | & | \end{bmatrix} \begin{bmatrix} -a_1^t- \\ -b_1^t- \\ -a_2^t- \\ -b_2^t- \\ \cdots \\ -a_{81}^t- \\ -b_{81}^t- \end{bmatrix}.$$

This is the algorithm by which we decompose M into its DFT *modes*, which are the basis elements, and their *dynamics*, which are $\{a_1, b_1, a_2, b_2, ..., a_{81}, b_{81}\}$. We can abbreviate this process, perhaps for image compression, by selecting a number of modes to keep in our recreation, and then replacing the other modes, and their associated dynamics, with columns and rows of zeros, respectively. If we choose to keep only the first r modes of our decomposition,

the resulting approximate decomposition would be

$$M \approx \begin{bmatrix} | & | & | & | & & | & | & | & | & & | \\ C_1^n & S_1^n & C_2^n & S_2^n & \dots & C_{\frac{r}{2}}^n & S_{\frac{r}{2}}^n & 0 & 0 & \dots & 0 \\ | & | & | & | & & | & | & | & | & & | \end{bmatrix} \begin{bmatrix} -a_1^t- \\ -b_1^t- \\ -a_2^t- \\ -b_2^t- \\ \vdots \\ -a_{\frac{r}{2}}^t- \\ -b_{\frac{r}{2}}^t- \\ -0- \\ -0- \\ \dots \\ -0- \\ -0- \end{bmatrix}.$$

In this case, the rank of your approximate matrix recreation would be r. Let us see the result of this procedure applied to an actual 160×160 IR scan of a hurricane:

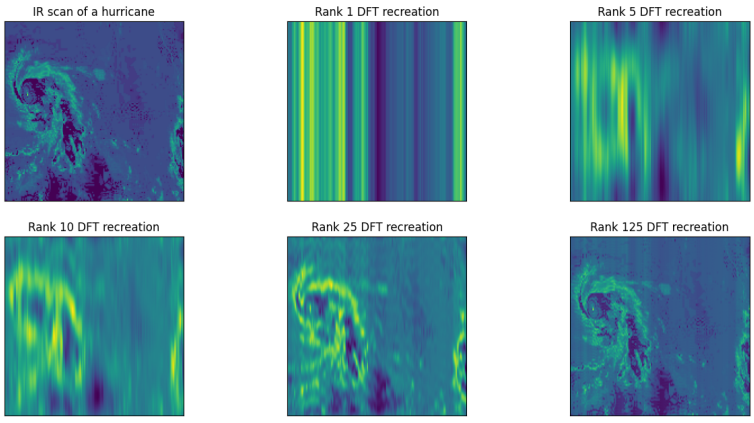

Figure 2.1: A 160×160 crop of IR data from the NOAA Gibbs dataset and five lower-rank recreations produced by my DFT python package for. Note that the maximum-rank recreation of this data would have rank 160.

2.2 Proper Orthogonal Decomposition

The proper orthogonal decomposition (POD) is a technique for decomposing data stored in a matrix into a product of modes and dynamics, with the contribution of each mode to the matrix is knowable without actually projecting our matrix data onto our modes. We encountered this kind of decomposition in the last example of the DFT discussion. In that example, we chose a number r of sinusoids from the discrete Fourier basis with which to decompose our data, specifically the first $\frac{r}{2}+2$ cosines from C^n and $\frac{r}{2}$ sines from S^n, and projecting each column of our data onto them. While there are several great advantages to the DFT decomposition of data, largely related to the relationship sines and cosines have to certain classes of differential equations, it is often valuable to approximate a data matrix as efficiently (i.e. with as few modes as possible). This is the promise of the POD: to determine the ordered, orthonormal basis B in which the sum of the projections of each column c_k of a matrix M along the first element b_1 of B, (by which we mean $\sum_{c_k \in M} \langle c_k, b_1 \rangle^2$) is maximized, and in which the sum of the projections of each column of M onto the first two elements of B (by which we mean $\sum_{c_k \in M} \langle c_k, b_1 \rangle + \sum_{c_k \in M} \langle c_k, b_2 \rangle^2$) is maximized, and so on for the first m modes ($\sum_{l=1}^{m} \sum_{c_k \in M} \langle c_k, b_l \rangle^2$) (Weiss 2019).

2.2.1 The POD Modes

The value of the POD is that the "optimal" modes that it generates can capture information about patterns in your data matrix. Specifically, POD is very good at finding persistent patterns in data and, relatedly, compressing data. The POD excells at these tasks because, unlike the DFT, which always uses the same basis, the POD creates the custom orthonormal basis that best describes the data matrix it is given. Let us try to see how we might discover such a basis.

Let our data matrix be $M \in \mathbb{R}^{m \times n}$. We will begin by seeking out the first basis element $b_1 \in \mathbb{R}^m$, which will satisfy $\langle b_1, b_1 \rangle = 1$ and $\sum_{c_k \in M} \langle b_1, c_k \rangle^2 \geq$

$\sum_{c_k \in M} \langle x, c_k \rangle^2$ for all $x \in \mathbb{R}^n$. An immediate way to approach this problem is via optimization on the unit sphere in $\mathbb{R}^m$. We compute the partial derivative of this sum with respect to a component u_i of u

$$\begin{aligned}\frac{\partial}{\partial u_i} \sum_{k=1}^{n} \langle u, c_k \rangle^2 &= \sum_{k=1}^{n} \frac{\partial}{\partial u_i} \langle u, c_k \rangle^2 \\ &= 2 \sum_{k=1}^{n} \langle u, c_k \rangle \frac{\partial}{\partial u_i} \langle u, c_k \rangle \\ &= 2 \sum_{k=1}^{n} \langle u, c_k \rangle c_{k_i},\end{aligned}$$

which yields the gradient

$$\begin{aligned}\nabla \sum_{k=1}^{n} \langle u, c_k \rangle^2 &= 2(\sum_{k=1}^{n} \langle u, c_k \rangle c_{k_1}, \sum_{k=1}^{n} \langle u, c_k \rangle c_{k_2}, \dots \sum_{k=1}^{n} \langle u, c_k \rangle c_{k_m}) \\ &= 2(\sum_{k=1}^{n} M_{1k} \langle u, c_k \rangle, \sum_{k=1}^{n} M_{2k} \langle u, c_k \rangle, \dots, \sum_{k=1}^{n} M_{mk} \langle u, c_k \rangle)\rangle \\ &= 2(\sum_{k=1}^{n} M_{1k} \sum_{l=1}^{m} u_l c_{kl}, \sum_{k=1}^{n} M_{2k} \sum_{l=1}^{m} u_l c_{kl}, \dots, \sum_{k=1}^{n} M_{mk} \sum_{l=1}^{m} u_l c_{kl}) \\ &= 2(\sum_{l=1}^{m} \sum_{k=1}^{n} M_{1k} M_{lk} u_l, \sum_{l=1}^{m} \sum_{k=1}^{n} M_{2k} M_{lk} u_l, \dots, \sum_{l=1}^{m} \sum_{k=1}^{n} M_{mk} M_{lk} u_l) \\ &= 2(\sum_{l=1}^{m} (MM^t)_{1l} u_l, \sum_{l=1}^{m} (MM^t)_{2l} u_l, \dots, \sum_{l=1}^{m} (MM^t)_{ml} u_l)\rangle \\ &= 2(((MM^t)u)_1, ((MM^t)u)_2, \dots, ((MM^t)u)_m) \\ &= 2(MM^t)u.\end{aligned}$$

Since we're performing our optimization task on the unit sphere in $\mathbb{R}^n$, and since we're assuming that $\langle u, u \rangle = 1$, we now must solve $\langle u_\perp, \nabla \sum_{k=1}^{n} \langle u, c_k \rangle^2 \rangle = 0$. This is equivalent to the condition that $(MM^t)u$ is parallel to u, necessitating that $(MM^t)u = \lambda u$ or that u is an eigenvector of MM^t.

Since MM^t is an $m \times m$ positive-definite matrix, it must have m distinct eigenvectors $\{v_1, v_2, ..., v_m\}$ with positive eigenvalues. Let us assume that the vectors v_i are sorted in descending order, such that for $j < m$ we have $\sum_{k=1}^{n} \langle v_j, c_k \rangle^2 \geq \sum_{k=1}^{n} \langle v_{j+1}, c_k \rangle^2$. In that case, we assert the following:

Theorem 2.2.1. Let $M \in \mathbb{R}^{m \times n}$, with $\{v_1, v_2, ..., v_m\}$ the sorted, normalized eigenvectors of MM^t. Also let c_k be the kth column of M. Then if $r \leq n$, $\sum_{j=1}^{r} (\sum_{k=1}^{n} \langle c_k, v_j \rangle^2) \geq \sum_{j=1}^{r} (\sum_{k=1}^{n} \langle c_k, u_j \rangle^2)$ for any orthonormal $\{u_1, u_2, ..., u_r\} \subset \mathbb{R}^m$.

This theorem is directly connected to a result called the Eckart-Young theorem (Brunton & J. Kutz 2019), which will not be proved in this thesis.

Thus we have the POD basis $\{v_1, v_2, ..., v_m\}$. It's also worth noting the following:

$$\begin{aligned} \sum_{k=1}^{n} \langle c_k, v_j \rangle^2 &= \sum_{k=1}^{n} (M^t v_j)_k^2 \\ &= (M^t v_j)^t M^t v_j \\ &= v_j^t M M^t v_j \\ &= \lambda_j, \end{aligned}$$

which immediately provides an ordering of the v_j's, wherein we simply sort them in decreasing order of their eigenvalues.

2.2.2 The POD Algorithm

Now that we have the sorted POD modes $\{v_1, v_2, ..., v_m\}$ for our data matrix M, the task of generating dynamics for with each mode is fairly straightforward; for each mode v_j, the associated dynamic a_j is the vector $(\langle v_j, c_1 \rangle, \langle v_j, c_2 \rangle, ..., \langle v_j, c_m \rangle)$, where c_k is the kth column of M as usual. With this final piece in place, let's outline the POD algorithm.

1. Given our data matrix M, calculate MM^t.

2. Find the eigenvalues $\{v_1, v_2, ..., v_m\}$ of MM^t and their associated eigenvectors $\{\lambda_1, \lambda_2, ..., \lambda_m\}$.

3. Sort $\{v_1, v_2, ..., v_m\}$ in order of decreasing eigenvalue.

4. Generate the dynamics $\{a_1, a_2, ..., a_m\}$ associated with $\{v_{1,2}, ..., v_m\}$ via $a_{jk} = \langle v_j, c_k \rangle$, where c_k is the kth column of M.

Now that we have this algorithm, we can consider applying it to some real data.

Example 2.2.1. An IR scan of a hurricane

Suppose you are given a 160×160 IR scan of an image, which can be represented by a 160×160 matrix M of, roughly, temperatures, and you want a lower-dimension recreation of the image. You may want this in order to reduce noise in the image, or to identify some important structure in it. In order to perform this decomposition, we proceed with our POD algorithm:

1. Interpreting out IR data as a matrix M, calculate MM^t.
2. Find the eigenvalues $\{v_1, v_2, ..., v_m\}$ of MM^t and their associated eigenvectors $\{\lambda_1, \lambda_2, ..., \lambda_m\}$.
3. Sort $\{v_1, v_2, ..., v_m\}$ in order of decreasing eigenvalue.
4. Generate the dynamics $\{a_1, a_2, ..., a_m\}$ associated with $\{v_{1,2}, ..., v_m\}$ via $a_{jk} = \langle v_j, c_k \rangle$, where c_k is the kth column of M.
   One should note, here, that the modes v_j correspond to columns of pixels, and that the products $v_j a_j^t$ are each rank-1 matrices which can be interpreted as new 160×160 images.
5. In order to create the optimal rank-r (r-dimensional) recreation of your image, interpret the sum $\sum_{k=1}^{r} v_k a_k^t$ as a 160×160 pixel image.

Let us see the result of this procedure applied (again) to a 160×160 IR scan of a hurricane:

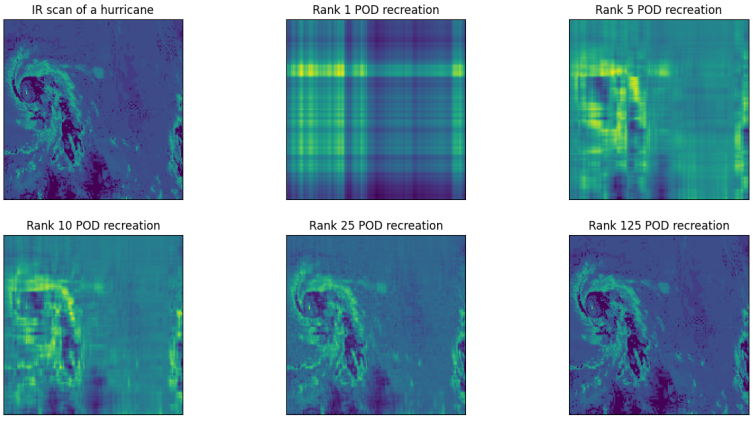

Figure 2.2: A 160 × 160 crop of IR data from the NOAA Gibbs dataset and five lower-rank recreations produced by the Von Karman Institute's modulo python package for POD (among other things). Note that the maximum-rank recreation of this data would have rank 160.

Let's quickly demonstrate a technique for video data.

Example 2.2.2. Video data

Consider a scenario in which you're handed a black-and-white video composed of a series of data matrices $\{T_1, T_2, \ldots T_m\}$ with values corresponding to the brightness of each pixel in the video. We want to identify coherent patterns in the video, which should be interpretable as an ordered set of videos, each containing a pattern from the original video. To do so, we must first create the typical data matrix M from our frames. We "flatten" each data matrix T_k into a column vector c_k, either by stacking all of the columns of T_k on top of each other, or by any other algorithm. Then we create $M = \begin{bmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_m \\ | & | & & | \end{bmatrix}$ and perform the POD on M as with any other data matrix, yielding $M = U\Sigma V^t$. We then perform our flattening algorithm in reverse on the columns u_k of U, creating new video frames T_k'. We can now create the rank-r POD

reconstruction of our video with $c_{q\text{ rank-r}} = \sum_{k=1}^{r} T_k' V_{kq}^t$. Finally, we transform c_k into a matrix with the dimension of the T_k's.

2.3 Dynamic Mode Decomposition

At the end of the last chapter, we introduced a series of obstacles preventing us from analyzing organically arising systems in the "natural" way, i.e. by discovering and then solving a system of differential equations governing the system. These roadblocks included too-large state vectors, unsolvable dynamics, and unknown dynamics. The DFT is, in part, a tool for widening the range of solvable dynamics. The POD is a great tool for reducing the dimension of complex data, and for identifying patterns in a dataset. The Dynamic Mode Decomposition (DMD) helps address the issue of unknown dynamics. Specifically, if we have a $m \times n$ data matrix M with columns $c_1, c_2, ..., c_n$, then DMD will give us the ideal (in the inner product sense) $n \times n$ matrix $\tilde{A}$ such that $c_{k+1} \approx \tilde{A} c_k$. In other words, the DMD provides the best-fitting set of linear dynamics for approximating the evolution of a nonlinear system.

2.3.1 Basic Structure of the DMD

In order to understand the basic mechanism of the DMD we will first need a particular property of matrix multiplication.

Theorem 2.3.1. Let $A \in \mathbb{R}^{m \times n}$, $M \in \mathbb{R}^{n \times p}$. Let M has columns $c_1, c_2, ..., c_p$, such that we may write $M = \begin{bmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_p \\ | & | & & | \end{bmatrix}$. Then $AM = \begin{bmatrix} | & | & & | \\ Ac_1 & Ac_2 & \cdots & Ac_p \\ | & | & & | \end{bmatrix}$.

Proof 2.3.1. Exercise.

Given this result, let us now assume we have a matrix A which perfectly describes the dynamics of M, such that $Ac_k = c_{k+1}$. Then $AM = \begin{bmatrix} | & | & & | \\ Ac_1 & Ac_2 & \cdots & Ac_n \\ | & | & & | \end{bmatrix} =$

$\begin{bmatrix} | & | & & | \\ c_2 & c_3 & \cdots & Ac_n \\ | & | & & | \end{bmatrix}$. Note that we have refrained from writing the last element of AM as c_{n+1}, since that doesn't exist. This is the key to the DMD. The basic structure of the DMD, leveraging this fact, is thus:

1. Split your data matrix M into two matrices X and X', where $X = \begin{bmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_{n-1} \\ | & | & & | \end{bmatrix}$ and $X' = \begin{bmatrix} | & | & & | \\ c_2 & c_2 & \cdots & c_n \\ | & | & & | \end{bmatrix}$.
2. Find the matrix A that best solves $X' = AX$ by minimizing $\langle X' - AX, X' - AX \rangle$.

The ideal output of the DMD algorithm would be the matrix $A = X'X^{-1}$, but there are two problems. Firstly, since X is almost certain to not have full rank, X will most likely not be invertable. Secondly, X is probably, huge, so even if it were invertable, the inverse would most likely be incredibly computationally difficult to calculate. Overcoming these obstacles will require the Moore-Penrose pseudoinverse, a tool we have not yet seen, and the POD, which we have.

2.3.2 The Moore-Penrose Matrix Inverse

The Moore-Penrose pseudoinverse is the next-best-thing to an inverse for a matrix A when A is uninvertible, and is the inverse when A is invertible. In order for a matrix $A^\dagger$ to be the pseudoinverse of A, $A^\dagger$ must satisfy the four Penrose conditions (Baksalary & Trenkler 2021):

1. $AA^\dagger A = A$
2. $A^\dagger AA^\dagger = A^\dagger$
3. $AA^\dagger = (AA^\dagger)^t$
4. $A^\dagger A = (A^\dagger A)^t$.

If A is $m \times n$, a matrix $A^\dagger$ satisfying these conditions is guaranteed to minimize $\langle x - A^\dagger Ax, x - A^\dagger Ax \rangle$ for $x \in \mathbb{R}^m$. In order to calculate the pseudoinverse of A, we will need to a tool called the singular value decomposition (SVD). Fortunately, this is almost identical to the POD. The SVD matrices of a matrix M are U, Λ, and V, where $M = U\Lambda V^t$, and where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthonormal and Λ is diagonal and positive definite (Weiss 2019). We calculate the SVD of M with a familiar algorithm (Weiss 2019):

1. Let $C_1 = MM^*$.
2. Let $C_2 = M^*M$.
3. Compute the diagonalization of C_1, $C_1 = \Phi_1 \Lambda_1 \Phi_1^*$.
4. Compute the diagonalization of C_2, $C_2 = \Phi_2 \Lambda_2 \Phi_2^*$.
   Since there exists some $U\Lambda V^* = M$, note that $C_1 = \Phi_1 \Lambda_1 \Phi_1^* = U\Lambda V^t V \Lambda U^t = U\Lambda^2 U^*$ and $C_2 = \Phi_2 \Lambda_1 \Phi_2^* = V\Lambda U^* U \Lambda V^* = \Phi_2 \Lambda_1 \Phi_2^* = V\Lambda^2 V^*$.
5. $U = \Phi_1$, $\Lambda = \Lambda_1^{\frac{1}{2}}$, $V = \Phi_2$, $M = U\Lambda V^*$.

Note that we have switched from the transpose to the conjugate-transpose. This is a necessary shift for this chapter, since we will be forced to diagonalize matrices with potentially complex eigenvalues later. With this result in hand, we now assert that the pseudoinverse $A^\dagger$ of A is $A^\dagger = V\Lambda^\dagger U^t$ (Holbrook 2019). We still need to calculate $\Lambda^\dagger$. Since Λ is diagonal, we calculate $\Lambda^\dagger$ by replacing every nonzero element of Λ^t with its inverse. With this technique in hand, we are now ready to tackle the next iteration of the DMD algorithm.

2.3.3 The Exact DMD and the Rank-r DMD

Let's revisit our DMD algorithm:

1. Split your data matrix M into two matrices X and X', where $X = \begin{bmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_{n-1} \\ | & | & & | \end{bmatrix}$ and $X' = \begin{bmatrix} | & | & & | \\ c_2 & c_2 & \cdots & c_n \\ | & | & & | \end{bmatrix}$.
2. Find the matrix A that best solves $X' = AX$ by minimizing $\langle X' - AX, X' - AX \rangle$.

Now, using the Moore-Penrose inverse, we can fill in the blank left in step two, leading us to a more complete algorithm:

1. Split your data matrix M into two matrices X and X', where $X = \begin{bmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_{n-1} \\ | & | & & | \end{bmatrix}$ and $X' = \begin{bmatrix} | & | & & | \\ c_2 & c_2 & \cdots & c_n \\ | & | & & | \end{bmatrix}$.
2. Calculate the SVD $X = U\Lambda V^t$.
3. Calculate $A = X'X^\dagger = X'V\Lambda^\dagger U^\dagger$.

This algorithm is the exact DMD (Kutz 2018). The exact DMD is a complete algorithm, you could transcribe it into code and you would receive the best-fitting linear transformation A to describe the dynamics of your system. The issues, then, are that we are likely forced to make very expensive calculations to calculate A, since the size of each column of M, i.e. the dimension of our data vectors, is likely huge, and that A is a $m \times m$ matrix with, potentially, rank m. This second point may not sound like a problem, and indeed it might not be, but if we aim to produce a reduced-order model of our system, we really want dynamics with rank $r << m$.

In the rank-r DMD $\tilde{A}_r$ of M is the projection of M onto the first r SVD modes of X. We calculate $\tilde{A}_r$ by first replacing X with its rank-r truncation $U_r\Lambda_r V_r^*$, where U_r is the $m \times r$ matrix obtained by throwing out all the first r columns of U, Λ_r is the $r \times m$ matrix obtained by throwing out all but the first r rows of Λ, and V_r is the $m \times r$ matrix obtained by throwing out all but the first r columns of V. The matrix $\tilde{A}_r$ is the $r \times r$ matrix obtained by projecting the rank- version of A, $A_r = X'V_r\Lambda_r U_r^*$, onto

the first r POD modes of X via $\tilde{A}_r = U_r^* A_r U_r = U_r^* X' V_r \Lambda_r U_r^* U_r = U_r^* X' V_r \Lambda_r$ Note that $\tilde{A}_r$ is an $r \times r$ matrix. We will deal with that in a moment. For now, however, we take advantage of the small size of $\tilde{A}_r$ to efficiently find its eigenvectors $\{v_1, v_2, ..., v_q\}$ and their associated eigenvalues $\{\lambda_1, \lambda_2, ..., \lambda_q\}$, where $q \leq r$. If we embed the eigenvectors of $\tilde{A}_r$ as the columns of a $r \times q$ matrix W, and embed the eigenvalues as the diagonal entries of a $q \times q$ diagonal matrix Σ, then we may state the solution to the eigenvalue problem as $\tilde{A}_r W = W\Sigma$.

In order to use this information to create the $m \times m$ rank-r approximation of A, first observe the following:

$$\begin{aligned} \tilde{A}_r &= W\Sigma W^* \\ \tilde{U}_r^* A \tilde{U}_r &= W\Sigma W^\dagger \\ A &\approx \tilde{U}_r W \Sigma W^\dagger \tilde{U}_r^* \end{aligned}$$

Thus A has eigenvalues Σ and approximate eigenvectors $\tilde{U}_r W$. These are the projected DMD modes (J. Kutz, Brunton, Brunton & L. Proctor 2016). Once we know the approximate eigenvectors and eigenvalues of A, we can reconstruct our data matrix M using the technique from the end of chapter 2.1. We consider the left-most column c_1 of M to be the "initial state" of our system, and express c_1 in terms of the eigenvectors of A via $c_1 \approx \sum_{k=1}^{r} a_k v_k$. If we assume that the columns of our matrix c_k represent states of our system sampled at time intervals t_{k-1}, separated uniformly by some $\Delta t = t_{k+1} - t_k$, then this allows us to write that the state of a system governed by linear dynamics A with initial state c_1 is $\mathbf{X}(t) = \sum_{k=1}^{r} a_k e^{\frac{\ln(\lambda_k)}{\Delta t} t} v_k$ (J. Kutz et al. 2016). Note that we are using the natural logarithms of our eigenvectors because our matrix A is not in fact the time-derivative of our system, but instead the matrix that advances our system in time by the fixed time-unit Δt. This, finally, allows us to assert that the rank-r reconstruction of a column

c_k of M is $\mathbf{X}((k-1)\Delta t) = \sum_{k=1}^{r} a_k e^{\frac{\ln(\lambda_k)}{\Delta t}(k-1)\Delta t} v_k = \sum_{k=1}^{r} a_k e^{\ln(\lambda_k)(k-1)} v_k$. This is reassuring, as it is equal to $A^{k-1}c_1 = \sum_{k=1}^{r} a_k \lambda_k^{(}k-1)v_k$. Lets apply this process to a couple of examples.

Example 2.3.1. An IR scan of a hurricane

One final time, suppose you are given a 160×160 IR scan of an image, which can be represented by a 160×160 matrix M of temperatures, and you want a lower-dimension recreation of the image. The DMD is an odd choice for this task, as we will see, but the DMD algorithm would be performed as follows:

1. Interpreting out IR data as a matrix M, calculate the eigenvectors $\{v_1, v_2, ..., v_r\}$ and eigenvalues $\{\lambda_1, \lambda_2, ..., \lambda_r\}$ of the rank-r dynamics matrix A of M, as calculated by the DMD algorithm.

2. Decompose the leftmost column c_1 of M into the eigenvectors of A to get $c_1 = \sum_{k=1}^{r} a_k v_k$.

3. The reconstructed matrix M' is the 160×160 matrix with each column c_k of M' given by $\sum_{k=1}^{r} a_k e^{\ln(\lambda_k)(k-1)} v_k$.

Let us see the result of this procedure applied to a 160×160 IR scan of a hurricane:

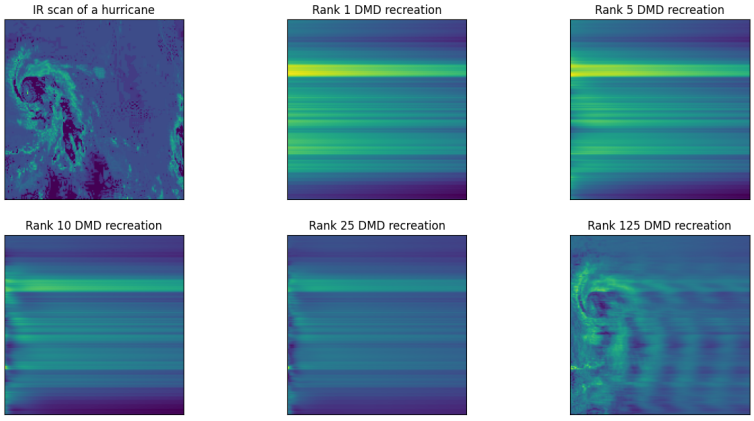

Figure 2.3: A 160 × 160 crop of IR data from the NOAA Gibbs dataset and five lower-rank recreations produced by the pydmd python package for DMD. Note that the maximum-rank recreation of this data would have rank 160.

Let's look at how we might apply DMD to video data, which is much closer to its general intended use case.

Example 2.3.2. Video data

Consider again a scenario in which you're handed a black-and-white video composed of a series of data matrices $\{T_1, T_2, \ldots T_m\}$ with values corresponding to the brightness of each pixel in the video. What follows will be, structurally, very similar to the POD analysis. We want to determine the linear operator which best describes the time evolution of that video. To do so, we must again create the typical data matrix M from our frames. We flatten each data matrix T_k into a column vector c_k. Then we create $M = \begin{bmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_m \\ | & | & & | \end{bmatrix}$ and perform the DMD on M as with any other data matrix, yielding a dynamics matrix A associated with dynamics $\mathbf{X}(t) = \sum_{k=1}^{r} a_k e^{\frac{\ln(\lambda_k)}{\Delta t} t} v_k$. In order to view either a reconstructed frame of our video or a predicted frame of our video,

we perform our flattening algorithm in reverse on the vector $\mathbf{X}((k-1)\Delta t) = \sum_{k=1}^{r} a_k e^{\frac{\ln(\lambda_k)}{\Delta t}(k-1)\Delta t} v_k$, creating the new video frame T_k'. It is important to note that we can now not only reconstruct our original video within the span of the eigenvectors of A, but can in fact make predictions beyond the length of our video, or before its beginning.

Chapter 3

Procedure

Our hope for these model reduction algorithms was to see whether they could identify some of the inherent structure of hurricanes captured in our IR data. In order to determine whether the DFT, POD, and DMD can each in fact find some hurricane-indicating property in an image, we first trained a convolutional neural network on 166 160 × 160 processed IR scans of hurricanes and non-hurricane weather events using k-fold cross validation, recording the mean accuracy of our training model. We then replaced each image with each of its recreations from each algorithm, iterating over the rank of the recreation, and repeated the k-fold cross validation training process on the new dataset.

3.0.1 Data collection and initial processing

The images used in this thesis were taken from the NOAA Gibbs project (see 7.1 - Data). The start and end dates of 18 Atlantic hurricanes which developed between 2000 and 2002 were fed into an MS-DOS script, which used them to make pull-requests of IR scans of the western hemisphere of the earth from within that interval. The satellite taking these scans (GOES-12) collected data every three hours, so, for example, requesting a one week interval would yield 56 IR images of the earth. These sets of images were then compared to tropical cyclone reports

in order to identify regions containing hurricanes. Five hurricane and five non-hurricane regions were selected from each time interval, and then another script was used to apply a 160×160 crop to those regions. Finally, each cropped image was converted into a two-dimensional numpy ndarray of normalized temperature values (corresponding to each pixel) using a NOAA-provided temperature scale of colors for their IR scans. It is worth noting that this is a relatively small dataset, and that better results would likely be achieved with more IR scans.

3.0.2 Neural net and training model

For the classification task central to this task, a convolutional neural network (CNN) was written using Torch. It was trained using k-fold validation on eight multi-threaded folds for 800 epochs of training. The training was conducted using Adam optimization with a learning rate of 0.0001. While an explanation of the mechanics of convolutional neural nets is out of the scope of this thesis, a great overview can be found here:
`https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`. Additionally, a great summary of k-fold validation is available here:
`https://machinelearningmastery.com/k-fold-cross-validation/`. The CNN used in this thesis had the following structure:

1. A 3×3 convolutional layer.
2. A 2×2 pooling layer.
3. A 79×79 (6241) $\rightarrow$ 64 fully-connected linear layer.
4. A $64 \rightarrow 8$ fully-connected linear layer.
5. An $8 \rightarrow 2$ fully-connected linear layer.
6. A softmax filter for output.

With the CNN structure in place, the multi-threaded CNN training procedure was as follows:

1. Import and shuffle a reconstructed dataset of IR scans.
2. Create an eight-entry array of zeros.
3. Separate the dataset into eight equal segments.
4. Open eight threads. Each thread is passed the same array from step 2 an a unique index k between 0 and 7. Set segment k of the dataset aside for validation, and reserve the rest for training. In each thread:
   (a) Create a new CNN with the structure given above.
   (b) Train the CNN on the training dataset for 800 epochs.
   (c) Record the accuracy of the CNN applied to the validation set, and record the output at index k of the array of zeros.
5. The accuracy mean and variance associated the particular dataset reconstruction are calculated from the eight-value array after all of the threads are joined.

The code for this neural net is available at my GitHub repository: `https://github.com/ryliefoster/thesis`.

3.1 The Control

After the IR scans were processed as described in 3.0.1, a CNN was trained was trained on them, and the mean and variance of the set of eight accuracies returned by each training fold was recorded.

3.2 The Discrete Fourier Transform

For the DFT reconstruction of IR scans, A DFT class was implemented using numpy arrays. The DFT class was used to reconstruct the dataset at every eighth rank starting from zero and ending at 152. The DFT class implementation is attached below.

```
import numpy as np


class DFT:
    def __init__(self, data_dimension, dft_rank):

        # A DFT instance stores the dimension of
        # ndarrays accepted by DFT.fit()

        if type(data_dimension) == int:
            data_dimension = (data_dimension, 1)
        self.data_dimension = data_dimension

        # A DFT instance stores the rank of the
        # recreations produced by DFT.fit()

        self.dft_rank = dft_rank

        # A DFT instance stores lambdas which
        # generate the cosine and sine modes
        # for its specific DFT rank

        ran = np.arange(-np.pi, np.pi, 2 * np.pi / data_dimension[0])
        self.dft_cos = lambda k: np.cos(k * ran)
        self.dft_sin = lambda k: np.sin(k * ran)

    def cos_coefficients(self, data):

```

```
29          # Generates the coefficients
30          # associated with each cosine DFT
31          # mode C^n_k by projecting the data
32          # matrix data onto the vector
33          # produced by self.cos(K)
34
35          n = self.dft_rank // 2
36          out = np.ndarray(n)
37          for k in range(n):
38              out[k] = np.matmul(self.dft_cos(k), data) * (2 / len(data))
39              if k == 0:
40                  out[k] /= 2
41          return out
42
43      def sin_coefficients(self, data):
44
45          # Generates the coefficients
46          # associated with each sine DFT
47          # mode S^n_k by projecting the data
48          # matrix data onto the vector
49          # produced by self.sin(K)
50
51          n = self.dft_rank // 2
52          out = np.ndarray(n)
53          for k in range(n):
54              out[k] = np.matmul(self.dft_sin(k), data) * (2 / len(data))
55              if k == 0:
56                  out[k] /= 2
57          return out
58
59      def modes_scalars_dynamics(self, data):
60
61          # Produces the mode, sccalar,
62          # and dynamics matrices
63          # obtained by applying the DFT
```

```
64          # algorithm to data
65
66          # Setting up the ndarrays
67
68          m = np.zeros((self.data_dimension[0], self.dft_rank))
69          s = np.zeros(self.dft_rank)
70          d = np.zeros((self.data_dimension[1], self.dft_rank))
71
72          # Iterates through the set of
73          # the first self.dft_rank
74          # DFT basis vectors B^n of
75          # R^n to populate the dynamics
76          # matrix d
77
78          for k in range(self.dft_rank):
79              if not self.data_dimension == np.shape(data):
80                  data = np.reshape(data, self.data_dimension)
81
82              # Alternating between cosine
83              # and sine modes
84
85              if k % 2 == 0:
86
87                  # projecting the dataset onto
88                  # a cosine mode
89
90                  m[:, k] += self.dft_cos(k // 2) / np.pi
91                  for l, col in enumerate(data.T):
92                      d[l, k] += np.matmul(self.dft_cos(k // 2), col) * (2
    * np.pi / len(data))
93                  s[k] = np.matmul(d[:, k], d[:, k].T)
94              elif k % 2 == 1:
95
96                  # projecting the dataset onto
97                  # a sine mode
```

```
98
99                  m[:, k] += self.dft_sin(k // 2) / np.pi
100                 for l, col in enumerate(data.T):
101                     d[l, k] += np.matmul(self.dft_sin(k // 2), col) * (2
    * np.pi / len(data))
102                 s[k] = np.matmul(d[:, k], d[:, k].T)
103         return m, s, d
104
105     def fit(self, data):
106
107         # Generates the DFT factorization
108         # of data and returns the
109         # rank-self.dft_rank recreation
110
111         m, s, d = self.modes_scalars_dynamics(data)
112         out = np.matmul(m, d.T)
113         return out
114
115     # Some fun visualizations of the
116     # DFT modes
117
118     def cos_matrix(self):
119         out = np.ndarray((self.data_dimension, self.data_dimension))
120         for k in range(self.data_dimension):
121             out[k] = self.dft_cos(k)
122         return out
123
124     def sin_matrix(self):
125         out = np.ndarray((self.data_dimension, self.data_dimension))
126         for k in range(self.data_dimension):
127             out[k] = self.dft_sin(k)
128         return out
```

3.3 The Proper Orthogonal Decomposition

For the POD reconstruction of IR scans, the POD class from the modulo-vki python library was used. The POD was used to reconstruct the dataset at every eighth rank starting from zero and ending at 152.

3.4 Dynamic Mode Decomposition

For the DMD reconstruction of IR scans, the DMD class from the pydmd python library was used. The DMD was used to reconstruct the dataset at every eighth rank starting from zero and ending at 152.

Chapter 4

Results and Discussion

4.1 Data

The average time taken to reconstruct each of the hurricane images for each rank, for each technique, was recorded before those reconstructions were sent to the neural net training procedure. After each image reconstruction stage, the data collected from each training stage, the mean and variance of the eight accuracies achieved by training a CNN on each fold of the 166 training images, was recorded. The resulting data was organized into three graphs, shown below.

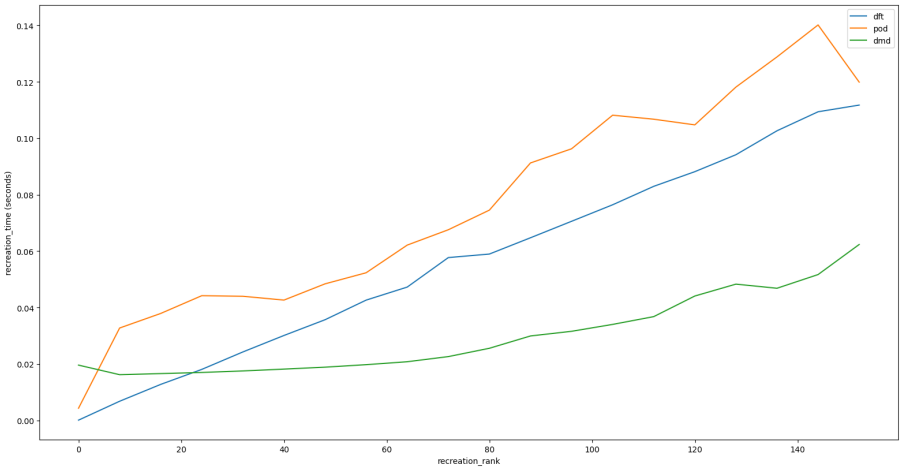

Figure 4.1: The average time taken to recreate each of the 166 training images using DFT, POD, and DMD at every eighth rank starting at zero and ending at 152.

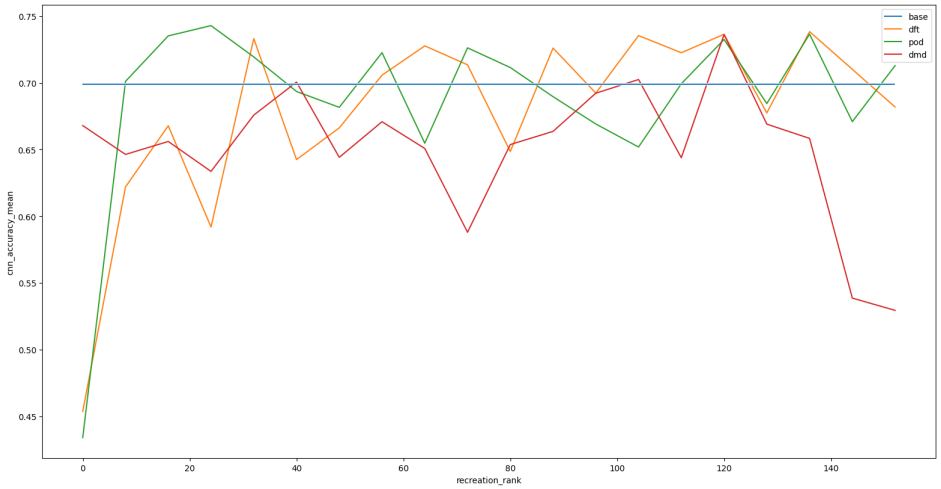

Figure 4.2: The average accuracy of each of the eight accuracies from the eight folds of CNN training on each rank of image reconstruction from each reconstruction technique. Note that the blue base-data line only represents one data point, which was extended by hand across the graph for the purpose of comparison.

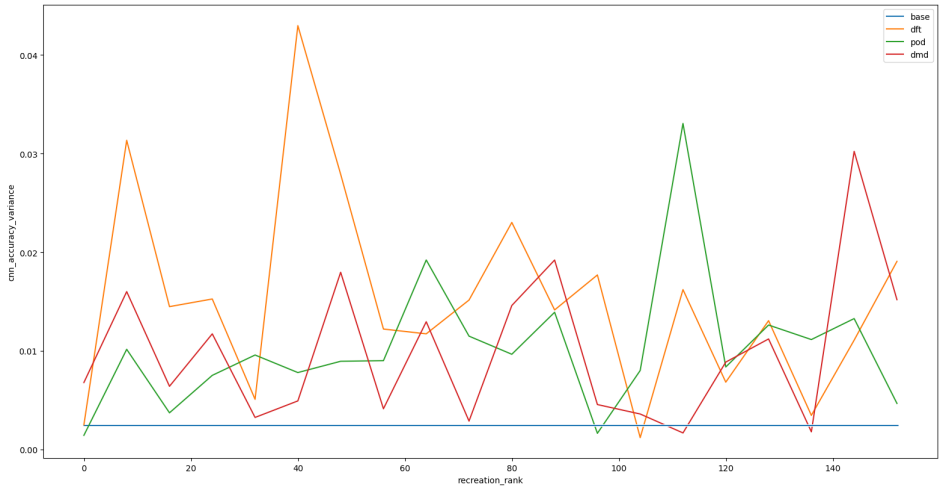

Figure 4.3: The variance of each of the eight accuracies from the eight folds of CNN training on each rank of image reconstruction from each reconstruction technique. Note that the blue base-data line only represents one data point, which was extended by hand across the graph for the purpose of comparison.

4.2 The Control

The CNN trained on unprocessed images of hurricanes reported a mean accuracy of almost exactly 70%, with extremely low variance. Given the small set of training images, this low variance is surprising. Furthermore, since the training variance for the POD at ranks above 80, where the reconstructed images are more-or-less guaranteed to be almost indistinguishable from the originals, has a fluctuating variance that only once dips below the base variance, it seems likely that the low base variance was a fluke.

4.3 The Discrete Fourier Transform

The DFT recreation times were unsurprisingly linear, likely because the only real work done by the algorithm is a number of dot products equal to the rank of image recreation. That being said, a real-world application of DFT modes would utilize the FFT (Fast Fourier Transform), which would, at least asymptotically, be dramatically faster than the regular DFT.

The accuracies of the DFT-trained CNNs reaches the base image accuracy around rank 60, and then oscillates around that value.

At low ranks, the DFT has the highest variances of the three techniques, likely because the DFT modes are not ordered by their contribution to each dataset, and thus carry no guarantee of reconstructing the images in any "optimal" order. Overall while it is somewhat remarkable that training CNNs the low-rank DFTs of the hurricane images yielded accuracies greater than 0.7, the DFT at no point seemed to particularly outperform base data.

4.4 The Proper Orthogonal Decomposition

The POD was the most computationally expensive of the three techniques, but the average time taken to generate POD recreations of the hurricane imagery was, like the DFT, roughly linear, and had roughly the same slope as the DFT.

The POD accuracy at low rank, i.e. between rank 8 and rank 32, was generally better than or at least on par with the base accuracy, making it the fastest algorithm to reach base accuracy, and likely suggesting that the best method for classifying hurricanes in this dataset is to train the CNN on rank $r < 40$ POD reconstructions of the dataset.

The POD variances are almost all higher than the base variance, which may be a fluke, and they have two large spikes between rank 60 and 120, which is odd given that, due to the nature of DMD, the POD reconstructions of images at rank

$r > 60$ should be mostly identical.

4.5 Dynamic Mode Decomposition

The DMD recreation times were the lowest among the algorithms after rank 24, likely duet to the fact that increasing the rank of a DMD reconstruction only requires solving a one-dimension higher eigenvalue problem, as opposed to computing the dot product of a new mode with each vector of the original image.

The accuracies of CNNs trained on DMD reconstructions were lower then those from the base data and the other techniques at almost every rank. This is likely due to the way in which DMD reconstructs an image, beginning with its leftmost column of pixels and "drawing" the image from left to right. The DMD reconstructions were the only ones to give better-than-random results at rank-0, which is due to the way pydmd handles its rank argument. Essentially, the rank argument used to construct an instance of pydmd's DMD class determines the rank of SVD used in the pseudoinverse of X, recall X and X' from the DMD algorithm, but it modifies that number before computing the SVD. Unlike the other techniques, the accuracies from DMD reconstructions start to fall off after rank 116, perhaps due to some strange artifacts from the high-rank reconstructions.

The DMD algorithm variances were generally lower than those from the other algorithms, likely because its accuracies were lower. The DMD variances spike at rank 144, approximately where the DMD accuracies begin to fall off towards 0.5. This supports the notion that an artifact is distorting the high-rank DMD reconstructions.

4.6 Conclusion

For the purpose of extracting image features via low-rank reconstruction, the POD seems to show the most immediate results, reaching the base accuracy by rank-8.

Since the POD is defined to extract the dominant features of an image as quickly as possible, and since hurricanes tend to take up a large portion of a 160×160 crop of a GIBBS IR scan (this is why we chose those dimensions), it makes sense that hurricanes would quickly contribute to the composition of a reconstructed image. We can observe this by looking at some low-rank reconstructions of IR scans showing hurricanes.

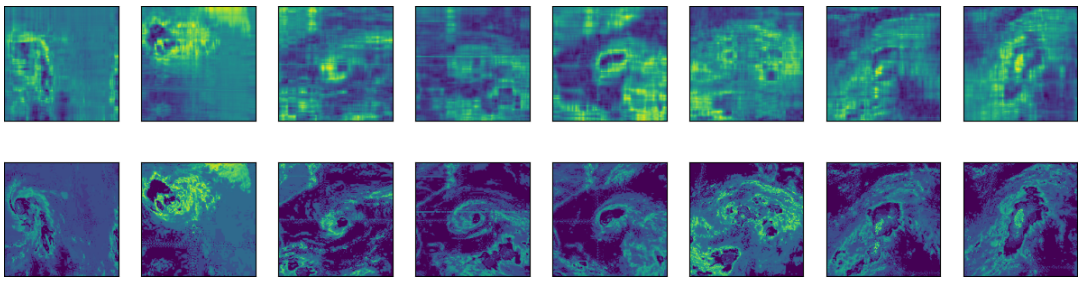

Figure 4.4: These are eight rank-8 POD reconstructions of IR scan crops containing hurricanes (top), and the crops from which they were constructed (bottom). In each reconstruction, a distinctly colored region can be observed corresponding to the hurricane in the original crop.

The DFT Showed middling performance among the techniques, only converging to oscillate about the base rate at around rank 48 at which point reconstructed images look convincingly like the originals (shown below).

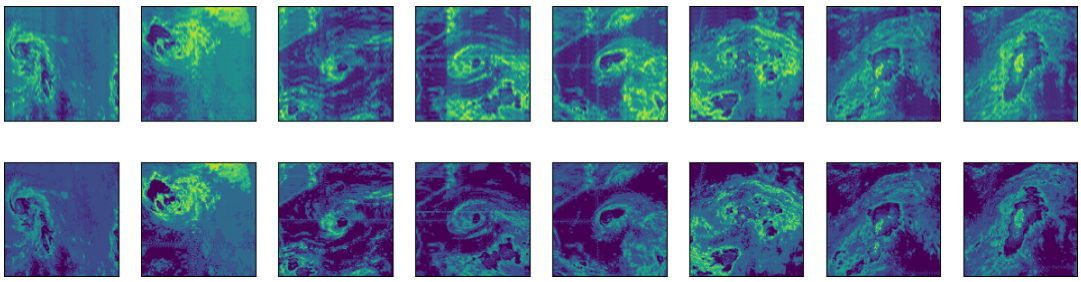

Figure 4.5: These are eight rank-48 DFT reconstructions of IR scan crops containing hurricanes (top), and the crops from which they were constructed (bottom). While there are distinct visual artifacts coming from the DFT reconstruction, at this point the reconstructed images seem to contain more-or-less every feature of the originals.

From this performance, it seems reasonable to infer that the DFT, while perfectly capable of reconstructing a dataset, does not naturally excel at extracting visual features of data. This is likely a feature of the DFT basis. While the DFT basis is ideal for many analysis tasks involving systems of differential equations, it is not particularly attuned to identifying patterns in a dataset that aren't themselves sinusoidal. Furthermore, since the DFT basis is fixed by the dimension of the dataset one is working on, it should not be expected to identify features of that dataset as well as a tailored basis such as is created by the POD.

The DMD reconstructed dataset was the computationally cheapest to create, but was also showed the worst results for the CNN training procedure, only reaching base performance at three ranks in the entire test. The DMD was likely cheaper than the other algorithms because of the lack of a need to project the dataset onto the DMD modes once they were created, instead just generating the "dynamics" of each mode with an exponential function of its associated eigenvalue. The poor performance of the DMD for classification is likely due to it plain unsuitedness to image reconstruction. DMD is ultimately an algorithm designed to generate models for time series data, and the procedure of "drawing" the image from left to right with a linear model should not be expected to work well for data not governed by any "natural" system of differential equations. The DMD experiences a particularly strong dip in performance at rank 72, though it's hard to tell why from visual inspection of the reconstructions (shown below).

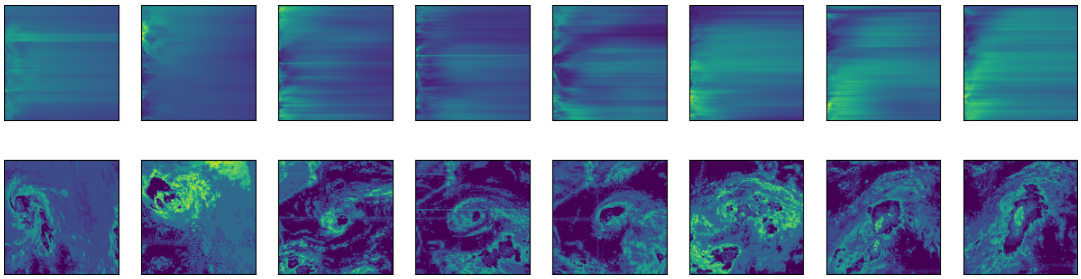

Figure 4.6: These are eight rank-72 DMD reconstructions of IR scan crops containing hurricanes (top), and the crops from which they were constructed (bottom). There aren't many distinguishable features in any of the images.

The DMD briefly reaches better-than-base performance at rank-120, before plummeting to around 0.52 at rank 152. To understand this, let's compare the rank-120 and rank-152 reconstructions of the hurricane scans we've been displaying thus far.

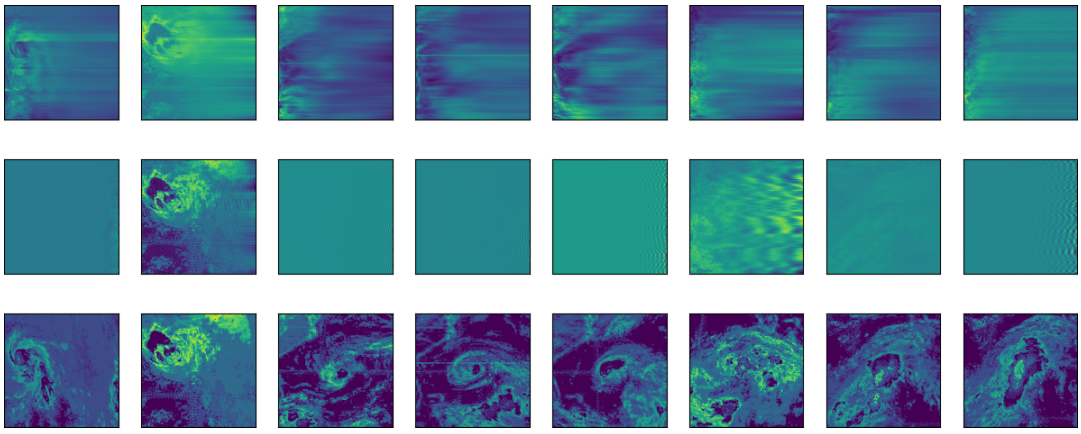

Figure 4.7: These are eight rank-120 DMD reconstructions of IR scan crops containing hurricanes (top), the rank-152 reconstructions of the same scans (middle), and the crops from which they were constructed (bottom). Most of the high-rank reconstructions are completely unrecognizable as scans of hurricanes.

It is worth noting here that, even though the DMD isn't able to perform noticeably better than the base data until it reaches rank-120, the rank-120 DMD is about as computationally expensive the rank-60 DFT or the rank-40 POD.

At high rank, the DMD reconstructions seem to devolve into rapidly-oscillating patterns of noise. Since DMD reconstructions are generated by adding together complex-exponential products of vectors, it seems possible that adding too many exponential functions would lead to an instability in the reconstructions, i.e. a high sensitivity to the left-most column of the image, but this is just speculation.

4.7 Future Work

Since the experimental design of this thesis changed many times, the actual code was run close to the end of the year. Due to this, and to some fear of power outages, many sacrifices in data resolution and CNN training time were made. It would be interesting to carry out the CNN training procedure with all ranks of data reconstruction, as opposed to just every eighth, and to increase the train time of the neural net. It could also be worthwhile to investigate the features of modulo and pydmd further before carrying out a follow-up experiment, and to implement more features in the custom DFT. In particular, one might investigate sparse POD and DMD, which I admit to not really understanding. I believe that one of the most profitable avenues of further research in using neural nets to measure the performance of techniques in modal analysis at specific-feature reconstruction is application to time series data. Each of the three techniques discussed in this thesis has a precedent for analyzing time-series data, and POD and DMD in particular have demonstrated a great ability to extract meaningful time-varying modes from the behavior of fluid vector fields (investigating this was my original intention with the hurricanes).

Data-driven modal analysis as an extremely rich and interdisciplinary field of study. It connects topics from linear algebra and analysis and utilizes a great deal of computer science, and it is applicable to an extremely wide variety of topics in the natural sciences and beyond. I strongly believe that any future students who choose to look into this rich field will find great academic benefit and fulfillment.

Chapter 5

Appendix

5.1 Euler's Formula and the complex trigonometric identities

The complex trigonometric identities form a crucial and satisfying bridge between geometry and complex analysis. In particular, they encode a useful way of thinking about the multiplication of complex numbers, that being as a composition scaling and rotation, and allow us to observe many trigonometric formulas as simple algebraic results. We will now proceed with the first result, Euler's formula.

Theorem 5.1.1. Euler's Formula

Let i be the usual $\sqrt{-1}$, $\theta \in \mathbb{R}$. Then $e^{i\theta} = \cos(\theta) + i\sin(\theta)$.

In order to prove this, we will need to cite two results from real analysis.

1. Let $x \in \mathbb{C}$, the set of complex numbers $a + ib$. Then $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$.
2. Let $\theta \in \mathbb{R}$, $\cos(\theta) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$.
3. Let $\theta \in \mathbb{R}$, $\sin(\theta) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$.

These follow from the Taylor expansions of e^x, $\cos(\theta)$, and $\sin(\theta)$. Now we can prove Euler's Formula.

Proof 5.1.1. Euler's Formula

Let $\theta \in \mathbb{R}$. Then we write

$$\begin{aligned} e^{i\theta} &= \sum_{k=0}^{\infty} \frac{(i\theta)^k}{k!} \\ &= \sum_{k=0}^{\infty} \frac{i^k \theta^k}{k!} \\ &= \sum_{k=0}^{\infty} \frac{i^{2k}\theta^{2k}}{(2k)!} + \sum_{k=0}^{\infty} \frac{i^{2k+1}\theta^{2k+1}}{(2k+1)!} \\ &= \sum_{k=0}^{\infty} (-1)^k \frac{\theta^{2k}}{(2k)!} + i \sum_{k=0}^{\infty} (-1)^k \frac{\theta^{2k+1}}{(2k+1)!} \\ &= \cos(\theta) + i\sin(\theta). \end{aligned}$$

This allows us to write two very useful trigonometric identities.

Corollary 5.1.1. Let $\theta \in \mathbb{R}$. Then we write

$$\begin{aligned} e^{i\theta} &= \cos(\theta) + i\sin(\theta) \\ e^{i\theta} + e^{-i\theta} &= 2\cos(\theta) \\ \cos(\theta) &= \frac{e^{i\theta} + e^{-i\theta}}{2}. \end{aligned}$$

Corollary 5.1.2. Let $\theta \in \mathbb{R}$. Then we write

$$\begin{aligned} e^{i\theta} &= \cos(\theta) + i\sin(\theta) \\ e^{i\theta} - e^{-i\theta} &= 2i\sin(\theta) \\ \sin(\theta) &= \frac{e^{i\theta} - e^{-i\theta}}{2i}. \end{aligned}$$

Chapter 6

Data and Software

6.1 Data

This project used the Tropical Cyclone Reports provided by the National Hurricane Center and Central Pacific Hurricane Center, of the National Oceanic and Atmospheric Administration (NOAA). They can be found at this link: https://www.nhc.noaa.gov/data/tcr/.

The meteorological image data sourced in this project came from the National Centers for Environmental Information Global ISCCP B1 Browse System (Gibbs), which can be accessed at this link: https://www.ncdc.noaa.gov/gibbs/.

6.2 Software

Almost all of the computational work in this thesis was done using Python. The python package PyDMD was used for all testing of the DMD algorithm. The Python package modulo was used for all testing of the POD algorithm. The Python packages matplotlib and numpy were used extensively throughout the entirety of the thesis. The Python package Celluloid was used to assemble videos from imported

and reconstructed data. The pipeline used for importing satellite images from NASA was written in MS-DOS.

ImageJ was used to select regions of interest from the IR weather data.

The Python package Torch was used to write the image classifier employed in testing the modal analysis algorithms.

The code used for this thesis can be found at my GitHub repository: `https://github.com/ryliefoster/thesis`.

Bibliography

Baksalary, O. & Trenkler, G. (2021), 'The moore–penrose inverse: a hundred years on a frontline of physics research.', *The European Physical Journal H* .
URL: *https://doi.org/10.1140/epjh/s13129-021-00011-y*

Brunton, S. & J. Kutz, N. (2019), *Data-Driven Science and Engineering*, Cambridge University Press.

Holbrook, R. (2019), 'Math For Machines least squares with the moore-penrose inverse'.
URL: *https://mathformachines.com/posts/least-squares-with-the-mp-inverse/*

J. Kutz, N., Brunton, S., Brunton, B. & L. Proctor, J. (2016), *Dynamic mode decomposition: data-driven modeling of complex systems*, David Marshall.

Kutz, N. (2018), 'Dynamic mode decomposition (theory)', `https://www.youtube.com/watch?v=bYfGVQ1Sg98`.

W. Smith, S. (2002), *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing.

Weiss, J. (2019), 'A tutorial on the proper orthogonal decomposition', *2019 AIAA Aviation Forum* pp. 17—21.
URL: *https://depositonce.tu-berlin.de/bitstream/11303/9456/5/podnotes_aiaa2019.pdf?msclkid=6370565bb3ce11ec8ff7f6cdf67629ff*