# Math 5467: Introduction to the Mathematics of Image and Data Analysis

Jeff Calder

University of Minnesota
School of Mathematics

jwcalder@umn.edu

March 16, 2021

## Contents

1 Introduction ................................................................. 3
   1.1 Course Information .............................................. 3
   1.2 Python ............................................................ 3
   1.3 Background on audio, images, and data analysis ............... 5
      1.3.1 Audio ...................................................... 6
      1.3.2 Images .................................................... 7
      1.3.3 Data science and machine learning ...................... 9

2 Linear Algebra Review .................................................. 10
   2.1 Notation ............................................................ 10
   2.2 Projection .......................................................... 12
   2.3 Diagonalization of symmetric matrices ....................... 14
   2.4 Vector calculus ................................................... 15

3 Principal Component Analysis ......................................... 16
   3.1 Fitting the best linear subspace .............................. 17
   3.2 How many principal directions? ............................... 22
   3.3 Robust PCA ...................................................... 23
   3.4 PCA-based Image Compression .................................. 26
   3.5 PCA-based Handwritten Digit Recognition ................... 32
1 Introduction

1.1 Course Information

This course is a modern introduction to the mathematics of image and data analysis. The course will cover the discrete Fourier and Wavelet transforms, with applications to image and audio processing. We will also cover the mathematics of common data analysis algorithms, including principal component analysis (PCA), data ranking (e.g., Google’s PageRank for ranking webpages), and clustering algorithms such as $k$-means and spectral clustering. Time-permitting, we will give an introduction to machine learning (ML), and cover basic ML classifiers, neural networks (in particular, convolutional neural networks for image classification), and graph-based learning.

The course will cover both mathematical theory and practical applications. We will use Python for all computational work in this course. Students will get hands on experience working with real data through a series of computational projects that will be completed throughout the term, on topics such as audio or image compression, facial recognition, or image classification. We will start the course with a gentle introduction to Python; no prior knowledge is required. See the course website, below, for details on how to get access to Python:

http://www-users.math.umn.edu/~jwcalder/5467S21/index.html

1.2 Python

These lecture notes are accompanied by Python notebooks that can be found on the course website, and in links throughout these notes. Below are links to Google Colab Python notebooks with basic introductions to various aspects of Python programming.

Introduction to Python Notebooks:

1. Introduction to Python
2. Introduction to Numpy
3. Reading and writing images and audio in Python
4. Introduction to Pandas

Below are some exercises to be completed in Python (most are in the notebooks above as well).
Exercise 1.1. Write a Python function that approximates \( \sin(x) \) using the Taylor expansion \( \sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \). Test your function for simple known values of \( \sin(x) \), such as \( \sin(0) = 0 \), \( \sin(\pi/4) = \frac{\sqrt{2}}{2} \), \( \sin(\pi/2) = 1 \), and \( \sin(\pi) = 0 \), etc. How accurate is the approximation?

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc.

Exercise 1.2. Write a Python function that computes the square root of a positive number using the Babylonian method. The Babylonian method to compute \( \sqrt{S} \) for \( S > 0 \) constructs the sequence \( x_n \) by setting \( x_0 = S \) and iterating

\[
x_{n+1} = \frac{1}{2} \left( x_n + \frac{S}{x_n} \right).
\]

Your code can take as input a tolerance parameter \( \varepsilon > 0 \), and should iterate until \( |x_n^2 - S| \leq \varepsilon \), and then return \( x_n \). Test your square root function to make sure it works.

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc.

Exercise 1.3. Write a Python program that uses the Sieve of Eratosthenes to find all prime numbers between 2 and a given integer \( n \).

Your function should use only basic Python programming. In particular, do not use any packages, like Numpy, Scipy, etc.

Exercise 1.4. Write a Python function that computes the largest magnitude eigenvalue of a square matrix with the power iteration. The power iteration is

\[
x_{n+1} = \frac{Ax_n}{\|Ax_n\|}.
\]

For a diagonalizable matrix, the iteration converges to the eigenvector of \( A \) with largest magnitude eigenvalue. The eigenvalue is

\[
\lambda = \lim_{n \to \infty} \|Ax_n\|.
\]

Compare your function to the true eigenvector and eigenvalue for small matrices where you can compute it by hand, to check that your function works.

In this exercise you may use Numpy. Try to write your code with only one loop, over the iterations in the power method.
Exercise 1.5. Write a Python function that numerically approximates \( \pi \) via the integral expression
\[
\pi = 4 \int_{0}^{1} \sqrt{1 - x^2} \, dx.
\]
Your function should not use any loops. Use a Numpy array and Numpy functions instead. How many decimal places of \( \pi \) can you accurately compute?

In this exercise you may use Numpy. Can you write the code without any loops? \( \square \)

The following exercise may be of interest, but does not involve Python.

Exercise 1.6. Prove that the iteration in the Babylonian method above converges quadratically to the square root of \( x \). In particular, show that the error \( \varepsilon_n = \frac{x_n}{\sqrt{x}} - 1 \) satisfies
\[
\varepsilon_{n+1} = \frac{\varepsilon_n^2}{2(\varepsilon_n + 1)}.
\]
From this, we get that \( \varepsilon_n \geq 0 \) for \( n \geq 1 \), and so
\[
\varepsilon_{n+1} \leq \frac{1}{2} \min\{\varepsilon_n^2, \varepsilon_n\}.
\]
Why does the inequality above guarantee convergence (i.e., that \( \varepsilon_n \to 0 \) as \( n \to \infty \))? \( \square \)

1.3 Background on audio, images, and data analysis

A major theme in this course is that of finding a good basis to represent your data. Usually data (e.g., images, audio, video, etc.) are captured in formats that are convenient for acquiring the data, but very inconvenient for storage or other kinds of processing. Much of this course is concerned with finding new bases for image and audio data that are better for tasks like image compression or image classification, among others. A change of basis can be linear, which the reader should be familiar with from linear algebra, or nonlinear as in modern deep neural networks. The change of basis can also be either hand-crafted (like in Fourier or Wavelet analysis), or learned from the data, as in Principal Component Analysis (PCA) or deep learning. Before we proceed with the course, we review the formats in which images and audio are typically recorded and stored.
1.3.1 Audio

A digital audio signal is a series of discrete samples of a continuous audio signal generated by the movement of the vibrating membrane of a microphone (of course, the vibrations in the membrane are caused by the rarefaction and compression of air, e.g., sound waves). Audio signals can be captured at different sampling rates and bit-depths. Standard CD quality audio has two channels (left and right) sampled at 44,100 Hz with 16-bits per sample. This means each channel has 44,100 samples per second, each encoded as a (signed) 16-bit integer to represent the magnitude of the sound wave at that sample. By the Sampling Theorem, this sampling rate allows CD audio to represent frequencies up to 22,050 Hz, which is well above what most people can hear. Figure 1.1 shows stereo audio signals for a piece of classical music.

The bit-rate for CD quality audio is the number of bits used per second of audio, which we can compute as

$$\frac{2 \text{ Channels}}{} \times \frac{44,100 \text{ Samples per Second}}{} \times \frac{16 \text{ Bits per Sample}}{} = 1,411,200 \text{ bits/second}.$$ 

Often this is written in terms of kilobits (kbit)—a kilobit is 1000 bits—so CD audio has a bitrate of 1,411 kbit/s (also denoted 1,411 kbps). A 4 minute long song would thus take

$$\frac{4 \text{ Minutes}}{} \times \frac{60 \text{ Seconds per Minute}}{} \times 1,411 \text{ kbits/second} = 338,640 \text{ kbits},$$

of space on disk. In terms of megabits (Mbit)—a megabit is 1000 kilobits—this is 339 Mbits. The reader may be more familiar with kilobytes (kB) and
megabytes (MB). A byte is 8 bits, and so 339 Mbits is actually around 42 MB.\(^1\)

The reader who is familiar with digital audio files will probably know that an mp3 audio file for a 4 minute song is usually around 5 MB, depending on the bitrate, which is much less than the 42 MB we computed for the raw data. In fact, a very high quality mp3 bitrate is 320 kbit/second\(^2\), which is less than one quarter of the bitrate of CD quality audio. Many mp3 audio files are compressed at much lower bitrates than this, with sometimes acceptable results. A main question we will address in this course is how to compress audio signals (and later images and video), without destroying important information. Other important tasks in audio processing are classifying audio signals (e.g., determining which song is playing automatically), and speech to text (e.g., determining what was said in an audio sample).

1.3.2 Images

A digital image is a discrete sampling of a two dimensional continuous signal given by the light hitting a rectangular image sensor in a digital camera. Thus, we can think of an image as a two-dimensional array of pixels. An image can be grayscale, in which case each pixel has a single number associated with it, representing the brightness, or image intensity, at that location. Figure 1.2 shows an example of a grayscale image, the famous cameraman image. On the right in the figure, we show the image plotted as a function (e.g., a surface), over the two-dimensional plane, where the height of the surface is the intensity of the pixel at that location. In a color image, each pixel is associated with several numbers, representing the intensity of different colors at that point, which mix to produce the correct color. The most common colorspace is RGB (red–green–blue), in which each pixel has three numbers representing the amount of red, green, and blue present at each pixel in the image. Thus, a grayscale image can be thought of as a matrix, while a color image can be thought of as 3 (or possibly more) matrices, one for each color channel. There are other colorspaces that are frequently used (e.g., by printers) that have more color channels, or use different colors. For example, in hyperspectral satellite imagery, it is not uncommon to have over 200 color channels in an image, each corresponding to a different frequency band of the electromagnetic spectrum.

---

\(^1\)Note we are using base 10 to define kilo and mega. It is also common to use base 2, so that a kilobit is \(2^{10} = 1024\) bits, and a megabit is 1024 kilobits.

\(^2\)It is widely accepted, except by some audio enthusiasts, that 320 kbit/second mp3 audio is indistinguishable to the human ear from CD quality audio.
Figure 1.2: Example of a grayscale digital image. The left shows the image in its usual form, while the right side depicts the image intensity as a surface (i.e., a function of 2 variables).

The cameraman image is of size $512 \times 512$, and so it has $262,144$ pixels. This is quite small by today’s standards. For example, a high-end modern smartphone has a camera with 12 million pixels (12 MP). For a color image, this requires storing 36 million numbers. Images usually have a bit depth of 8-bit, allowing $2^8 = 256$ different values for each color channel, or about 16 million different colors. Some newer smartphones use higher bit depths of 10-bits or 12-bits, yielding over 1 billion different colors. Let’s consider an 8-bit color image with 12 megapixels. The 36 million 8-bit numbers that are stored take up 36 MB of space. A 10-bit image would take 45 MB of space and a 12-bit image would take 54 MB of space. High end digital cameras can have even higher resolution, currently up to around 46 million pixels, yielding 8-bit images that take around 130 MB of space, and 14-bit images that take up to 240 MB.

However, images are rarely stored or transmitted in raw form, since it is a colossal waste of space. Common image compression algorithms, like jpeg, can compress images to roughly one tenth of their size, without noticeable loss in image quality. Building on our discussion of audio compression, we will learn in this course how to compress images without removing important information. Aside from image compression, there are many other important tasks in computer vision and image processing, such as image segmentation (determin-
ing the object of interest in an image), object recognition (determining what
the object is), image restoration (removing noise or blur), image inpainting
(recovering lost portions of images), and image classification. We will touch
on some of these other applications in the course.

A few words about video compression. We will not touch on this in the
course, but since video is simply a sequence of images, one could apply im-
age compression in this setting as well. However, there is a lot of temporal
information that is missed by such an approach, and modern video compres-
sion uses motion tracking and compresses only the differences between subse-
quent frames of video, which has significant advantages over image compression
alone.

1.3.3 Data science and machine learning

Finally, this course is not only about image and audio processing. Part of
the course will cover some fundamentals of data science and machine learning
from a mathematical perspective. In data analysis, one typically has many
datapoints, and the goal is to uncover structure in the data to perform tasks
like classification, clustering, dimension reduction, etc. Many problems in ma-
chine learning are related to image analysis (e.g., image classification), but
are substantially different in character than the image and audio compression
problems described in the preceding sections. In machine learning, we are
given many (possibly thousands or millions) of images and the task is to un-
derstand how the images naturally group together (e.g., clustering), and how
to automatically distinguish images from different classes (e.g., image classifi-
cation).

Figure 1.3 shows an example of some images from the MNIST dataset,
which we will use for illustrating machine learning throughout the course.
The MNIST dataset contains 70,000 grayscale images of handwritten digits 0
through 9. Each image is very small, containing only $28 \times 28$ pixels. We can
still consider the image compression problem, but here the goal would be to
compress the entire dataset, and not a single image. A more common task is
to train a machine learning classifier to recognize the number written in each
image. This is called optical character recognition, and is very commonly used
for many tasks, including archiving old newspapers or books, and teaching
self-driving cars to read street signs and house numbers. We will cover various
methods for image classification in the course, starting with a method based
on principal component analysis (PCA), and then moving on to more sophisti-
cated methods based on Fourier and Wavelet features, and deep convolutional
neural networks.
2 Linear Algebra Review

We recall some basic facts about linear algebra in this section.

2.1 Notation

We will use capital letters like $A, B, C, \ldots$ for matrices, and lower case letters, such as $x, y, z, \ldots$ for vectors. Scalars will be denoted by lower case letters $a, b, c$, or $a_1, a_2, \ldots$. When not specified, vectors are of length $n$ and matrices of size $m \times n$. Vectors are always treated as column vectors. If we have more than a handful of vectors, we will use subscripts to denote a collection of vectors, so $x_1, x_2, \ldots, x_p$ will always refer to a collection of $p$ vectors. For example, $e_1, e_2, \ldots, e_n$ will denote the $n$ standard basis vectors in $\mathbb{R}^n$. The reader should be careful not to confuse $x_i$ with the coordinates of the vector $x$.

We will rarely need to notate the coordinates of vectors or matrices, and if we do, we will use the notation $x(i)$ for the $i$th coordinate of $x$, or $e_i^T x$. Similarly, we will denote the entries of a matrix $A$ as $A(i, j)$.

The dot product of vectors $x$ and $y$ is given by the product $x^T y$, treating $x$ and $y$ as $n \times 1$ column vectors. To write this out in coordinate notation we
have

\[ x^T y = \sum_{i=1}^{n} x(i) y(i). \]

The dot product induces a norm

\[ \|x\| = \sqrt{x^T x} = \sqrt{x(1)^2 + x(2)^2 + \cdots + x(n)^2} \]

on the Euclidean space \( \mathbb{R}^n \). The quantity \( \|x - y\| \) is exactly the Euclidean distance between \( x \) and \( y \). A commonly used algebraic expansion is

\[ \|x \pm y\|^2 = \|x\|^2 \pm 2x^T y + \|y\|^2. \]

This can be verified immediately by writing \( \|x \pm y\|^2 = (x \pm y)^T (x \pm y) \) and expanding.

We say two vectors \( x \) and \( y \) are **orthogonal** if \( x^T y = 0 \). A vector \( x \) is a **normal** vector if \( \|x\| = 1 \), and two orthogonal vectors \( x \) and \( y \) that are also normal vectors are called **orthonormal**. A sequence of vectors \( v_1, v_2, \ldots, v_p \) is called orthogonal (resp. orthonormal) if each pair \( v_i, v_j \) is orthogonal (resp. orthonormal), provided \( i \neq j \). Consider a vector \( x \) in the span of orthonormal vectors \( v_1, v_2, \ldots, v_p \), that is

\[ x = \sum_{i=1}^{p} a_i v_i. \]

Then the norm of \( x \) can be computed by

\[ \|x\|^2 = \sum_{i=1}^{p} a_i v_i^T \sum_{j=1}^{p} a_j v_j = \sum_{i=1}^{p} \sum_{j=1}^{p} a_i a_j v_i v_j^T = \sum_{i=1}^{p} a_i^2. \]

**Exercise 2.1.** Given vectors \( x \) and \( y \), both of length \( n \), we will sometimes find it useful to construct the rank-one matrix \( xy^T \). The matrix is called “rank-one” since the range of \( xy^T \) is spanned by the vector \( x \), and is thus one-dimensional so the matrix has rank equal to one. Using the definition of matrix multiplication, \( xy^T \) is the \( n \times n \) matrix whose \((i,j)\) entry is \( x(i)y(j) \).

Let \( x_1, x_2, x_3, \ldots, x_m \) be a collection of vectors of length \( n \). Show that

\[ \sum_{i=1}^{m} x_i x_i^T = X^T X, \]

where \( X \) is the \( m \times n \) matrix whose \( i^{th} \) row is \( x_i^T \), which can be written as \( X = [x_1 \ x_2 \ \cdots \ x_m]^T \). \( \triangle \)
2.2 Projection

Let $L \subset \mathbb{R}^n$ be the linear subspace spanned by the collection of orthonormal vectors $v_1, v_2, \ldots, v_p$, where $p \leq n$. That is

$$L = \left\{ \sum_{i=1}^{p} a_i v_i : a_i \in \mathbb{R} \right\}.$$

In this case, $L$ is $p$-dimensional. The projection of a point $x \in \mathbb{R}^n$ onto $L$, denoted $\text{Proj}_L x$, is the closest point in the subspace $L$ to $x$. That is, $\text{Proj}_L x \in L$ satisfies

$$\|\text{Proj}_L x - x\| \leq \|y - x\| \quad \text{for all } y \in L.$$

We recall here some basic properties of the projection. First, we claim that

(2.2) \quad \text{Proj}_L x = \sum_{i=1}^{p} (x^T v_i) v_i

To see this, let us write $y \in L$ as

$$y = \sum_{i=1}^{p} a_i v_i,$$

and compute

$$\|x - y\|^2 = \|x\|^2 - 2x^T y + \|y\|^2 = \|x\|^2 - 2 \sum_{i=1}^{p} a_i x^T v_i + \sum_{i=1}^{p} a_i^2.$$

Minimizing over $a_i$ yields $a_i = x^T v_i$, which establishes the claim.

Since the $v_i$ are orthonormal, we have by (2.1) that

(2.3) \quad \|\text{Proj}_L x\|^2 = \sum_{j=1}^{p} (x^T v_j)^2

It can be useful to write (2.2) in matrix form. Let $V$ be the $n \times p$ matrix whose columns are $v_1, v_2, \ldots, v_p$, that is

$$V = \begin{bmatrix} v_1 & v_2 & \cdots & v_p \end{bmatrix}.$$

Then we have

(2.4) \quad \text{Proj}_L x = V V^T x.
Note that since the columns of $V$ are orthonormal, we have $V^T V = I$, and so

$$ (VV^T)^2 = VV^T V V^T = VV^T. $$

This is a natural property, and simply says that if we apply the projection twice, the second operation leaves the point unchanged, since it already lies in $L$. It is also clear that $V V^T$ is a symmetric matrix.

The residual is the difference $x - \text{Proj}_L x$. The residual is orthogonal to $L$, and we thus say the projection is *orthogonal projection*. Indeed, we compute

$$(x - \text{Proj}_L x)^T V = (x - V V^T x)^T V = x^T V - x^T V V^T V = 0.$$ 

Note we used that $V V^T = I$ in the third equality above. Subsequently, we have

$$\|x\|^2 = \|x - \text{Proj}_L x + \text{Proj}_L x\|^2 = \|x - \text{Proj}_L x\|^2 + \|\text{Proj}_L x\|^2,$$

and so

$$ (2.6) \quad \|x - \text{Proj}_L x\|^2 = \|x\|^2 - \|\text{Proj}_L x\|^2. $$

The matrix corresponding to the residual mapping is $I - V V^T$, and, as with the projection, the residual matrix satisfies

$$ (2.7) \quad (I - V V^T)^2 = I - 2 V V^T + V V^T V V^T = I + V V^T. $$

This is the analogous property to $(2.5)$. The matrix $I - V V^T$ is also clearly a symmetric matrix.

It is sometimes useful to project onto *affine spaces*. An affine space has the form $x_0 + L$\(^3\), where $L$ is a linear subspace. The key difference between affine and linear subspaces is that linear subspaces must contain the origin 0, while an affine space translates the origin to a new point $x_0$. To project onto an affine subspace, we simply translate the affine space and the point $x$ to the origin, project onto $L$, and translate back. That is, the projection onto the affine space $A = x_0 + L$, denoted $\text{Proj}_A$, is given by

$$ (2.8) \quad \text{Proj}_A x = x_0 + \text{Proj}_L (x - x_0). $$

We can also write this as

$$ \text{Proj}_A x = x_0 - \text{Proj}_L x_0 + \text{Proj}_L x.$$

\(^3\)The notation means $x_0 + L = \{x_0 + y : y \in L\}$. 

13
Exercise 2.2. Let \( L \) be a linear subspace of \( \mathbb{R}^n \).

(i) Show that \( \|\text{Proj}_L x\| \leq \|x\| \).

(ii) Show that \( \text{Proj}_L x = x \) if and only if \( x \in L \).

(iii) Show that if \( \text{Proj}_L x = x \) for all \( x \in \mathbb{R}^n \), then \( L = \mathbb{R}^n \).

\( \triangle \)

Exercise 2.3. Let \( V \) and \( W \) be orthogonal linear subspaces of \( \mathbb{R}^n \). This means that for each \( v \in V \) and \( w \in W \) we have \( w^T v = 0 \). Define

\[ V + W = \{ v + w : v \in V \text{ and } w \in W \}. \]

Show that

\[ \text{Proj}_{V+W} x = \text{Proj}_V x + \text{Proj}_W x. \]

\( \triangle \)

2.3 Diagonalization of symmetric matrices

Every symmetric matrix can be diagonalized. That is, for any symmetric matrix \( A \), there exists an orthogonal matrix \( Q \) and a diagonal matrix \( D \) such that

\[ A = QDQ^T. \]

An orthogonal matrix is a square matrix whose columns are orthonormal vectors. In this case, the columns of \( Q \) are exactly the eigenvectors of the matrix \( A \), which are orthogonal due to the symmetry of \( A \). An orthogonal matrix also has the property that all rows are orthonormal and thus

\[ Q^T Q = I = QQ^T. \]

Thus, the inverse of \( Q \) is \( Q^T \), and vice versa.

We generally arrange the eigenvalues of \( A \) from smallest to largest

\[ \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n. \]

The diagonal matrix \( D \) has exactly the eigenvalues \( \lambda_i \) on the diagonal, so \( D(i,i) = \lambda_i \). Thus the decomposition \( A = QDQ^T \) acting on a vector \( x \) can be interpreted as changing basis into the coordinates of the eigenvectors by taking \( Q^T x \), then multiplying by the diagonal matrix \( D \), and then converting back to the standard coordinates.
Exercise 2.4. Let $Q$ be an orthogonal matrix. Show that $\|Qx\| = \|x\|$. △

Exercise 2.5. Let $A$ be a symmetric matrix, and consider the optimization problem

$$(2.9) \quad \min \{x^T Ax : \|x\| = 1\}.$$ 

Since the set $\{x \in \mathbb{R}^n : \|x\| = 1\}$ is compact (closed and bounded), and the function $x \mapsto \|Ax\|$ is continuous, the optimization problem (2.9) admits a minimizer. Show that every minimizer $x^*$ is an eigenvector of $A$ with smallest eigenvalue. What happens if we switch the min to a max in (2.9)? [Hint: Diagonalize $A$ as $A = QDQ^T$ and write write the optimization problem in terms of $y = Q^T x$. Compute the optimal $y$ and convert back to the $x$ coordinates.] △

Exercise 2.6. We say a square matrix $A$ is positive semi-definite if $x^T Ax \geq 0$ for all vectors $x$. Show that a symmetric matrix is positive semi-definite if and only if all its eigenvalues are nonnegative. [Hint: Diagonalize $A$.] △

2.4 Vector calculus

It will sometimes be useful to differentiate real-valued functions of a vector $x$, and we record some basic identities here. We recall that for a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient $\nabla f$ is defined by

$$\nabla f(x) = \left( \frac{\partial f}{\partial x(1)}, \frac{\partial f}{\partial x(2)}, \ldots, \frac{\partial f}{\partial x(n)} \right).$$

For example, for the function $f(x) = x(1)^2 - x(2)^2$ on $\mathbb{R}^2$, the gradient is $\nabla f(x) = (2x(1), -2x(2))$. The gradient is important since it characterizes minima and maxima of $f$, via the necessary condition $\nabla f = 0$.

For a linear function $f(x) = y^T x$, we clearly have $\nabla f(x) = y$. For a quadratic function $f(x) = x^T Ax$, where $A$ is an $n \times n$ matrix, we have

$$(2.10) \quad \nabla f(x) = (A + A^T)x.$$
Indeed, we compute
\[
\frac{\partial f}{\partial x(k)} = \frac{\partial}{\partial x(k)} \sum_{i=1}^{n} \sum_{j=1}^{n} A(i,j)x(i)x(j)
\]
\[
= \sum_{i=1}^{n} \sum_{j=1}^{n} A(i,j)(\delta(i,k)x(j) + \delta(j,k)x(i))
\]
\[
= \sum_{j=1}^{n} A(k,j)x(j) + \sum_{i=1}^{n} A(i,k)x(i)
\]
\[
= \sum_{i=1}^{n} (A(k,i) + A(i,k))x(i),
\]
which establishes the claim. Above, the notation \(\delta(i, j)\) is the Kronecker delta, which satisfies \(\delta(i, j) = 1\) when \(i = j\) and \(\delta(i, j) = 0\) when \(i \neq j\).

**Exercise 2.7.** Assume \(A\) is a symmetric matrix. Show that
\[
(2.11) \quad \nabla \|Ax\|^2 = 2A^2x,
\]
\(\triangle\)

**Exercise 2.8.** Let \(A\) be a symmetric matrix. Show that any minimizer \(x\) of (2.9) is an eigenvector of \(A\) with eigenvalue \(\lambda = x^TAx\), without using that \(A\) is diagonalizable. [Hint: Any minimizer of (2.9) is also a minimizer of the Rayleigh quotient
\[
f(x) = \frac{x^T Ax}{x^Tx}.
\]
Compute \(\nabla f(x)\), set \(\nabla f(x) = 0\), and use that \(\|x\| = 1\).] \(\triangle\)

### 3 Principal Component Analysis

One of the most important tasks in data analysis is that of finding simpler structures in data. One of the simplest mathematical structures is a linear subspace. In this lecture we will discuss how to find the best linear subspace approximating a collection of data points. This process is called principal component analysis (PCA). Projecting onto this best linear subspace effectively reduces the dimensionality of our data to the dimension of the subspace, yielding a useful dimension reduction algorithm. After developing the mathematical theory of PCA, we will explore applications to image compression and classification.
3.1 Fitting the best linear subspace

Let $x_1, x_2, \ldots, x_m$ be a collection of vectors in $\mathbb{R}^n$, which we treat as our data points. Figure 3.1 shows an example of a point cloud in $\mathbb{R}^2$ which is roughly one dimensional. Depending on the application, we may wish to approximate the data points by a linear or an affine subspace of $\mathbb{R}^n$. To find a line of best fit for the data in Figure 3.1, we would seek a one dimensional affine subspace that best approximates the data.

We proceed with the analysis for an affine space, and we will see that we can immediately reduce to the linear case. We seek an affine subspace $A = x_0 + L$, where $L$ is a $k$-dimensional linear subspace of $\mathbb{R}^n$ and $x_0 \in \mathbb{R}^n$, that best approximates our data in the mean squared sense. That is, we seek to minimize the mean squared error

$$E(x_0, L) = \sum_{i=1}^{m} \| x_i - \text{Proj}_A x_i \|^2$$

over all $k$-dimensional linear subspaces $L$ and translations $x_0 \in \mathbb{R}^n$. Let us write $L$ as the span of orthonormal vectors $v_1, v_2, \ldots, v_k$, and let $V$ be the matrix whose columns are the $v_i$.

We first consider the translation $x_0$. Recalling (2.8) we can write the energy $E$ as

$$E(x_0, L) = \sum_{i=1}^{m} \| x_i - x_0 - \text{Proj}_L (x_i - x_0) \|^2.$$
Then by (2.2) we have

\[ E(x_0, L) = \sum_{i=1}^{m} \| (I - VV^T)(x_i - x_0) \|^2. \]

Differentiating in \( x_0 \) using (2.11) we find that the optimal value for \( x_0 \) must satisfy

\[ 0 = \nabla_{x_0} E(x_0, L) = -2 \sum_{i=1}^{m} (I - VV^T)^2 (x_i - x_0) = 0. \]

Let us define the mean, or centroid, of the data to be

\[ (3.2) \quad \bar{x} = \frac{1}{m} \sum_{i=1}^{m} x_i, \]

and recall that \((I - VV^T)^2 = (I - VV^T)\). Bringing the summation above inside we obtain

\[ 0 = (I - VV^T)(x_0 - \bar{x}) = x_0 - \bar{x} - \text{Proj}_L(x_0 - \bar{x}). \]

Therefore \( x_0 - \bar{x} = \text{Proj}_L(x_0 - \bar{x}) \in L \). It is clear that the energy \( E \) is unchanged by adding an element of \( L \) to \( x_0 \), so we will take \( x_0 - \bar{x} = 0 \), or \( x_0 = \bar{x} \), for simplicity.

**Exercise 3.1.** Show that if \( v \in L \) then \( E(x_0 + v, L) = E(x_0, L) \). \( \triangle \)

The discussion above allows us to reduce to the case of fitting a linear subspace to our data. Indeed, if we wish to fit an affine subspace, the optimal offset is the centroid \( \bar{x} \), and we can simply center our data, by replacing \( x_i \) with \( x_i - \bar{x} \), and reduce to the problem of fitting a linear subspace to the centered data. Thus, without loss of generality we consider the problem of minimizing the energy

\[ (3.3) \quad E(L) = \sum_{i=1}^{m} \| x_i - \text{Proj}_L x_i \|^2 \]

over all \( k \)-dimensional linear subspaces \( L \) of \( \mathbb{R}^n \). Equivalently, we can view the problem as minimizing \( E \) over the orthonormal vectors \( v_1, v_2, \ldots, v_k \) that span \( L \). To proceed further, we rewrite the energy in a more convenient form.
Lemma 3.2. The energy $E(L)$ can be expressed as

$$E(L) = \text{Trace}(M) - \sum_{j=1}^{k} v_j^T M v_j,$$

where $M$ is the covariance matrix of the data, given by

$$M = \sum_{i=1}^{m} x_i x_i^T.$$

Remark 3.3. By Exercise 2.1, the covariance matrix $M$ can also be written as $M = X^T X$, where $X$ is the $m \times n$ matrix whose $i$th row is $x_i^T$, which can be written as $X = [x_1 \; x_2 \; \cdots \; x_m]^T$. In practice, one is usually given the data matrix $X$ directly, and the formula $M = X^T X$ is both more convenient and more efficient in many programming languages.

Proof of Lemma 3.2. We recall (2.3) and (2.6), and compute

$$E(L) = \sum_{i=1}^{m} (\|x_i\|^2 - \|\text{Proj}_L x_i\|^2)$$

$$= \sum_{i=1}^{m} \|x_i\|^2 - \sum_{i=1}^{m} \sum_{j=1}^{k} (v_j^T x_i)^2$$

$$= \sum_{i=1}^{m} \|x_i\|^2 - \sum_{i=1}^{m} \sum_{j=1}^{k} v_j^T x_i x_i^T v_j$$

$$= \sum_{i=1}^{m} \|x_i\|^2 - \sum_{j=1}^{k} v_j^T \left( \sum_{i=1}^{m} x_i x_i^T \right) v_j$$

$$= \sum_{i=1}^{m} \|x_i\|^2 - \sum_{j=1}^{k} v_j^T M v_j.$$

To complete the proof, we note that

$$\text{Trace}(M) = \sum_{i=1}^{m} \text{Trace}(x_i x_i^T) = \sum_{i=1}^{m} \|x_i\|^2. \quad \Box$$

The first term in (3.4) is independent of $L$, so we may focus on the second term. Due to the minus sign, Lemma 3.2 shows that minimizing $E$ is equivalent to maximizing the quantity

$$\sum_{j=1}^{k} v_j^T M v_j$$

19
over orthonormal vectors $v_1, v_2, \ldots, v_k$. We note that the covariance matrix $M$ is a symmetric matrix, thus it is diagonalizable. This means there exists an orthogonal matrix $P$, whose columns are the orthonormal eigenvectors of $M$, and a diagonal matrix $D$, whose diagonal entries are the corresponding eigenvalues, such that $M = PDP^T$. We arrange the eigenvalues from largest to smallest, so $D_{ii} = \lambda_i$ and

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n.$$  

Let $p_1, \ldots, p_n$ denote the corresponding orthonormal eigenvectors of $M$, which are just the columns of $P$.

We claim that $M$ is positive semi-definite, that is, that $\lambda_i \geq 0$ for all $i$. To see this, simply note that

$$\lambda_j = p_j^T M p_j = \sum_{i=1}^{m} p_j^T x_i x_i^T p_j = \sum_{i=1}^{m} (p_j^T x_i)^2 \geq 0.$$  

Since $p_j^T x_i$ is the projection of $x_i$ onto $p_j$, we see that $\lambda_j$ is simply the variance of the data in the direction $p_j$ (up to a normalizing factor of $1/m$). We now compute

$$\sum_{j=1}^{k} v_j^T M v_j = \sum_{j=1}^{k} v_j^T PDP^T v_j$$

$$= \sum_{j=1}^{k} \|D^{1/2}P^T v_j\|^2$$

$$= \sum_{j=1}^{k} \left( \sum_{i=1}^{n} \lambda_i (p_i^T v_j)^2 \right)$$

$$= \sum_{i=1}^{n} \lambda_i \sum_{j=1}^{k} (p_i^T v_j)^2$$

$$= \sum_{i=1}^{n} \lambda_i \|\text{Proj}_L p_i\|^2.$$  

Notice that

$$\sum_{i=1}^{n} \|\text{Proj}_L p_i\|^2 = \sum_{j=1}^{k} \sum_{i=1}^{n} (p_i^T v_j)^2 = \sum_{j=1}^{k} \|v_j\|^2 = \sum_{j=1}^{k} 1 = k,$$  

20
where the identity $\|v_j\|^2 = \sum_{i=1}^{n} (p_i^T v_j)^2$ follows from the fact that the $p_i$ form an orthonormal basis for $\mathbb{R}^n$. Thus, we must choose the $v_i$ so as to distribute the weights $\|\text{Proj}_L p_i\|^2$ among the largest eigenvalues as much as possible. A natural choice is to set $v_i = p_i$ for $i = 1, \ldots, k$. Then $p_i \in L$ for $i = 1, \ldots, k$, and the $p_i$ are orthogonal to $L$ for $i \geq k + 1$, yielding

$$\|\text{Proj}_L p_i\|^2 = \begin{cases} 1, & \text{if } 1 \leq i \leq k \\ 0, & \text{otherwise.} \end{cases}$$

This choice of $v_i$ then yields

$$(3.8) \quad \sum_{j=1}^{k} v_j^T M v_j = \sum_{i=1}^{k} \lambda_i. \quad \triangle$$

The exercise below verifies that this choice of $v_i = p_i$ is indeed optimal.

**Exercise 3.4.** Suppose that $\lambda_i$ satisfy (3.6) and let $a_1, \ldots, a_n$ satisfy $0 \leq a_i \leq 1$ and $\sum_{i=1}^{n} a_i = k$, where $k$ is an integer $1 \leq k \leq n$. Show that

$$\sum_{i=1}^{n} \lambda_i a_i \leq \sum_{i=1}^{k} \lambda_i. \quad \triangle$$

We summarize our findings in a theorem.

**Theorem 3.5.** Let $p_1, p_2, \ldots, p_n$ be the orthonormal eigenvectors of the covariance matrix $M$, defined in (3.5), with corresponding eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, given in decreasing order. The energy $E(L)$, defined in (3.3), is minimized over $k$-dimensional linear subspaces $L \subset \mathbb{R}^n$ by setting

$$L = \text{span}\{p_1, p_2, \ldots, p_k\}$$

and the optimal energy is given by

$$E(L) = \sum_{i=k+1}^{n} \lambda_i.$$  

**Proof.** All that remains to prove is the formula for $E(L)$. This follows from Eq. (3.8), Lemma 3.2, and the identity

$$\text{Trace}(M) = \sum_{i=1}^{n} \lambda_i. \quad \square$$
The vectors $p_1, p_2, \ldots$ in Theorem 3.5 are called the principal components of the data, and the eigenvalues $\lambda_1, \lambda_2, \ldots$ describe the amount of variation of the data in the direction of each principal component, due to (3.7). In Figure 3.2 we plot the principal components of the point cloud from Figure 3.1, scaled to length $2\sqrt{\lambda_i}$ to match the variation in each direction of the data.

Let $P_k = [p_1 \ p_2 \ \cdots \ p_k]$. Then, recalling Section 2.2, the projection of a vector $x$ onto the PCA subspace $L$ is given by

$$\text{Proj}_L x = P_k P_k^T x.$$  

The coordinates of the point $x$ in the subspace $L$ are the contents of the length $k$ vector $P_k^T x$. In this sense, we can view PCA as dimension reduction, since the length $n$ vector $x$ is approximated by the length $k$ vector $P_k^T x$. Here, PCA gives a dimension reduction from $\mathbb{R}^n$ to $\mathbb{R}^k$.

### 3.2 How many principal directions?

A basic question concerns how to chose the number of principal components $k$ to use in PCA. A standard way to do this is to instead specify how much of the variation in the data one wishes to capture with the subspace $L$. Let $0 < \alpha \leq 1$ describe this quantity. For example, $\alpha = 0.95$ is interpreted as requiring that the subspace $L$ capture 95% of the variation in the data. Since the eigenvalue $\lambda_i$ describes the amount of variation in the principal direction
we can simply choose $k$ as small as possible while ensuring that

$$
\sum_{i=1}^{k} \lambda_i \geq \alpha \sum_{i=1}^{n} \lambda_i.
$$

As in the proof of Theorem 3.5, we have $\sum_{i=1}^{n} \lambda_i = \text{Trace}(M)$ and so we may rewrite the condition as

$$(3.9) \quad \sum_{i=1}^{k} \lambda_i \geq \alpha \text{Trace}(M).$$

The condition (3.9) can be checked without computing all of the eigenvalues of $M$, which may be computationally intensive in high dimensional applications, where $n$ is very large. Instead, one can use an iterative eigenvalue solver, which finds the eigenvectors in order of decreasing eigenvalue, and stop the first time (3.9) holds.

### 3.3 Robust PCA

While PCA is simple to work with mathematically and computationally, it can be very sensitive to outliers in the data. This is due to its use of the mean squared error in (3.3), which strongly penalizes outliers. We show an example in Figure 3.3 of how a single outlier can lead to a large negative affect on the ability of PCA to accurately fit the main part of the point cloud.

Thus, it is important to remove outliers before applying PCA. Outlier detection can, however, be difficult, and so there has been substantial interest in more robust versions of PCA that are not sensitive to outliers. Many variants of robust PCA have been proposed in the literature. Some are based on minimizing a sum of distances energy of the form

$$
(3.10) \quad F(L) = \sum_{i=1}^{m} ||x_i - \text{Proj}_L x_i||.
$$

By omitting the square, we are placing a far lower penalty on severe outliers, and can achieve better performance. However, it is far more computationally challenging to minimize $F(L)$, compared to the mean-squared error $E(L)$, since there is no longer a simple relationship to the eigenvectors of the covariance matrix.
Figure 3.3: An illustration of how PCA is sensitive to outliers. Here, we used the same point cloud as in Figure 3.1, but added a single outlying point to the lower right of the point cloud (not depicted), whose distance from the point cloud is twice the length (longest dimension) of the point cloud. Since PCA minimize the sum of squared errors, it is overly concerned with approximating outliers, whose distance is far from the inlying point cloud.

Exercise 3.6. Consider the weighted PCA energy

\[ E_w(L) = \sum_{i=1}^{m} w_i \| x_i - \text{Proj}_L x_i \|^2, \]

where \( w_1, w_2, \ldots, w_m \) are nonnegative numbers (weights).

(i) Show that the weighted energy \( E_w \) is minimized over \( k \)-dimensional subspaces \( L \subset \mathbb{R}^n \) by setting

\[ L = \text{span}\{ p_1, p_2, \ldots, p_k \}, \]

where \( p_1, p_2, \ldots, p_n \) are the orthonormal eigenvectors of the covariance matrix

\[ M_w = \sum_{i=1}^{m} w_i x_i x_i^T, \]

with corresponding eigenvalues \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \), given in decreasing order.

(ii) Show that the weighted covariance matrix can also be expressed as

\[ M_w = X^T W X, \]
where $W$ is the $m \times m$ diagonal matrix with diagonal entries given by $w_1, w_2, \ldots, w_m$, and

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}^T.$$

(iii) Show that the optimal energy is given by

$$E_w(L) = \sum_{i=k+1}^{n} \lambda_i.$$

(iv) Suppose we minimize $E_w$ over affine spaces $x_0 + L$, so

$$E_w(x_0, L) = \sum_{i=1}^{m} w_i \|x_i - x_0 - \text{Proj}_L(x_i - x_0)\|^2.$$

Show that an optimal choice for $x_0$ is the weighted centroid

$$x_0 = \frac{\sum_{i=1}^{m} w_i x_i}{\sum_{i=1}^{m} w_i}.$$

Project 3.1 (Robust PCA). Python Notebook: .ipynb

Robust PCA refers to a class of algorithms that minimize an energy of the form

$$E(x_0, L) = \sum_{i=1}^{m} \Phi(\|x_i - x_0 - \text{Proj}_L(x_i - x_0)\|^2).$$

Choosing $\Phi(s) = s$ yields ordinary PCA, while $\Phi(s) = \sqrt{s}$ yields the robust sum of distances energy $F$ given in (3.10). In this project you will implement an iteratively re-weighted least squares (IRLS) algorithm for minimizing the robust PCA energy (3.11), and test it on some data with outliers. Please refer to the Python notebook linked above and complete the steps below.

1. Write a Python function to implement the weighted PCA from Exercise 3.6.

2. Write Python code to minimize (3.11) using the IRLS method, which solves a sequence of weighted PCA problems

$$(x_0^{k+1}, L^{k+1}) = \min_{(x_0, L)} \sum_{i=1}^{m} w_i^k \|x_i - x_0 - \text{Proj}_L(x_i - x_0)\|^2,$$
where

\[ w_i^k = \Phi(\|x_i - x_0^k - \text{Proj}_{L_k}(x_i - x_0^k)\|^2) \over \|x_i - x_0^k - \text{Proj}_{L_k}(x_i - x_0^k)\|^2. \]

Start with only a handful of iterations. You can then play with a stopping condition that checks how much the weights \( w_i^k \) change each iteration.

You may find that you divide by zero, or very small numbers, in the definition of the weights \( w_i^k \), which can cause problems. A common way to regularize is to define

\[ w_i^k = \Phi(\|x_i - x_0^k - \text{Proj}_{L_k}(x_i - x_0^k)\|^2) \over \|x_i - x_0^k - \text{Proj}_{L_k}(x_i - x_0^k)\|^2 + \varepsilon, \]

where \( \varepsilon \) is a small number, say \( \varepsilon = 10^{-5} \).

3. Try your Robust PCA method on data with outliers. How many outliers do you need to add to break the algorithm?

\[ \Delta \]

3.4 PCA-based Image Compression

Python Notebook: .ipynb

We now give a first application of PCA to the problem of image compression. To apply PCA in this context, we must break an image up into pieces in such a way that the pieces are expected to be particularly simple and can be expressed well by PCA with a small number of principal components. The simplest way to do this would be split the image up into its rows or columns. Let \( X \) be an \( m \times n \) matrix representing an \( m \times n \) grayscale image. If we split the image up by rows, then this amounts to applying PCA directly to the matrix \( X \), where each row is considered as a data point. In this case we do not center the data, so based on Remark 3.3, the covariance matrix is \( M = X^T X \). Let \( k \) denote the number of principal components to keep and let \( P_k = [p_1 \ p_2 \ \cdots \ p_k] \) be the matrix whose columns are the principal components, and let \( L_k \) denote the span of \( p_1, \ldots, p_k \). For a given vector \( x \), PCA approximates \( x \) by its projection onto \( L_k \), that is \( \text{Proj}_{L_k} x = P_k P_k^T x \). Since we work with row vectors in \( X \), let’s consider the transpose of this quantity, so that \( x_i^T P_k P_k^T \) is the PCA approximation of the \( i \)th row of \( X \), if \( X = [x_1 \ x_2 \ \cdots \ x_m]^T \). Thus, the PCA approximation of \( X \) is given by

\[ X \approx X P_k P_k^T. \]
Figure 3.4: The cameraman image and the decomposition of part of the image into patches of size $8 \times 8$.

Why does this amount to compression? To compute the right hand side above, we only need to store the matrix $P_k$, which is $n \times k$, and the matrix $XP_k$, which is $m \times k$. The latter matrix $XP_k$ represents the coordinates of each row of $X$ in the basis of $L_k$. Thus, instead of storing the $m \times n$ matrix $X$, we store two matrices of size $n \times k$ and $m \times k$. Often $m = n$ (the image is square), and so the compression ratio is $n : k$.

While row-wise compression of images is an obvious first choice, the method misses important structure in the image, and the rows are not necessarily simple pieces of the image to compress. Since each row spans the entire image, the pixel values across the entire row may be entirely unrelated and have very little redundancy (which is useful for compression). Furthermore, why not use columns? Neighboring pixels vertically in the image should also have some correlation, but this is entirely ignored by row-wise image compression.

A better way to split up an image for compression is to use patches, or blocks, that are localized in space. The pixel intensities do not often vary rapidly in local areas of the image, and so small patches are expected to be well-approximated by a low dimensional PCA approximation. We work with the $512 \times 512$ cameraman image shown in Figure 1.2. We use $8 \times 8$ pixel patches in a regular grid, so the image contains $64 \times 64 = 4096$ patches, each containing $8 \times 8 = 64$ pixels. Figure 3.4 shows some of the patches of the cameraman image. Working with patches instead of rows requires a small amount of preprocessing of the image, to split it into patches. This produces
a matrix $X$ of size $4096 \times 64$, and we apply PCA to this matrix, instead of to the image itself, as we did in the row-wise compression example above. After this preprocessing, the compression proceeds exactly the same as in the row-wise compression example, and the decompressed image then needs to be reconstructed from its patches.

The reconstruction error in image compression is measured with the peak signal to noise ratio (PSNR). The PSNR computation is based on the mean squared error

$$\text{MSE} = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (I(i, j) - I_0(i, j))^2.$$ 

Here, $I_0$ is the original image and $I$ is the reconstructed image after compression. Both images have size $m \times n$. The PSNR computation also uses the peak signal value, $S_{\text{peak}}$, which is the largest possible value of the pixel intensity. For 8-bit images, the largest value is $S_{\text{peak}} = 2^8 - 1 = 255$. The PSNR is then given by

$$\text{PSNR} = 10 \log_{10} \left( \frac{S_{\text{peak}}^2}{\text{MSE}} \right).$$

The PSNR is measured in decibels dB. PSNR values of 20 dB to 30 dB are very low quality images you may see on wireless devices, while 30 dB to 50 dB are respectable, and above 50 dB are very good quality compressions. This discussion is for 8-bit images, and the values would change for higher bit-depth images.

Figure 3.5 shows the compressed and difference cameraman images at three different compression ratios, with PSNR ranging from 34 dB up to 51 dB. The reader should note the blocking-type artifacts at higher compression ratios. These are caused by the decomposition of the image into patches, which allows for the reconstructed patches to differ greatly near the patch boundary. In Figure 3.6 we plot the PSNR versus compression ratio for patch-based (or block-based) image compression and row-wise image compression. This clearly shows the advantage of using block-based compression instead of row-wise compression.

The same types of blocking artifacts are present in older jpeg compression algorithms, which are based on the same principle of splitting the image into patches. Instead of using PCA to find a basis for the patch space, the jpeg algorithm uses the Fourier series basis, which we will learn about later in the course. We show in Figure 3.7 the first 30 principal components obtained
Figure 3.5: Examples of PCA-based image compression on the cameraman image at different compression ratios. Left is original, center is compressed, and right is the difference image.
Figure 3.6: PSNR vs Compression Ratio for block-wise and row-wise compression of the cameraman image.

Figure 3.7: The first 30 principal components extracted via PCA on $8 \times 8$ pixel patches of the cameraman image.

by applying PCA to the image patches. The principal components start of as low frequency, smooth, features, while the later components describe more high frequency content, like texture. When we study Fourier analysis later in the course, we will find that the Fourier basis for patch space looks strikingly similar to the principal components in Figure 3.7.

Let us also mention one aspect of compression we are omitting from this
discussion. Normally the compressed data, which is $XP_k$ and $P_k$, would be
further compressed with a lossless compression algorithm (like zip compres-
sion) to save additional space. This adds further compression above the lossy
compression we have described in this section. The mathematics of lossless
compression are unfortunately beyond the scope of this course.

**Project 3.2 (Audio Compression). Python Notebook: .ipynb**

This project will use the same PCA-based compression algorithm for audio
compression. There are two parts to the project, and the notebook above
contains some code to get you started.

1. Use the same block-based image compression algorithm, described above,
   for audio compression. You can use any audio file you like; the python
   notebook linked above downloads a classical music sample from the
course website that you can use. A stereo audio signal is an array of
   size $n \times 2$, where $n$ is the number of samples. Use blocks of size $N \times 2$
   for compression.

2. When you play back the compressed audio file, you will likely hear some
   static noise artifacts, even at very low compression rates. These are
   caused by blocking artifacts, where the signals do not match up on
   the edges of the blocks, which introduces discontinuities into the sig-
   nal (which are very high in frequency). This is similar to the blocking
   artifacts we observed in image compression (see Figure 3.5), however,
   the artifacts are more noticeable in audio than in images.

To fix this, audio compression algorithms use overlapping blocks, and
apply a windowing function in order to smoothly patch together the au-
dio in each block. The blocks are structured so that half of the first block
overlaps with half of the second block, and so on. To implement this in
python, just shift the signal by half of the block width, and apply the
`images_to_patches` function on the original and shifted signals. Then
compress and decompress both signals. After decompressing, and before
converting back from the patch format to the audio signal, you’ll need
to multiply by a windowing function to smooth the transition between
patches. If the patch size is $2 \times N$, then each channel should be multi-
plied by a window function $w_i$, $i = 0, 1, \ldots, N - 1$. A common window
function that is used, for example, in mp3 compression, is

$$w_i = \sin^2 \left( \frac{\pi}{N} \left( i + \frac{1}{2} \right) \right).$$
After you decompress and apply the window, undo the shift and add the signals together to get the decompressed audio. Does this improve the audio quality?

As a note, in order to make sure the shifted signals add up correctly, we need that
\[ w_i + w_{i+N/2} = 1. \]

As an exercise, the reader should check that the window function above satisfies this condition, which is called the Princen-Bradley condition.

\[ \triangle \]

### 3.5 PCA-based Handwritten Digit Recognition

**Python Notebook:** .ipynb

We now give a further application of PCA to the problem of handwritten digit recognition on the MNIST dataset, depicted in Figure 1.3. Recall MNIST is a dataset consisting of 70,000 images of handwritten digits 0 through 9. Each image is a 28 × 28 pixel grayscale image, which we can view as a vector in \( \mathbb{R}^{784} \) by flattening the image array, since \( 28^2 = 784 \).

PCA allows us to deduce a simple linear model for the images belonging to each digit. We choose a number of principal components \( k \), and use PCA to learn affine subspaces \( A_0, A_1, \ldots, A_9 \) for each of the 10 digits. A new image of a handwritten digit is classified by projecting the image onto each of the 10 affine spaces, and finding which it is closest to. In order have held-out testing images that are not used for constructing the affine subspaces, we use the first 60,000 images in the dataset to build the affine spaces, and the last 10,000 to test the accuracy of the digit recognition. This is a standard training/testing split for the MNIST dataset.

To describe this mathematically, let \( P_0, P_1, \ldots, P_9 \) denote the \( 784 \times k \) matrices whose columns are the \( k \) principal components for each class. In this case, we center the data before applying PCA, so let \( \overline{x}_0, \overline{x}_1, \ldots, \overline{x}_9 \) denote the mean images from each class. The mean MNIST images are shown in Figure 3.8, and the first 10 principal components per class are shown in Figure 3.9. To project a flattened MNIST image \( x \in \mathbb{R}^{784} \) onto the affine space \( A_i \), we recall from Section 2.2 that the formula for projection onto an affine space is

\[ \text{Proj}_{A_i} x = \overline{x}_i + P_i P_i^T (x - \overline{x}_i). \]
To check how far $x$ is from the affine space $A_i$, we compute the norm of the difference between $x$ and its projection, that is

$$d_i(x) := \|x - \text{Proj}_{A_i} x\| = \|(I - P_i P_i^T)(x - \pi_i)\|.$$ 

The quantity $d_i$ is the distance between $x$ and the affine space $A_i$. We can then classify the image $x$ by choosing the closest affine space, that is, the label $\ell_i(x)$ for image $x$ is given by

$$\ell_i(x) = \arg\min_{0 \leq i \leq 9} d_i(x).$$
Figure 3.10: Some MNIST images that were incorrectly classified, along with
the incorrectly predicted label.

Running this on the MNIST dataset produces a classifier that achieves 95.8%
accuracy on the 10,000 held out testing images. This is a very good result
for such a basic algorithm. State of the art deep neural networks can achieve
over 99% accuracy. Figure 3.10 shows some of the incorrectly labeled digits,
along with the incorrectly predicted label. We can see many of the incorrectly
classified digits are poorly written and thus difficult to classify.

Remark 3.7. The PCA-based classifier we have constructed is a simple affine
mapping \((I - P_i P_i^T)(x - \pi_i)\), followed by a nonlinear operation—computing
the norm of this quantity. We then check which of \(d_0(x), d_1(x), \ldots, d_9(x)\) is
smallest to decide on the label for the image \(x\). This is similar to a single
layer neural network (a neuron is a linear function composed with a nonlinear
activation function). We will cover the basics of neural networks later in the
course.

Remark 3.8. In Python, and other programming languages, it is inefficient
to compute with single images, so it is useful to write the formulas above in
terms of a matrix \(X\), whose rows are images from the MNIST dataset. Then
we must work with the transposes of the quantities above, so

\[(X - \pi_i^T)(I - P_i P_i^T)\]

represents the difference between the projections of all rows of \(X\) onto the \(i^{th}\)
affine space \(A_i\).

Let us mention briefly that the expression \(X - \pi_i^T\) is intended to mean that
we subtract the row vector \(\pi_i^T\) from all rows of \(X\). This type of expression is
allowed in Python and other languages like Matlab. To be absolutely correct
mathematically, the expression should be written as

\[(X - 1\pi_i^T)(I - P_i P_i^T)\]

where \(1\) is the all-ones vector whose length is the same as the number of rows
in \(X\).
Project 3.3 (EigenFaces). Python Notebook: .ipynb

In this project you will explore a PCA-based facial recognition algorithm that is known as *EigenFaces*. The Python notebook above has code to get you started, including downloading and viewing a database of face images. The dataset has 2414 images of faces, from 38 different subjects. Each person appears many times in the database under different lighting conditions, etc. The goal of face recognition is to match a new image of a face to an image in an existing database. Please refer to the Python notebook above while completing the project steps below.

1. To have held-out data for testing, we will first split the face dataset into training and testing sets. Use the `train_test_split` function in `sklearn.model_selection` to split the dataset randomly into training (70%) and testing (30%).

2. Run PCA on the training images to learn an affine space that well approximates the training set. You can leave the number of principal components, $k$, as a parameter; around $k = 100$ gives good results.

3. Project both the training images and testing images onto the affine space, using the lower dimensional coordinates of the affine space. If the testing (or training) images are stored in a matrix $X$, where each row is a flattened image, and $\bar{x}$ is the mean face image computed by PCA in Step 1 (as a row vector), then the PCA coordinates are given by

$$ (X - \bar{x})P, $$

where $P$ is the matrix containing the principle components as columns. Then match each testing image to the training image that is closest in Euclidean distance in the PCA coordinates.

4. You may not get very good results with the method above (around 60% accuracy). Try using the Mahalanobis distance instead of Euclidean distance to match faces in the PCA coordinates. For two vectors $x$ and $y$ of length $k$, representing the PCA coordinates of two images, the Mahalanobis distance is given by

$$ d_M(x, y) = \sum_{i=1}^{k} \lambda_i^{-1}(x(i) - y(i))^2, $$

where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$ are the eigenvalues of the covariance matrix (obtained by PCA in Step 1). The Mahalanobis distance rescales the
norm along each coordinate to match the variation in the data in the corresponding principal direction. You should be able to get around 90% accuracy with the Mahalanobis distance, with some variation depending on the random training/testing split.

\[ \Delta \]

4 Clustering

We now turn to the problem of clustering, or grouping, data. Figure 4.1 shows a sample dataset consisting of 500 points that appear to belong to three distinct clusters. Two of the clusters are very close together, compared to the third, which more isolated. The goal of clustering is to separate the points in Figure 4.1 into the three natural clusters. In general, when working with real data and not synthetic examples, it is difficult to visualize the “natural clusters” within data, and it can be difficult to define what constitutes a good or bad clustering (since there are various natural ways one can group data).

In this section we will study two popular and widely used algorithms for clustering: \( k \)-means clustering, and spectral clustering. We will study the mathematics behind each algorithm, to the extent that we can in this course, and highlight the advantages and drawbacks of each method. We will also give applications to real data, by considering the problem of clustering pairs of digits from the MNIST dataset.
4.1 $k$-Means Clustering

**Python Notebook: .ipynb**

The $k$-means algorithm aims to find a single good representative point from each of $k$ clusters. The dataset is then clustered into $k$ groups by assigning each datapoint to the cluster corresponding to the closest such representative point in the Euclidean distance. To describe the setting mathematically, let $x_1, x_2, \ldots, x_m$ by a dataset consisting of $m$ points in $\mathbb{R}^n$. Let $c_1, c_2, \ldots, c_k$ be the cluster centers, which are vectors in $\mathbb{R}^n$ and are yet to be determined. The $k$-means algorithm is guided by the task of minimizing the $k$-means clustering energy

$$E(c_1, c_2, \ldots, c_k) = \sum_{i=1}^{m} \min_{1 \leq j \leq k} \| x_i - c_j \|^2.$$  

Minimizing $E$ over the cluster centers (also called “means”) $c_1, \ldots, c_k$ aims to find $k$ points that well-represent the dataset, in the sense that all points are close to at least one $c_j$ in the squared Euclidean distance. If we are able to minimize $E$, then the $j^{\text{th}}$ cluster in the dataset, denoted $\Omega_j$, consists of all points $x_i$ that are closer to $c_j$ than they are to any other cluster center. That is

$$\Omega_j = \left\{ x_i : \| x_i - c_j \|^2 = \min_{1 \leq \ell \leq k} \| x_i - c_\ell \|^2 \right\}.$$  

It turns out that minimizing the $k$-means clustering energy $E$ is very hard computationally (it has been shown to be NP-hard). However, it is possible to construct a simple algorithm that descends on the energy $E$, is provably convergent (to a local minimizer), and often gives good clustering results. This is called the $k$-means algorithm, and is outlined below.

**$k$-means algorithm:** We start with some randomized initial values for the means $c_1^0, c_2^0, \ldots, c_k^0$, and iterate the steps below until convergence.

1. Update the clusters

$$\Omega_j^t = \left\{ x_i : \| x_i - c_j^t \|^2 = \min_{1 \leq \ell \leq k} \| x_i - c_\ell^t \|^2 \right\}.$$  

2. Update the cluster centers

$$c_j^{t+1} = \frac{1}{\#\Omega_j^t} \sum_{x \in \Omega_j^t} x.$$  

37
Figure 4.2: An illustration of the intermediate steps in the $k$-means clustering algorithm. The red points are the cluster centroids. The algorithm converged in 8 steps, but steps 6, 7, 8 showed very little change in the clustering.

Above, the notation $\#\Omega^t_j$ denotes the number of points in the $j^{th}$ cluster $\Omega^t_j$ at the $t^{th}$ step in the algorithm. Hence, $c_j^{t+1}$ is exactly the mean of the $j^{th}$ cluster $\Omega^t_j$. The $k$-means algorithm generates a sequence of clusterings $\Omega^0_j, \Omega^1_j, \Omega^2_j, \ldots$ and cluster means $c_j^0, c_j^1, c_j^2, \ldots$, for $j = 1, \ldots, k$, that get progressively better in the sense that the $k$-means clustering energy (4.1) is decreasing (which we prove below). The algorithm converges when the clusters (and hence the cluster means) do not change from one iteration to the next, that is $\Omega^t_j = \Omega^{t+1}_j$ for all $j = 1, \ldots, k$. We also note that a point $x_i$ may be equally close to more than one cluster center, and in this case we can make any reasonable choice of which cluster to assign it to, such as the cluster whose index is smallest.

We show in Figure 4.2 an illustration of some of the intermediate steps in applying $k$-means clustering with $k = 3$ (i.e., the 3-means clustering algorithm) to the point cloud from Figure 4.1. The algorithm converged in 8 steps to a good clustering, although this depends on the randomized initial condition. For some initializations the algorithm converged in fewer iterations, sometimes as few as three, while for other initializations the algorithm took longer to converge. The clustering obtained can also depend on the initial condition. We show in Figure 4.3 two examples of poor clusterings obtained by 3-means
clustering of the same point cloud.

The \( k \)-means algorithm often gives good results, but it does not find a global minimizer of the \( k \)-means clustering energy \( E \) defined in (4.1), and can converge to local minimizers that give poor clustering results. Nevertheless, we can prove that the \( k \)-means algorithm descends on the energy \( E \), and converges in a finite number of iterations. The proof requires a preliminary lemma, which shows that the centroid (or mean) minimizes the sum of squared distances to the cluster center.

**Lemma 4.1.** Let \( y_1, y_2, \ldots, y_m \) be points in \( \mathbb{R}^n \), and define the function \( f : \mathbb{R}^n \to \mathbb{R} \) by

\[
f(x) = \sum_{i=1}^{m} \| y_i - x \|^2.
\]

Then the unique minimizer of \( f \) is the centroid

\[
c = \frac{1}{m} \sum_{i=1}^{m} y_i.
\]

**Proof.** First, we claim that

(4.4) \[
f(c) = \sum_{i=1}^{m} (\| y_i \|^2 - \| c \|^2).
\]

To see this, we first compute

\[
\| c \|^2 = c^T c = \left( \frac{1}{m} \sum_{i=1}^{m} y_i \right)^T \left( \frac{1}{m} \sum_{j=1}^{m} y_j \right) = \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} y_i^T y_j.
\]
From this, it follows that

\[
    f(c) = \sum_{i=1}^{m} \|y_i - c\|^2
\]

\[
    = \sum_{i=1}^{m} \left( \|y_i\|^2 - 2y_i^T \frac{1}{m} \sum_{j=1}^{m} y_j + \|c\|^2 \right)
\]

\[
    = \sum_{i=1}^{m} \|y_i\|^2 - \frac{2}{m} \sum_{i=1}^{m} \sum_{j=1}^{m} y_i^T y_j + m\|c\|^2
\]

\[
    = \sum_{i=1}^{m} \|y_i\|^2 - 2m\|c\|^2 + m\|c\|^2
\]

\[
    = \sum_{i=1}^{m} \|y_i\|^2 - m\|c\|^2 = \sum_{i=1}^{m} (\|y_i\|^2 - \|c\|^2),
\]

which establishes the claim.

Now, using (4.4) we obtain

\[
f(z) = \sum_{i=1}^{m} \|y_i - z\|^2
\]

\[
    = \sum_{i=1}^{m} \left( \|y_i\|^2 - 2z^T y_i + \|z\|^2 \right)
\]

\[
    = \sum_{i=1}^{m} (\|y_i\|^2 - \|c\|^2) + \sum_{i=1}^{m} (\|c\|^2 - 2z^T y_i + \|z\|^2)
\]

\[
    = f(c) + m\|c\|^2 - 2mz^T c + m\|z\|^2
\]

\[
    = f(c) + m\|c - z\|^2
\]

for any \(z \in \mathbb{R}^n\). Therefore \(f(c) \leq f(z)\) for all \(z \in \mathbb{R}^n\), with equality if and only if \(z = c\). This completes the proof. \(\square\)

We can now prove convergence of the \(k\)-means algorithm.

**Theorem 4.2.** The \(k\)-means algorithm descends on the energy (4.1), that is

(4.5) \[
    E(c_1^{t+1}, c_2^{t+1}, \ldots, c_k^{t+1}) \leq E(c_1^t, c_2^t, \ldots, c_k^t).\]

*Furthermore, we have equality in (4.5) if and only if \(c_j^{t+1} = c_j^t\) for \(j = 1, \ldots, k\), and hence the \(k\)-means algorithm converges in a finite number of iterations.*
Proof. The proof is based on re-writing the $k$-means energy in the following way:

$$E(c^t_1, c^t_2, \ldots, c^t_k) = \sum_{j=1}^{k} \sum_{x \in \Omega^t_j} \|x - c^t_j\|^2.$$  

This follows from the definition of the clusters $\Omega^t_j$ defined in (4.2). Now, it follows from Lemma 4.1 that

$$\sum_{x \in \Omega^t_j} \|x - c^{t+1}_j\|^2 \leq \sum_{x \in \Omega^t_j} \|x - c^t_j\|^2$$

with equality if and only if $c^{t+1}_j = c^t_j$, due to the definition of $c^{t+1}_j$ in (4.3) as the mean of the $j^{th}$ cluster $\Omega^t_j$ at step $t$. Therefore

$$\sum_{j=1}^{k} \sum_{x \in \Omega^t_j} \|x - c^{t+1}_j\|^2 \leq E(c^t_1, c^t_2, \ldots, c^t_k)$$

with equality if and only if $c^{t+1}_j = c^t_j$ for $j = 1, \ldots, k$. Since the $k$-means energy uses the squared distance to the closest cluster center, we trivially have

$$E(c^{t+1}_1, c^{t+1}_2, \ldots, c^{t+1}_k) = \sum_{i=1}^{m} \min_{1 \leq j \leq k} \|x_i - c^{t+1}_j\|^2$$

$$\leq \sum_{j=1}^{k} \sum_{x \in \Omega^t_j} \|x - c^{t+1}_j\|^2$$

$$\leq E(c^t_1, c^t_2, \ldots, c^t_k),$$

again with equality if and only if $c^{t+1}_j = c^t_j$ for $j = 1, \ldots, k$.

We now show that this implies convergence of the $k$-means algorithm. Note that if $c^{t+1}_j \neq c^t_j$ for some $j$ (so the algorithm has not converged), then we proved above that the energy is strictly decreasing, so

$$E(c^{t+1}_1, c^{t+1}_2, \ldots, c^{t+1}_k) < E(c^t_1, c^t_2, \ldots, c^t_k).$$

Hence, prior to convergence, we can never revisit the same clustering

$$\Omega^t_1, \Omega^t_2, \ldots, \Omega^t_k$$

at any step in the $k$-means algorithm. Indeed, if we were to revisit the same clustering at some future step of the algorithm, then since the means
$c_{1}^{t+1}, c_{2}^{t+1}, \ldots, c_{k}^{t+1}$ depend only on the clusters $\Omega_{1}^{t}, \Omega_{2}^{t}, \ldots, \Omega_{k}^{t},$ we would revisit the same configuration of cluster centers, which is impossible since the $k$-means energy is strictly decreasing with each iteration prior to convergence. Since there are only a finite number of possible ways to cluster the dataset into $k$ groups, and the $k$-means algorithm cannot revisit any given clustering, the algorithm must eventually converge.

Remark 4.3. The proof of Theorem 4.2 uses strict energy monotonicity to prove convergence. The proof is non-quantitative, meaning it does not say anything about how many iterations the $k$-means algorithm may take to converge. In practice, the algorithm often converges very quickly, in only a handful of iterations, but it is possible for $k$-means to take substantially longer to converge. Indeed, in the worst case, the algorithm must visit every possible clustering before converging. Even for the 2-means problem with $n$ points, there $2^n$ possible ways to cluster the data, so the search space is exponentially large.

We also remark that convergence of the $k$-means algorithm simply means that the cluster centers stop changing from one iteration to the next. This does not mean the algorithm has converged to a minimizer of the $k$-means energy (4.1), and in general the algorithm does not find global minimizers. In fact, due to the random choice of initialization, the algorithm can find different clusterings every time it is executed.

Exercise 4.4 (Robust $k$-means clustering). The $k$-means clustering algorithm is sensitive to outliers, since it uses the squared Euclidean distance. We consider the robust $k$-means energy

$$E_{\text{robust}}(c_1, c_2, \ldots, c_k) = \sum_{i=1}^{m} \min_{1 \leq j \leq k} \|x_i - c_j\|.$$  

The robust $k$-means algorithm attempts to minimize (4.6). We start with some randomized initial values for the means $c_{1}^{0}, c_{2}^{0}, \ldots, c_{k}^{0},$ and iterate the steps below until convergence.

1. Update the clusters as in (4.2).

2. Update the cluster centers

$$c_{j}^{t+1} \in \arg \min_{y \in \mathbb{R}^n} \sum_{x \in \Omega_{j}^{t}} \|x - y\|.$$  

Complete the following exercises.
(i) Show that the Robust PCA algorithm descends on the energy $E_{\text{robust}}$.

(ii) The cluster center (4.7) does not admit a closed form expression and is sometimes inconvenient to work with in practice. Consider changing the Euclidean norm in (4.6) to the $\ell^1$-norm $\|x\|_1 = \sum_{i=1}^n |x(i)|$, and define

$$E_{\ell^1}(c_1, c_2, \ldots, c_k) = \sum_{i=1}^m \min_{1 \leq j \leq k} \|x_i - c_j\|_1.$$ 

Formulate both steps of the $k$-means algorithm so that it descends on $E_{\ell^1}$. Show that the cluster centers $c_j^{t+1}$ are the coordinatewise medians of the points $x \in \Omega_j^t$, which are simple to compute.

(iii) Can you think of any reasons why the Euclidean norm would be preferred over the $\ell^1$ norm in the $k$-means energy?

\[\triangle\]

Exercise 4.5 (Optimal clustering in 1D). We consider here the 2-means clustering algorithm in dimension $n = 1$. Let $x_1, x_2, \ldots, x_m \in \mathbb{R}$ and recall the 2-means energy is

$$E(c_1, c_2) = \sum_{i=1}^m \min \{(x_i - c_1)^2, (x_i - c_2)^2\}.$$ 

Throughout the question we assume that the $x_i$ are ordered so that

$$x_1 \leq x_2 \leq \cdots \leq x_m.$$

For $1 \leq j \leq m - 1$ we define

$$\mu^-(j) = \frac{1}{j} \sum_{i=1}^j x_i, \quad \mu^+(j) = \frac{1}{m-j} \sum_{i=j+1}^m x_i,$$

and

$$F(j) = \sum_{i=1}^j (x_i - \mu^-(j))^2 + \sum_{i=j+1}^m (x_i - \mu^+(j))^2.$$ 

(i) Let $c_1 < c_2$ such that $x_1 \leq \frac{1}{2}(c_1 + c_2) \leq x_m$. Show that there exists $1 \leq j \leq m - 1$ such that

$$F(j) \leq E(c_1, c_2).$$
(ii) This suggests that minimizing \( F(j) \) over \( j = 1, \ldots, m - 1 \) is also a reasonable clustering objective (though not exactly the same as 2-means). Explain how you can find the global minimum of \( F(j) \) in \( O(m^2) \) floating point operations.

(iii) We can actually minimize \( F \) in \( O(m \log m) \) time. To see this, show that

\[
\mu^-(j + 1) = \frac{j}{j + 1} \mu^-(j) + \frac{x_{j+1}}{j + 1},
\]

\[
\mu^+(j + 1) = \frac{m - j}{m - j - 1} \mu^+(j) - \frac{x_{j+1}}{j + 1},
\]

and

\[
F(j) = j \mu^-(j)^2 + (m - j) \mu^+(j)^2 - 2 \mu^-(j) \sum_{i=1}^{j} x_i - 2 \mu^+(j) \sum_{i=j+1}^{m} x_i + \sum_{i=1}^{m} x_i^2.
\]

Explain how you can use the formulas above to compute \( F(j + 1) \) from \( F(j) \) in only \( O(1) \) operations. Thus, you can compute all \( F(j) \), for \( j = 1, \ldots, m \), in \( O(m) \) time, given the \( x_i \) are already sorted, which takes \( O(m \log m) \) time.

4.1.1 Clustering MNIST digits

Python Notebook: .ipynb

We now consider a brief application of \( k \)-means clustering to real data. Again, we use the MNIST dataset of handwritten digits, and we evaluate the 2-means algorithm for clustering pairs of MNIST digits. We consider all pairs of MNIST digits (there are around 7000 images per digit), yielding \( \binom{10}{2} \) binary clustering problems, each with around 14000 datapoints in dimension \( \mathbb{R}^{784} = \mathbb{R}^{28 \times 28} \). The \( k \)-means algorithm converged very quickly, in around 15 iterations taking around 1 second per clustering problem. Table 4.1.1 shows the clustering accuracy obtained by the 2-means algorithm for each pair of MNIST digits. The numbers can vary depending on the random choice of initial condition. We can see that many pairs of digits are very easy to cluster into the correct classes with the 2-means algorithm, while a handful of pairs of digits, such as (4,9), (5,3), (7,9), (5,8), and (8,9) are more difficult. It is also natural to run the 10-means algorithm on the whole MNIST dataset, but the algorithm takes a long time to converge and generally gives poor clustering results.
Table 4.1: Accuracy for binary (2-means) clustering of pairs of MNIST digits. We see most pairs of digits are easy to separate, while a few pairs, such as (4,9), (5,3), (7,9), (5,8), and (8,9), are more difficult.

<table>
<thead>
<tr>
<th>Digit</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99.1</td>
<td>95.0</td>
<td>95.0</td>
<td>96.5</td>
<td>86.8</td>
<td>92.9</td>
<td>96.9</td>
<td>94.5</td>
<td>95.6</td>
</tr>
<tr>
<td>1</td>
<td>92.7</td>
<td>96.1</td>
<td>97.9</td>
<td>91.1</td>
<td>96.4</td>
<td>95.8</td>
<td>94.3</td>
<td>97.2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>89.7</td>
<td>95.7</td>
<td>94.6</td>
<td>93.8</td>
<td>95.6</td>
<td>91.2</td>
<td>95.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>97.3</td>
<td>66.3</td>
<td>97.6</td>
<td>96.4</td>
<td>80.0</td>
<td>94.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>88.2</td>
<td>95.6</td>
<td>95.3</td>
<td>95.7</td>
<td>52.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>91.4</td>
<td>87.2</td>
<td>52.3</td>
<td>94.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>99.1</td>
<td>96.6</td>
<td>98.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>95.9</td>
<td>60.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2 Spectral Clustering

Python Notebook: .ipynb

The $k$-means clustering algorithm works well for clusters that are roughly spherical (e.g., blob data). For clusters with more complicated geometries, a single cluster center may not be a good representative of the whole cluster, and Euclidean distance to cluster centers may not be a good indication
of which cluster a datapoint belongs to. We show an example of this on the famous 2-moons dataset in Figure 4.4. This dataset has two clusters that have a nonlinear, and non-convex, shape, and are interleaved so that a single there are no good choice of Euclidean cluster centers for either cluster. In this case, 2-means clustering performs poorly. In this section we will study another algorithm, called spectral clustering, which can uncover more complicated cluster geometries.

Spectral clustering takes a different perspective on clustering, compared to k-means clustering. In k-means clustering we attempted to find k cluster centers, and clustered the data by finding, for each datapoint, which cluster center is closest in Euclidean distance. In contrast, spectral clustering aims to ensure that points that are nearby spatially are assigned to the same cluster. It does not compute cluster centers, and makes no assumption on the shape or geometry of the clusters.

Spectral cluster requires the computation of an adjacency matrix, or weight matrix, for the dataset, which encodes the similarities between pairs of points. If our dataset consists of m points \( x_1, x_2, \ldots, x_m \) in \( \mathbb{R}^n \), the weight matrix \( W \) is an \( m \times m \) symmetric matrix, where the entry \( W(i,j) \) represents the similarity between datapoints \( x_i \) and \( x_j \). The similarity is always nonnegative \( (W(i,j) \geq 0) \), and should be large when \( x_i \) and \( x_j \) are close together spatially, and small (or zero), when \( x_i \) and \( x_j \) are far apart. A common choice for the weight matrix is Gaussian weights

\[
W(i, j) = \exp \left( -\frac{\|x_i - x_j\|^2}{2\sigma^2} \right),
\]

where the \( \sigma \) is a free parameter that controls the scale at which points are connected by strong similarities \( W(i,j) \) in the weight matrix. In fact, the weight matrix \( W \) endows the dataset with a graph structure, where each pair of points \( (x_i, x_j) \) is connected by an edge with edge weight \( W(i,j) \). Other choices of weight matrix are possible, such as the k-nearest neighbor graph explored in Section 4.2.2, so we proceed with a general nonnegative and symmetric \( m \times m \) weight matrix \( W \) in what follows.

While spectral clustering can be applied to general problems with k clusters, it is simplest to explain for the binary clustering problem, where we seek \( k = 2 \) clusters. Let \( z \in \{0, 1\}^m \) be a vector giving the cluster labels for each datapoint, so \( z(i) \in \{0, 1\} \) is the cluster label for \( x_i \). Here, we have assigned one cluster to have the label 0 and the other cluster the label 1. Given a weight matrix \( W \), we wish to find a binary clustering for which \( x_i \) and \( x_j \) are assigned to the same cluster when the weight \( W(i,j) \) is large (since this means
the points are nearby). It is thus natural to consider the graph cut energy

\[ E(z) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j) |z(i) - z(j)|^2. \]  

The graph cut energy sums the edge weights \( W(i,j) \) corresponding to pairs of points \((x_i, x_j)\) that are assigned to different clusters, so that \( z(i) \neq z(j) \). A graph cut approach to clustering amounts to attempting to minimize \( E \) over cluster labels \( z \), which attempts to ensure that similar datapoints, where \( W(i,j) \) is large, get assigned to the same cluster. In the presence of outliers, this can give poor clusterings, since it tends to put a single outlier in one class, and all other datapoints in the other class. Thus, it is more common to consider a balanced graph-cut energy, which penalizes unbalanced clusters. One possibility is the balanced energy

\[ E_{\text{balanced}}(z) = \frac{1}{2} \frac{\sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j) |z(i) - z(j)|^2}{\sum_{i=1}^{m} z(i) \sum_{j=1}^{m} (1 - z(j))}. \]

The denominator in the balanced energy is the product of the sizes of the two clusters, which is largest when the classes are balanced. Minimizing a balanced graph-cut energy gives much better clusterings, but it turns out to be a very hard problem computationally (in fact, it is NP hard).

Spectral clustering is a tractable relaxation of the graph cut problem described above. Instead of enforcing that the cluster labels \( z(i) \) belong to the discrete set \( \{0, 1\} \), we relax the problem and allow \( z \in \mathbb{R}^m \) to be any real-valued vector. This introduces the problem that any constant vector \( z = t1 \) is a minimizer of the graph-cut energy \( E \), where \( t \in \mathbb{R} \) and \( 1 \) is the all ones vector. We could instead consider minimizing the balanced energy (4.10), but it is more convenient to use slightly different balancing conditions. The two natural balancing conditions used in spectral clustering are

\[ 1^T z = \sum_{i=1}^{m} z_i = 0 \quad \text{and} \quad \|z\|^2 = \sum_{i=1}^{m} z(i)^2 = 1. \]

The two constraints in (4.11), taken together, ensure that the any minimizer \( z \) of \( E(z) \) is not a trivial constant labeling \( z = t1 \), and that the classes are balanced to some degree. This leads to the binary spectral clustering problem:

\[ \text{Minimize } E(z) \text{ over } z \in \mathbb{R}^m \text{ subject to (4.11)}. \]

After finding a minimizer \( z^* \), the clustering is obtained by the sign of the minimizer, so \( \{x_i : z^*(i) > 0\} \) is one cluster, while \( \{x_i : z^*(i) \leq 0\} \) is the other cluster.
4.2.1 The graph Laplacian and Fiedler vector

We now show how to solve the binary spectral clustering problem (4.12), which will explain the spectral part of the name.

**Definition 4.6.** Let $W$ be a symmetric $m \times m$ matrix with nonnegative entries. The associated graph Laplacian matrix $L$ is the $m \times m$ matrix

\[(4.13) \quad L = D - W\]

where $D$ is the diagonal matrix with diagonal entries

\[D(i,i) = \sum_{j=1}^{m} W(i,j).\]

We now show that the graph cut energy can be rewritten in terms of the graph Laplacian matrix $L$.

**Lemma 4.7.** Let $W$ be a symmetric $m \times m$ matrix with nonnegative entries. Then the graph cut energy $E$ defined in (4.9) can be expressed as

\[(4.14) \quad E(z) = z^T L z,\]

where $L = D - W$ is the graph Laplacian matrix.

**Proof.** We simply compute

\[
E(z) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)|z_i - z_j|^2 \\
= \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)(z_i^2 - 2z_i z_j + z_j^2) \\
= \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)z_i^2 - \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)z_i z_j + \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)z_j^2 \\
= \frac{1}{2} \sum_{i=1}^{m} D(i,i)z_i^2 - \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)z_i z_j + \frac{1}{2} \sum_{j=1}^{m} D(j,j)z_j^2 \\
= \sum_{i=1}^{m} D(i,i)z_i^2 - \sum_{i=1}^{m} \sum_{j=1}^{m} W(i,j)z_i z_j \\
= z^T Dz - z^T Wz = z^T L z,
\]

which completes the proof. \(\square\)
We recall from Exercise 2.5 that minimizing the quantity $z^T L z$, subject to the constraint $\|z\| = 1$, is related to finding eigenvectors of $L$. Due to (4.11) we have an additional constraint $1^T z = 0$ that needs to be accounted for. Before proceeding, we record some basic properties of the graph Laplacian.

**Lemma 4.8.** Let $L = D - W$ be the graph Laplacian corresponding to a symmetric matrix $W$ with nonnegative entries. The following properties hold.

(i) $L$ is symmetric.

(ii) $L$ is positive semi-definite (i.e., $z^T L z \geq 0$ for all $z \in \mathbb{R}^m$).

(iii) All eigenvalues of $L$ are nonnegative, and the constant vector $z = 1$ is an eigenvector of $L$ with eigenvalue $\lambda = 0$.

**Proof.** (i) Both $D$ and $W$ are symmetric, so $L$ is as well.

(ii) By Lemma 4.7 we have

$$z^T L z = E(z) \geq 0,$$

for any $z \in \mathbb{R}^m$, thus $L$ is positive semi-definite.

(iii) By Exercise 2.6, all eigenvalues of $L$ are nonnegative, since $L$ is positive semi-definite. To check that $z = 1$ belongs to the kernel of $L$, let $y = L 1 = D 1 - W 1$, and check that

$$y(i) = D(i, i) - \sum_{j=1}^m W(i, j) = 0$$

for all $i = 1, \ldots, m$. This completes the proof. \qed

Since $L$ is a real symmetric matrix, it is diagonalizable, and there exists an orthonormal basis $v_1, v_2, \ldots, v_m$ of $\mathbb{R}^m$ consisting of eigenvectors of $L$, with corresponding eigenvalues

$$0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_m.$$

That is, $L v_i = \lambda_i v_i$. Note that $\lambda_1 = 0$ by Lemma 4.8 (iii).

**Definition 4.9.** The second eigenvector $v_2$ of the graph Laplacian $L$ is called the *Fiedler vector*.
The Fiedler vector is the first non-trivial eigenvector of $L$, when they are ordered by increasing eigenvalue. The first eigenvector is constant, $v_1 = \frac{1}{\sqrt{m}} 1$, and is considered trivial.

It turns out, due to the theorem below, that the Fiedler vector is exactly the solution of the binary spectral clustering problem (4.12). Thus, spectral clustering amounts to assigning points to one of two clusters depending on the sign of the Fiedler vector. This is also the reason for the name spectral clustering—we are using the spectrum (i.e., the eigenvectors) of the graph Laplacian matrix for clustering.

**Theorem 4.10.** The Fiedler vector $v_2$ solves the spectral clustering problem (4.12).

**Proof.** A minimizer of (4.12) exists, due to the fact that we are minimizing a continuous function $E(z)$ over a closed and bounded set

$$\{ z \in \mathbb{R}^m : \|z\| = 1 \text{ and } 1^T z = 0 \}.$$ 

Let $z$ be a minimizer of (4.12), and write $z$ in the basis of eigenvectors of $L$ as

$$z = \sum_{i=1}^{m} a_i v_i.$$ 

Since $\|z\| = 1$ we have by (2.1) that

$$\sum_{i=1}^{m} a_i^2 = 1.$$ 

Since $v_1 = \frac{1}{\sqrt{m}} 1$, and the $v_i$ are orthonormal vectors, the condition $1^T z = 0$ is equivalent to

$$0 = 1^T z = \sqrt{m} v_1^T \sum_{i=1}^{m} a_i v_i = \sqrt{m} \sum_{i=1}^{m} a_i v_1^T v_i = \sqrt{m} a_1.$$ 

Therefore $a_1 = 0$. We now compute, by Lemma 4.7, that

$$E(z) = z^T L z = z^T L \sum_{i=2}^{m} a_i v_i = \sum_{i=2}^{m} a_i z^T L v_i = \sum_{i=2}^{m} \lambda_i a_i z^T v_i = \sum_{i=2}^{m} \lambda_i a_i^2.$$ 

since $z^T v_i = a_i$. Since $\lambda_2 \leq \lambda_3 \leq \cdots \leq \lambda_m$, $a_1 = 0$, and (4.15) holds, we have

$$E(z) = \sum_{i=2}^{m} \lambda_i a_i^2 \geq \lambda_1 \sum_{i=2}^{m} a_i^2 = \lambda_1.$$ 

Therefore, setting $a_2 = 1$ and $a_i = 0$ for $i \neq 1$ minimizes $E(z)$. This amounts to $z = v_2$, which completes the proof. \qed
We return briefly to the two-moons clustering problem from Figure 4.4, which was not clustered correctly by \( k \)-means. We applied spectral clustering with Gaussian weights \( W(i,j) \) (see (4.8)) with \( \sigma = 0.15 \). Figure 5(a) shows the Fiedler vector colored with lowest values dark blue, and highest values yellow. Figure 5(b) shows the result of spectral clustering on the two-moons dataset, where the clusters are based on the sign of the Fiedler vector. Spectral clustering finds the correct clustering for the two-moons dataset, which illustrates the ability of spectral clustering to handle more complicated nonlinear cluster geometry.

**Remark 4.11.** We briefly remark that spectral clustering is not only used for binary clustering, and there are versions of the algorithm that work for \( k \) classes. The general idea is to compute the first \( k \) eigenvectors \( v_1, v_2, \ldots, v_k \) of the graph Laplacian \( L = D - W \) and to perform a spectral embedding of the data into \( \mathbb{R}^k \). The spectral embedding \( \Phi_k : \mathbb{R}^n \to \mathbb{R}^k \) is

\[
\Phi_k(x_i) = (v_1(i), v_2(i), \ldots, v_k(i)).
\]

That is, at each datapoint \( x_i \), we evaluate all \( k \) eigenvectors of the graph Laplacian and those values are the coordinates of the points \( x_i \) in the \( k \)-dimensional embedded space. This is a very useful method for dimension reduction. The data is then clustered in the spectral embedding space \( \mathbb{R}^k \), often using the \( k \)-means clustering algorithm. There are several different variants of spectral clustering for more than two classes, based on using different normalizations for the graph Laplacian (two common normalizations are \( D^{-1}L \) and \( D^{-1/2}LD^{-1/2} \)), and on using further normalizations of the points in the em-
Table 4.2: Accuracy for binary spectral clustering of pairs of MNIST digits. The results are generally an improvement over 2-means clustering (see Table 4.1.1), but we still see some pairs of digits, such as (4,9) and (7,9) are hard to separate.

bedded space $\mathbb{R}^k$. We refer to [9] for more information on spectral clustering.

### 4.2.2 Clustering MNIST digits

**Python Notebook: .ipynb**

We now return to the problem of clustering MNIST digits from Section 4.1.1, to see if we can improve upon the results of $k$-means clustering. The choice of Gaussian weights (4.8) is not practical for real data for a couple of reasons. First, it results in a dense matrix $W$, where all entries need to be stored, which is not tractable for large datasets. Second, there are difficulties using the same scale $\sigma$ over the whole graph, since some areas of the graph may be very dense and a smaller connectivity scale $\sigma$ would be appropriate, while some areas may be more sparse and require a larger $\sigma$ to ensure they are well-connected to neighboring points in the graph.

A better way to construct a graph over a dataset in practice is to build a $k$-nearest neighbor graph. To do this, we find for each $x_i$, the $k$ nearest points $x_j$ in the Euclidean distance, and we assign positive edge weights $W(i, j)$ only for these $k$-nearest neighbors. All other weights are zero, and the matrix is stored in a sparse matrix format so the zero entries do not need to be stored in memory, nor used in computations. The weights can be defined as $W(i, j) = 1$
when \( j \) is a \( k \)-nearest neighbor of \( i \), or we can use Gaussian weights with a length scale that adapts to the graph. In this experiment we used

\[
W(i, j) = \exp \left( -\frac{4\|x_i - x_j\|^2}{d_k(x_i)^2} \right),
\]

where \( d_k(x_i) \) is the Euclidean distance from \( x_i \) to its \( k \)th nearest neighbor.

A minor problem with \( k \)-nearest neighbor graphs is that they are not symmetric, that is \( W(i, j) \neq W(j, i) \). This means the graph Laplacian matrix \( L = D - W \) will not be symmetric, and hence it may not be diagonalizable, the spectrum may be complex-valued, and the Fiedler vector will have no meaning. Thus, we always symmetrize the weight matrix for a \( k \)-nearest neighbor graph in some reasonable way before applying spectral clustering. In this experiment, we replace \( W \) with \( \frac{1}{2}(W + W^T) \) to symmetrize. Other choices are possible, such as symmetrizing the weights directly

\[
W(i, j) = \exp \left( -\frac{4\|x_i - x_j\|^2}{d_k(x_i)d_k(x_j)} \right).
\]

We show in Table 4.2.2 the results of binary spectral clustering of all pairs of MNIST digits using a \( k = 10 \) nearest neighbor graph. We generally see a good improvement over the corresponding results for the \( k \)-means algorithm given in Table 4.1.1. However, there are still some pairs of digits that are difficult to separate, such as (4, 9) and (7, 9). We will return to this example later in the lecture notes, and we will see how the results can be further improved by using neural network autoencoders.

5 PageRank

Python Notebook: .ipynb

Having introduced graph-based methods briefly in Section 4.2, we turn here to study the PageRank algorithm, which was used by Google to rank internet search results until around 2006. While PageRank originally gained popularity as Google’s search engine, it has found a wide range of applications in other fields, including biology (GeneRank), chemistry, ecology, neuroscience, physics, sports, and computer systems. We refer to [6] for a review of PageRank and a summary of the abundance of applications outside of search engines.

PageRank is a graph-based ranking algorithm that ranks websites based on their link structure. The idea behind the PageRank algorithm is to take a
random walk on the internet for a long time (by randomly following links on each webpage visited) and record how often each website is visited along the way. The websites that are visited more often get higher ranks than those that are visited less often. While this is the core of the idea behind PageRank, there is a small problem with a simple random walk on a graph like the internet—the walker may get stuck in small disconnected components of the graph that have no outgoing links. The walker will then spend all of its time in a small part of the internet (which may be very sensitive to where the walk starts), and will be unable to rank the majority sites.

To resolve this issue, PageRank uses a random surfer, which is a random walker that at random times teleports to a completely random website, one that is not necessarily connected to the current website by a link. This allows the surfer to escape poorly connected pockets of the internet and produce a ranking for all pages. Furthermore, specific choices of the type of teleportation lead to localized or personalized versions of PageRank, which are discussed in Section 5.2.

To describe the PageRank algorithm mathematically, the starting point is an adjacency matrix $W$, which encodes the links between websites. Often the adjacency matrix is a binary matrix with

$$W(i, j) = \begin{cases} 1, & \text{if site } i \text{ links to site } j \\ 0, & \text{otherwise.} \end{cases}$$

The adjacency matrix $W$ is an $n \times n$ matrix, where $n$ is the number of webpages to rank. The matrix $W$ is a very large matrix that is usually very sparse. That is, most of the entries of $W$ are zero, since most websites link to only a small fraction of the internet. Matrix operations with $W$ are thus tractable if one stores the matrix in a sparse format, so that the zero entries are not stored in memory, and not used in matrix/vector multiplications.

To define a random walk on a graph, we need to define a probability transition matrix $P$, which is an $n \times n$ matrix whose $(i, j)$ entry is the one-step transition probability

$$P(i, j) = \text{Probability of stepping from } j \text{ to } i.$$ 

Clicking on a link at random from webpage $j$ leads to the transition probabilities

$$P(i, j) = \frac{W(j, i)}{\sum_{k=1}^{n} W(j, k)}.$$
The columns of $P$ are probability distributions, and hence for all $j$ we have

$$
\sum_{i=1}^{n} P(i, j) = 1.
$$

**Exercise 5.1.** Show that $P = W^T D^{-1}$, where $D$ is the diagonal matrix with diagonal entries $D(i, i) = \sum_{j=1}^{n} W(i, j)$.

To model the random teleportation of the surfer, we flip a biased coin at each step, and with probability $\alpha \in [0, 1)$ the surfer takes a random walk step, and with probability $1 - \alpha$ the surfer teleports somewhere else at random in the internet. The teleportation step is itself drawn at random, according to a teleportation probability distribution $v$, which satisfies $v(i) \geq 0$ and $\sum_{i=1}^{n} v(i) = 1$.

The teleportation distribution can be uniform $v(i) = 1/n$, in which case the random surfer jumps uniformly at random to another site on the internet. Another typical choice is a localized PageRank, obtained by setting $v(i) = \delta_{ij}$, where $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ otherwise. In this case, the random surfer always jumps back to the same website $j$ each time it teleports, which leads to a ranking of all pages relative to $j$. This can be useful in retrieval problems.

There are of course a continuum of intermediate teleportation distributions between the uniform random distribution and localized PageRank.

Let $x_0(i)$ denote the probability that the random surfer is initially at site $i$ at step $k = 0$ of the walk. We can take $x_0$ to be any probability vector (i.e., $x_0(i) \geq 0$ and $\sum_{i} x_0(i) = 1$). For instance, setting $x_0(i) = \delta_{ij}$ amounts to initially placing the random surfer at page $j$.

Any other distribution $x_0$ amounts to placing the surfer at node $j$ with probability $x_0(j)$ at the initial time. For $k \geq 1$ define

$$
x_k(i) = \text{Probability that the random surfer is at page } i \text{ on step } k.
$$

To see how $x_k$ transitions to $x_{k+1}$ requires some probability. We condition on the location of the surfer at step $k$, and on the outcome of the coin flip, to obtain

$$
x_{k+1}(i) = (1 - \alpha)v(i) + \alpha \sum_{j=1}^{n} P(i, j)x_k(j).
$$

We can write this in matrix/vector form as

$$
x_{k+1} = (1 - \alpha)v + \alpha Px_k.
$$

The PageRank vector is defined as the limit of $x_k$ as $k \to \infty$—we will prove the limit exists below. Figure 5.1 shows an illustration of the PageRank vector.
Figure 5.1: An illustration of the PageRank vector on a toy graph with $\alpha = 2/3$. The colors from blue to yellow indicate the value of the PageRank vector at each node, and the black arrows indicate directed links between webpages. (small values are blue and large are yellow) on a toy graph with $\alpha = 2/3$. The black arrows indicate links between websites. Notice that the two lowest ranked sites (the dark blue ones) have no incoming edges, so they are not visited except for on teleportation steps of the random surfer, which is one of the reasons for their low ranking. On the other hand, the highest ranked yellow node has many incoming edges and is visited very often by the random surfer.

5.1 Convergence of the random surfer

If we send $k \to \infty$ in the PageRank iteration (5.2), and if $x_k$ converges to some vector $x$, then clearly $x$ should be a solution of the linear equation

$$x = (1 - \alpha)v + \alpha Px.$$  

We call (5.3) the PageRank problem, and its solution $x$ is the PageRank vector. In this section, we will prove that the probability distribution $x_k$ of the random
surfer converges as $k \to \infty$ to the solution of the PageRank problem (5.3).

It will be more convenient to work in the $\ell_1$-norm $\| \cdot \|_1$ defined by

$$\| x \|_1 = \sum_{i=1}^{n} |x(i)|.$$ 

In the $\ell_1$-norm, the transition matrix $P$ is non-expansive, according to the following proposition.

**Proposition 5.2.** We have $\| Px \|_1 \leq \| x \|_1$.

**Proof.** The proof is a straightforward computation

$$\| Px \|_1 = \sum_{i=1}^{n} \left| \sum_{j=1}^{n} P(i,j)x(j) \right| \leq \sum_{i=1}^{n} \sum_{j=1}^{n} P(i,j)|x(j)| = \sum_{j=1}^{n} |x(j)| = \| x \|_1,$$

where we used (5.1) in the third step. This completes the proof. \qed

We first establish that the PageRank problem (5.2) has a unique solution.

**Lemma 5.3.** Let $v \in \mathbb{R}^n$ and $0 \leq \alpha < 1$. Then there is a unique vector $x \in \mathbb{R}^n$ solving the PageRank problem (5.3). Furthermore, the following hold.

(i) We have $\sum_{i=1}^{n} x(i) = \sum_{i=1}^{n} v(i)$.

(ii) If $v(i) \geq 0$ for all $i$, then $x(i) \geq 0$ for all $i$.

**Remark 5.4.** Taking (i) and (ii) together shows that whenever $v$ is a probability distribution (i.e., $v(i) \geq 0$ and $\sum_i v(i) = 1$), the same is true of the unique solution $x$ of the PageRank problem (5.3).

**Proof of Lemma 5.3.** Note that we can re-write (5.3) as $Ax = v$ where

$$A = (1 - \alpha)^{-1}(I - \alpha P).$$

To prove there is a unique solution $x$, we need only show that the kernel of $A$ is trivial. Then the mapping $A : \mathbb{R}^n \to \mathbb{R}^n$ is injective (one-to-one), and hence it is also surjective (onto), by the rank-nullity theorem.

To show that $\ker(A) = \{0\}$, let $z \in \ker(A)$. Then $Az = 0$ and so $z = \alpha Pz$. By Proposition 5.2 we have

$$\| z \|_1 = \| \alpha Pz \|_1 = \alpha \| Pz \|_1 \leq \alpha \| z \|_1.$$
Since $\alpha < 1$ we must have $\|z\|_1 = 0$, and so $z = 0$, which establishes that $A$ is injective, and so $Ax = v$ has a unique solution $x$, for each $v$.

We now prove (i). We use (5.1) to obtain

\[
\sum_{i=1}^{n} x(i) = \sum_{i=1}^{n} \left( (1 - \alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)x(j) \right)
\]

\[
= (1 - \alpha) \sum_{i=1}^{n} v(i) + \alpha \sum_{j=1}^{n} x(j) \sum_{i=1}^{n} P(i,j)
\]

\[
= (1 - \alpha) \sum_{i=1}^{n} v(i) + \alpha \sum_{j=1}^{n} x(j).
\]

Re-arranging we have

\[
(1 - \alpha) \sum_{i=1}^{n} x(i) = (1 - \alpha) \sum_{i=1}^{n} v(i).
\]

Since $\alpha < 1$, we can divide by $1 - \alpha$ to complete the proof of (i).

We finally prove (ii). Assume $v(i) \geq 0$ for all $i$. We first compute, using (5.3), that

\[
|x(i)| = \left| (1 - \alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)x(j) \right|
\]

\[
\leq (1 - \alpha)|v(i)| + \alpha \left| \sum_{j=1}^{n} P(i,j)x(j) \right|
\]

\[
\leq (1 - \alpha)v(i) + \alpha \sum_{j=1}^{n} P(i,j)|x(j)|.
\]

Summing both sides over $i = 1, \ldots, n$ and using (5.1) we find that

\[
\sum_{i=1}^{n} |x(i)| \leq (1 - \alpha) \sum_{i=1}^{n} v(i) + \alpha \sum_{i=1}^{n} \sum_{j=1}^{n} P(i,j)|x(j)|
\]

\[
= (1 - \alpha) \sum_{i=1}^{n} v(i) + \alpha \sum_{j=1}^{n} |x(j)|.
\]

Re-arranging and using part (i) we have

\[
(1 - \alpha) \sum_{i=1}^{n} |x(i)| \leq (1 - \alpha) \sum_{i=1}^{n} v(i) = (1 - \alpha) \sum_{i=1}^{n} x(i).
\]

58
Since $\alpha < 1$ we can divide by $1 - \alpha$ to deduce that

$$\sum_{i=1}^{n} |x(i)| \leq \sum_{i=1}^{n} x(i).$$

It follows that $x(i) = |x(i)| \geq 0$ for all $i$, which completes the proof. \hfill \Box

**Remark 5.5.** It is common to re-write the PageRank problem (5.3) as an eigenvector problem. Let us assume we are in the setting where $v$ and $x$ are probability distributions, so $x(i) \geq 0$ and $\sum_{i=1}^{n} x(i) = 1$. Noting that $1^T x = \sum_{i=1}^{n} x(i) = 1$ we can re-write (5.3) as

$$x = (1 - \alpha)v 1^T x + \alpha Px = ((1 - \alpha)v 1^T + \alpha P)x.$$

The matrix

(5.4) \hspace{1cm} P_\alpha := (1 - \alpha)v 1^T + \alpha P

is exactly the probability transition matrix for the random surfer, and the PageRank problem $P_\alpha x = x$ is an eigenvector problem, which merely states that the PageRank vector is the invariant (or stationary) distribution of the induced Markov chain. In this light, Lemma 5.3 (ii) is a direct consequence of the Perron-Frobenius Theorem.

We can now prove convergence, with a linear rate, of the random surfer’s distribution $x_k$ to the PageRank vector.

**Theorem 5.6.** Let $v \in \mathbb{R}^n$ and $0 \leq \alpha < 1$. Let $x_k$ satisfy the PageRank iteration (5.2), and let $x$ be the unique solution of the PageRank problem (5.3) (i.e., the PageRank vector). Then we have

(5.5) \hspace{1cm} \|x_k - x\|_1 \leq \alpha^k \|x_0 - x\|_1.

**Proof.** We simply subtract

$$x = (1 - \alpha)v + \alpha Px$$

from

$$x_k = (1 - \alpha)v + \alpha Px_{k-1}$$

to obtain

$$\|x_k - x\|_1 = \|\alpha Px_{k-1} - \alpha Px\| = \alpha \|P(x_{k-1} - x)\|_1.$$  

Using Proposition 5.2 we obtain

$$\|x_k - x\|_1 \leq \alpha \|x_{k-1} - x\|_1.$$  

The proof is completed by induction. \hfill \Box
Remark 5.7. Since $0 \leq \alpha < 1$, Theorem 5.6 shows that the PageRank iterations $x_k$ converge to the PageRank vector $x$ as $k \to \infty$ at the linear rate of $\alpha$ in the $\ell_1$-norm. In particular, the convergence can be very slow if $\alpha$ is close to one. This shows another advantage to introducing the teleportation step; not only does it guarantee convergence of the PageRank iterations, but it gives the user control over the convergence rate.

In practice, the standard way to compute the solution of the PageRank problem (5.3) is via the PageRank iteration (5.2), since it is simple to compute (especially with large sparse matrices), and can be terminated early to obtain an approximate solution. The PageRank iteration (5.2) can be interpreted as the power iteration for computing the largest eigenvector of a matrix. Indeed, following Remark 5.5, we can rewrite the PageRank iteration as $x_{k+1} = P_\alpha x_k$, where $P_\alpha$ is defined in (5.4). This is exactly the power iteration, except the normalization step appears to be omitted (normally it would be $x_{k+1} = P_\alpha x_k / \|P_\alpha x\|$). The normalization is not needed since $P$ is non-expansive (see Proposition 5.2).

### 5.2 Personalized PageRank for image retrieval

**Python Notebook: .ipynb**

We give here an example of using personalized, or localized, PageRank for image retrieval. Image retrieval takes a query image and tries to find similar images in a dataset. We consider the MNIST dataset of handwritten digits, and construct a $k$-nearest neighbor graph over the dataset as described in Section 4.2.2. To retrieve images similar to image $j$, we set the teleportation distribution to be $v(i) = \delta_{ij}$, in order to rank all images based on their similarity to image $j$. We then solve the PageRank problem (5.3) with the iteration (5.2), and return the top ranked images. We took one image from each of the 10 MNIST digits and ran personalized PageRank to retrieve the top 14 similar images. Figure 5.2 shows the results. The image on the left is the query image, and the following 14 images are the retrieved images. We note that most digits are from the same class, and are in fact written in a very similar way to the query digit. There are a handful retrieved digits that are from different classes than the query; for example, a 5 in the 8 row and some 4’s in in the 9 row. There is also what appears to be an 8 in the 7 row, but this is hard to distinguish by eye.
Figure 5.2: An example of using personalized PageRank for image retrieval. In each row the image on the left is the query image, and the following 14 images are the top retrieved images using personalized PageRank.

6 The Discrete Fourier Transform

We saw in Section 3.4 how to use PCA to find a good basis to represent image blocks in order to perform image compression. The basis is one for which only a few basis functions are required to reconstruct with good accuracy most of the blocks in the image. We experimented with a similar PCA-based method for audio compression in Project 3.2. In this case, the basis vectors learned by PCA are good at representing very short segments of audio files, such as music or speech. PCA finds the best such basis; the one that captures the most variation in the data with the fewest basis vectors.

It is instructive to take a closer look at these basis functions. For image compression we show the first 30 principal component vectors in Figure 3.7. These start off as low frequency images that are roughly constant over the block, and gain additional higher frequency components as we look at higher principal components. This means that most blocks in the image are well-
approximated by these slowly varying roughly constant image blocks. Similarly, for audio compression we show the first 4 principal components on length 64 blocks of an audio file in Figure 6.1. Here, some of the basis functions strongly resemble low frequency pure tones, i.e., sin and cos functions.

While PCA finds the best basis, and is adapted to the data, it is costly to compute the principal components, since this requires solving large eigenvector problems. This is impossible to do in embedded environments (e.g., a digital camera, smartphone, video surveillance, digital TVs, etc.), which have limited processing power and have to process images and video in real-time. In some settings, individual image frames in video must be processed in real-time, and only a few lines of each frame can be stored in memory. In such cases, a learned basis, such as that obtained via PCA, is intractable, and instead a hand-crafted basis with similar properties is desired.

The Fourier Transform is a hand-crafted change of basis that is extremely useful for analyzing audio, images, and video. The Fourier Transform expresses a signal (i.e., a function) in terms of a basis consisting of pure tones, that is, cos and sin waves with different shifts and of different frequencies. We will see that the Fourier basis shares many similarities with a PCA-basis, namely that images and audio signals are well-approximated locally in space/time by only a few Fourier basis functions. This is called sparsity, and audio and image
signals are locally sparse in the Fourier domain. This sparsity is what allows lossy compression algorithms, like jpeg for images and mpeg for video, to work so well. These algorithms use Discrete Fourier Transforms, instead of a PCA-based compression, and are selective about which Fourier basis function to encode, and which to discard at each instance in time or location in the image.

We will study the Discrete Fourier Transform in these notes, which applies to discretized signals (e.g., vectors), but the main ideas are similar across the different variants of the Fourier Transform. The advantage of studying the Discrete Fourier Transform is that we are studying exactly the algorithm used in practice, and we do not need to use advanced mathematical tools like measure theory and $L^p$ spaces, which are needed to study the continuous Fourier transform.

We briefly mention that there are many other advantages to a hand-crafted change of basis, like the Fourier Transform, beyond the computational advantages of not needing access to an eigensolver. The change of basis itself can be computed far more efficiently with a Fast Fourier Transform (FFT) in $O(n \log n)$ operations, compared to the matrix multiplication required for a general change of coordinates, which requires $O(n^2)$ operations. Furthermore, with PCA we did not know before hand what our basis functions would look like and could only do a limited analysis of their properties. On the other hand, for a hand-crafted transform, we know the basis functions explicitly and can perform an in-depth mathematical analysis of their properties. The Discrete Fourier Transform has many very nice mathematical properties that are worth studying on their own, independent of the applications to image or audio analysis. Furthermore, we will find that some properties of the Discrete Fourier Transform, such as its compatibility with convolutions, make it a very useful tool for solving difference equations, which are discretizations of partial differential equations. In fact, one can argue that the most important applications of the Fourier Transform are no longer in image processing or computer vision, since deep convolutional neural networks outperform Fourier (and Wavelet) methods by orders of magnitude. However, in scientific computing problems, like solving linear partial differential equations, the Fourier Transform is still one of the most useful tools, and will not be replaced anytime soon. As such, these notes will emphasize a variety of applications of Fourier and Wavelet methods, including to solving partial differential equations.

It is important to point out that we will make use of complex numbers in this section. The letter $i$ will always denote the imaginary number $i = \sqrt{-1}$, and will never be used for an index of summation, unless it is a typo. This convention is not followed in any other sections of these notes.
6.1 Complex numbers and Euler’s formula

We recall that a complex number has the form \( z = a + ib \) where \( a, b \in \mathbb{R} \) and \( i = \sqrt{-1} \). The set of all complex numbers is denoted \( \mathbb{C} \). For a complex number \( z = a + ib \), the complex conjugate, denoted \( \overline{z} \), is given by

\[
\overline{z} = a - ib.
\]

For two complex numbers \( z_1, z_2 \) we have \( \overline{z_1 z_2} = \overline{z_1} \overline{z_2} \). The modulus of \( z \), denoted \( |z| \), is given by

\[
|z| = \sqrt{a^2 + b^2} = \sqrt{z \overline{z}}.
\]

The complex exponential of \( z \in \mathbb{C} \) is defined by the Taylor series expansion

\[
e^z = \sum_{k=0}^{\infty} \frac{z^k}{k!}.
\]

The Taylor series is absolutely convergent in the whole complex plane. A very important identity involving the complex exponential is Euler’s identity

\[
e^{it} = \cos t + i \sin t
\]

for all real numbers \( t \in \mathbb{R} \). In particular, the complex exponential is a \( 2\pi \)-periodic function, so \( e^{i(t+2\pi k)} = e^{it} \) for any integer \( k \). We also note that

\[
e^{-it} = \cos t - i \sin t = \cos(-t) + i \sin(-t) = e^{-it}.
\]

Exercise 6.1. Prove Euler’s identity in three different ways.

1. Substitute Taylor expansions for \( e^{it} \), \( \cos t \), and \( \sin t \) in (6.1), and show that both sides are equal.

2. Write the complex number \( e^{it} \) in polar coordinates

\[
e^{it} = r(\cos \theta + i \sin \theta),
\]

where \( r = r(t) \) and \( \theta = \theta(t) \) are real numbers. Differentiate both sides in \( t \), and equate real and imaginary parts to show that \( r'(t) = 0 \) and \( \theta'(t) = 1 \). Use the initial values \( r(0) = 1 \) and \( \theta(0) = 0 \) to show that \( r(t) = 1 \) and \( \theta(t) = t \).
3. Write \( f(t) = \cos t + i \sin t \) for \( t \in \mathbb{R} \). Show that \( f'(t) = if(t) \). Use this to show that

\[
\frac{d}{dt}(f(t)e^{-it}) = 0.
\]

Therefore \( f(t) = ze^{it} \) for a complex number \( z \). Use that \( f(0) = 1 \) to show that \( z = 1 \).

\[\triangle\]

Exercise 6.2. Use Euler’s formula (6.1) to prove the following trigonometric identities.

(i) \( \cos(s + t) = \cos(s) \cos(t) - \sin(s) \sin(t) \)

(ii) \( \sin(s + t) = \sin(s) \cos(t) + \cos(s) \sin(t) \)

[Hint: Apply Euler’s formula to both sides of \( e^{i(s+t)} = e^{is}e^{it} \).]

\[\triangle\]

6.2 The Forward and Inverse Transforms

Python Notebook: .ipynb

Let \( \mathbb{Z}_n = \{0, 1, \ldots, n - 1\} \). It will be useful to think of complex vectors of length \( n \) as functions \( f : \mathbb{Z}_n \to \mathbb{C} \). Since the trigonometric functions \( \cos \) and \( \sin \) are periodic, it will be convenient to work with periodic signals, so we are really thinking of \( \mathbb{Z}_n \) as the cyclic group of integers modulo \( n \) (written \( \mathbb{Z}_n = \mathbb{Z}/n \)). When integers \( p, q \in \mathbb{Z}_n \) are added, subtracted, or multiplied, the result is interpreted modulo \( n \). For example, in \( \mathbb{Z}_4 \), \( 2 + 2 = 4 = 0 \) mod 4.

This has the effect of extending a signal \( f : \mathbb{Z}_n \to \mathbb{C} \) to a periodic function \( f : \mathbb{Z} \to \mathbb{C} \) of period \( n \), so that \( f(k + mn) = f(k) \) for all integers \( k \) and \( m \).

Let \( L^2(\mathbb{Z}_n) \) denote the vector space of functions \( f : \mathbb{Z}_n \to \mathbb{C} \). We define the inner product on \( L^2(\mathbb{Z}_n) \) by

\[
\langle f, g \rangle = \sum_{k=0}^{n-1} f(k)\overline{g(k)}.
\]

We note that \( L^2(\mathbb{Z}_n) \) is nothing other than \( \mathbb{C}^n \)—it is simply more convenient to take the viewpoint of functions instead of vectors. It is important to note that the inner product is not symmetric, and in fact

\[
\langle f, g \rangle = \overline{\langle g, f \rangle}.
\]
The norm of \( f \in L^2(\mathbb{Z}_n) \) is defined by \( \|f\| = \sqrt{\langle f, f \rangle} \).

The Discrete Fourier Transform (DFT) is an orthogonal change of basis in \( L^2(\mathbb{Z}_n) \) that expresses a function \( f : \mathbb{Z}_n \rightarrow \mathbb{C} \) in terms sinusoidal basis functions of different frequencies. Due to Euler’s formula (6.1), it is far more convenient to work with complex exponential functions of the form

\[
k \mapsto e^{2\pi i \sigma k} = \cos(2\pi \sigma k) + i \sin(2\pi \sigma k).
\]

The real and imaginary parts of (6.2) are exactly the sinusoidal basis functions we are interested in, and \( \sigma \geq 0 \) is the corresponding frequency (i.e., the number of cycles per unit). It is possible to avoid complex analysis and work with the sinusoidal functions directly (indeed, there are transforms called the cosine and sine transforms). However, encoding the sinusoids into the complex exponential via Euler’s formula greatly simplifies the analysis and leads to an elegant mathematical theory.

Now, we do not need all of the frequencies \( \sigma \geq 0 \) for a change of basis. In fact, since \( L^2(\mathbb{Z}_n) \) is \( n \)-dimensional (it is just \( \mathbb{C}^n \)), we should just need \( n \) basis vectors. To see which frequencies to choose, note that any \( f \in L^2(\mathbb{Z}_n) \) can be extended to a periodic function on \( \mathbb{Z} \) with period \( n \). Thus, we should only need to choose frequencies \( \sigma \) resulting in \( n \)-periodic complex exponentials.

This means the smallest period \( T = 1/\sigma \) of the sinusoid should divide evenly into \( n \), in other words, \( \sigma = \ell/n \) for some integer \( \ell = 0, 1, \ldots, n-1 \). Note that we do not take \( \ell = n \) since then \( \sigma = 1 \) and \( e^{2\pi i \sigma k} = 1 = e^0 \) for all \( k \), which is the same as taking \( \ell = 0 \). In fact, due to the periodicity of the complex exponential, taking any other integer values for \( \ell \) is redundant.

Thus, it is natural to consider the \( n \) complex exponential functions

\[
u_\ell(k) := e^{2\pi i k \ell/n}, \quad \ell = 0, 1, \ldots, n-1.
\]

It is often useful to note that we can set \( \omega = e^{2\pi i/n} \) and write \( u_\ell(k) = \omega^{\ell k} \).

The complex number \( \omega \) is an \( n \)th root of unity, meaning that \( \omega^n = e^{2\pi i} = 1 \). We also have \( \overline{\omega} = e^{-2\pi i/n} = \omega^{-1} \).

It is important to point out that the frequency of the sampled function \( u_\ell \) does not always match the frequency of the continuous function it is sampled from, due to an effect known as aliasing. Indeed, since \( \omega \) is an \( n \)th root of unity we have

\[
u_{n-\ell}(k) = \omega^{(n-\ell)k} = \omega^{nk} \omega^{-\ell k} = \overline{u_\ell(k)}.
\]

So at the discrete level, the functions \( u_{n-\ell} \) and \( u_\ell \) have the same frequency, and are in fact just complex conjugates of each other. Hence, the exponentials \( u_\ell \)
with the highest frequencies correspond to $\ell \approx n/2$. The frequencies increase with $\ell$ for $0 \leq \ell \leq n/2$, and decrease with $\ell$ for $n/2 < \ell \leq n - 1$. We give an illustration of this for $n = 8$ in Figure 6.2.

It turns out that the complex exponential functions $u_\ell$ are mutually orthogonal.
Lemma 6.3. The functions $u_0, u_1, \ldots, u_{n-1}$ are orthogonal. In particular

$$\langle u_\ell, u_m \rangle = \begin{cases} n, & \text{if } \ell = m \\ 0, & \text{otherwise.} \end{cases}$$  

(6.5)

Proof. We compute

$$\langle u_\ell, u_m \rangle = \sum_{k=0}^{n-1} u_\ell(k) \overline{u_m(k)}$$

$$= \sum_{k=0}^{n-1} \omega^{\ell k} \overline{\omega^{k m}}$$

$$= \sum_{k=0}^{n-1} \omega^{\ell k} \omega^{-km} = \sum_{k=0}^{n-1} (\omega^{\ell-m})^k.$$  

If $\ell = m$ then we have $\omega^{\ell-m} = \omega^0 = 1$ and $\langle u_\ell, u_m \rangle = n$. If $\ell \neq m$ then we note that the expression above is a (truncated) geometric series in the variable $r := \omega^{\ell-k}$, and we thus have

$$\langle u_\ell, u_m \rangle = \frac{r^n - 1}{r - 1}.$$  

The proof is completed by noting that $r^n = (\omega^n)^{\ell-k} = 1$, since $\omega$ is an $n^{\text{th}}$ root of unity.

Since $u_0, u_1, \ldots, u_{n-1}$ are orthogonal, they are linearly independent and span $L^2(\mathbb{Z}_n)$. The DFT is simply the change of basis that writes a function $f \in L^2(\mathbb{Z}_n)$ in this new basis, that is, in the form

$$f = \frac{1}{n} \sum_{\ell=0}^{n-1} c_\ell u_\ell.$$  

Note that the $\frac{1}{n}$ factor is simply a convention, and could have been omitted, in which case it would appear in the formula for $c_\ell$ below. To determine the coefficients $c_\ell$, we take the inner product of both sides above with $u_m$ and use the orthogonality shown in Lemma 6.3 to obtain

$$\langle f, u_m \rangle = \frac{1}{n} \sum_{\ell=0}^{n-1} c_\ell \langle u_\ell, u_m \rangle = c_m.$$
Therefore we have \( c_m = \langle f, u_m \rangle \), and we can write \( f \) in the form

\[
f = \frac{1}{n} \sum_{\ell=0}^{n-1} \langle f, u_\ell \rangle u_\ell.
\]

The coefficients \( c_\ell \) can be written out as

\[
c_\ell = \langle f, u_\ell \rangle = \sum_{k=0}^{n-1} f(k) \overline{u_\ell(k)} = \sum_{k=0}^{n-1} f(k) \omega^{-k\ell} = \sum_{k=0}^{n-1} f(k) e^{-2\pi ik\ell/n}.
\]

This leads us to the definition of the DFT, which simply computes the coefficients \( c_\ell \).

**Definition 6.4.** The *Discrete Fourier Transform (DFT)* is the mapping \( \mathcal{D} : L^2(\mathbb{Z}_n) \rightarrow L^2(\mathbb{Z}_n) \) defined by

\[
\mathcal{D}f(\ell) = \sum_{k=0}^{n-1} f(k) \omega^{-k\ell} = \sum_{k=0}^{n-1} f(k) e^{-2\pi ik\ell/n},
\]

where \( \omega = e^{2\pi i/n} \).

It is important to point out that the DFT has symmetries when applied to real-valued signals.

**Proposition 6.5.** If \( f \in L^2(\mathbb{Z}_n) \) is real-valued (i.e., \( f(k) \in \mathbb{R} \) for all \( k \)), then

\[
\mathcal{D}f(\ell) = \overline{\mathcal{D}f(n-\ell)}.
\]

**Proof.** The proof follows from (6.4). Indeed, since \( f \) is real-valued, we use (6.4) to compute

\[
\mathcal{D}f(\ell) = \langle f, u_\ell \rangle = \langle f, \overline{u_{n-\ell}} \rangle = \overline{\langle f, u_{n-\ell} \rangle} = \overline{\mathcal{D}f(n-\ell)}.
\]

Proposition 6.5 shows that we only need to record the first half of the DFT coefficients, \( \mathcal{D}f(\ell) \) for \( 0 \leq \ell \leq \frac{n}{2} \), when transforming a real-valued signal.

We immediately have the following inversion theorem.

**Theorem 6.6 (Fourier Inversion Theorem).** For any \( f \in L^2(\mathbb{Z}_n) \) we have

\[
f(k) = \frac{1}{n} \sum_{\ell=0}^{n-1} \mathcal{D}f(\ell) \omega^{k\ell} = \frac{1}{n} \sum_{\ell=0}^{n-1} \mathcal{D}f(\ell) e^{2\pi ik\ell/n}.
\]
Proof. Let \( g(k) \) denote the right hand side of (6.6). We need to show that \( g(k) = f(k) \). We compute

\[
g(k) = \frac{1}{n} \sum_{\ell=0}^{n-1} Df(\ell) \omega^{k\ell}
\]

\[
= \frac{1}{n} \sum_{\ell=0}^{n-1} \sum_{j=0}^{n-1} f(j) \omega^{-j\ell} \omega^{k\ell}
\]

\[
= \frac{1}{n} \sum_{j=0}^{n-1} \sum_{\ell=0}^{n-1} f(j) \sum_{j=0}^{n-1} \langle u_j, u_{k\ell} \rangle
\]

\[
= \frac{1}{n} \sum_{j=0}^{n-1} f(j) \langle u_k, u_j \rangle = f(k),
\]

due to Lemma 6.3.

Based on Theorem 6.6, we make the following definition.

**Definition 6.7** (Inverse Discrete Fourier Transform). The *Inverse Discrete Fourier Transform (IDFT)* is the mapping \( D^{-1} : L^2(\mathbb{Z}_n) \rightarrow L^2(\mathbb{Z}_n) \) defined by

\[
D^{-1} f(\ell) = \frac{1}{n} \sum_{k=0}^{n-1} f(k) \omega^{k\ell} = \frac{1}{n} \sum_{k=0}^{n-1} f(k) e^{2\pi i k \ell/n}.
\]

**Remark 6.8.** By Theorem 6.6, we have \( D^{-1}Df = f \) for all \( f \in L^2(\mathbb{Z}_n) \). The other direction, \( DD^{-1}f = f \), follows from standard linear algebra facts, but it is also very easy to show using the argument in Theorem 6.6. Indeed, we have

\[
D(D^{-1}f)(k) = \sum_{\ell=0}^{n-1} D^{-1}f(\ell) \omega^{k\ell} = \frac{1}{n} \sum_{\ell=0}^{n-1} \sum_{j=0}^{n-1} f(j) \omega^{j\ell} \omega^{-k\ell} = f(k),
\]

where the final equality is proved in a similar way as in Theorem 6.6.

**Remark 6.9.** Define the \( n \times n \) complex-valued matrix with entries \( W(k, \ell) = \omega^{k\ell} \), that is

\[
W = \begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\]

(6.7)
Then the DFT can be expressed via matrix multiplication as $Df = \overline{W}f$. The inverse DFT can be expressed as $D^{-1}f = \frac{1}{n}Wf$. In both cases we treat $f$ as a vector $f \in \mathbb{C}^n$. Theorem 6.6 (Fourier Inversion) can be restated as saying that $W\overline{W} = nI$.

**Exercise 6.10.** Show that the DFT enjoys the following basic shift properties.

1. Recall that $u_\ell(k) := e^{2\pi ik\ell/n} = \omega^{k\ell}$. Show that

   $$D(f \cdot u_\ell)(k) = Df(k - \ell).$$

2. Let $T_\ell : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ be the translation operator $T_\ell f(k) = f(k + \ell)$. Show that

   $$D(T_\ell f)(k) = e^{-2\pi ik\ell/n}Df(k).$$

   [Hint: You can equivalently show that $D^{-1}(f \cdot u_\ell)(k) = D^{-1}f(k - \ell)$, using an argument similar to part 1.]

△

### 6.3 The Fast Fourier Transform (FFT)

**Python Notebook:** .ipynb

For $f \in L^2(\mathbb{Z}_n)$, computing $Df$ or $D^{-1}f$ requires on the order of $n^2$ operations (multiplications and additions) if we use perform the computation exactly as in Definition 6.4. Indeed, the sum defining $Df(\ell)$ has $n$ terms, and we have to compute $n$ coefficients, for $\ell = 0, 1, \ldots, n - 1$. From the viewpoint of the DFT as matrix multiplication discussed in Remark 6.9, we are multiplying an $n \times n$ matrix by a vector, which requires $O(n^2)$ operations. If we always computed the DFT in this way, it would be too slow for most applications. Indeed, taking $n = 1000$ samples in an audio file with a sample rate of 44.1 kHz is only 22 ms of audio, but would take millions ($n^2 = 10^6 = 1$ million) of operations to compute the DFT of very short audio sample. One whole second of audio has $n = 44,100$ samples, and it would take on the order of $n^2 \approx 2 \times 10^9$ operations to compute the DFT. This quickly becomes computationally intractable.

It turns out that many of the $O(n^2)$ operations performed to compute the DFT as per Definition 6.4 are redundant, and if the computation is done carefully it can be reduced to $O(n \log n)$ operations. The resulting algorithm is called the Fast Fourier Transform (FFT). The computational efficiency of the FFT is perhaps the dominant factor in the widespread applicability of the
Fourier Transform in problems like image and audio compression, and solving partial differential equations, among many others. The FFT has been ranked as one of the top 10 algorithms of 20th Century by IEEE Computing in Science & Engineering magazine, and according to Gilbert Strang (MIT), is “the most important numerical algorithm of our lifetime.”

The FFT is based on subsampling the signal into its even and odd samples, applying the DFT to these two components separately. The algorithm then proceeds recursively by applying the even/odd subsampling to each component from the first step. The whole idea is based on the Lemma 6.11 below, which allows us to efficiently compute the DFT of a signal from the DFT of its even and odd samples.

Before presenting the lemma, we introduce some additional notation. For \( f \in L^2(\mathbb{Z}_n) \) with \( n \) even, the even and odd parts of \( f \), denoted \( f_e \) and \( f_o \), respectively, are the functions in \( L^2(\mathbb{Z}_n^2) \) defined by

\[
f_e(k) = f(2k) \quad \text{and} \quad f_o(k) = f(2k + 1),
\]

for \( k = 0, 1, \ldots, \frac{n}{2} - 1 \). In this section we will work with the DFT for signals of different lengths so we will denote by \( \mathcal{D}_n \) and \( \mathcal{D}^{-1}_n \) the forward and inverse DFT, respectively, on \( L^2(\mathbb{Z}_n) \). It is important to note that \( \mathcal{D}_1 \) is the identity, since \( \mathcal{D}_1 f(0) = f(0) \).

**Lemma 6.11.** For each \( f \in L^2(\mathbb{Z}_n) \) with \( n \) even we have

\[
\mathcal{D}_n f(\ell) = \mathcal{D}_\frac{n}{2} f_e(\ell) + e^{-2\pi i \ell/n} \mathcal{D}_\frac{n}{2} f_o(\ell),
\]

for \( \ell = 0, 1, \ldots, n - 1 \).

**Proof.** The proof is a straightforward computation based on splitting the sum defining \( \mathcal{D} f(\ell) \) into even and odd parts. We have

\[
\mathcal{D}_n f(\ell) = \sum_{k=0}^{n-1} f(k)e^{-2\pi i k\ell/n} = \sum_{k=0}^{\frac{n}{2}-1} f(2k)e^{-2\pi i 2k\ell/n} + \sum_{k=0}^{\frac{n}{2}-1} f(2k+1)e^{-2\pi i (2k+1)\ell/n}
\]

\[
= \sum_{k=0}^{\frac{n}{2}-1} f_e(k)e^{-2\pi i 2k\ell/n} + e^{-2\pi i \ell/n} \sum_{k=0}^{\frac{n}{2}-1} f_o(k)e^{-2\pi i 2k\ell/n}
\]

\[
= \sum_{k=0}^{\frac{n}{2}-1} f_e(k)e^{-2\pi i k\ell/(n/2)} + e^{-2\pi i \ell/n} \sum_{k=0}^{\frac{n}{2}-1} f_o(k)e^{-2\pi i k\ell/(n/2)}
\]

\[
= \mathcal{D}_\frac{n}{2} f_e(\ell) + e^{-2\pi i \ell/n} \mathcal{D}_\frac{n}{2} f_o(\ell),
\]

72
which completes the proof.

**Remark 6.12.** In (6.8), it is important to point out that $\mathcal{D}_n f \in L^2(\mathbb{Z}_n)$ and $\mathcal{D}_{\frac{n}{2}} f_e, \mathcal{D}_{\frac{n}{2}} f_o \in L^2(\mathbb{Z}_{\frac{n}{2}})$. For $\frac{n}{2} \leq \ell \leq n - 1$, the periodicity of $\mathbb{Z}_{\frac{n}{2}}$ gives that

$$\mathcal{D}_n f(\ell) = \mathcal{D}_{\frac{n}{2}} f_e(\ell - \frac{n}{2}) + e^{-2\pi i \ell/n} \mathcal{D}_{\frac{n}{2}} f_o(\ell - \frac{n}{2}).$$

This is important to keep in mind in practical implementations, since indexing arrays in Python or Matlab does not work cyclically.

At a high level, the FFT works by using Lemma 6.11 to recursively split the problem up into smaller subproblems. Each split reduces the length of the signals by half, and after $\log_2 n$ such splits, the problem is reduced to computing the DFT of very short signals, say, of length $n = 1$, which can be computed in constant time. Each time we recombine the results from subproblems using (6.8) we incur a cost of $O(n)$ operations, yielding an overall complexity of $O(n \log_2 n)$.

The description above is just a summary of the main ideas. We now consider the FFT algorithm in detail and prove that the computational complexity is $O(n \log_2 n)$. From now on we restrict our analysis to signal lengths $n$ that are powers of 2, so $n = 2^k$ for some positive integer $k$. This ensures that every time we split we end up with even-length signals that can be further split evenly, and makes the FFT algorithm simpler to state.

In Algorithm 6.3 we show Python code for the FFT algorithm using a recursive implementation. The algorithm checks if the input signal has length $n = 1$, and if so, it returns $f$, since $\mathcal{D}_1$ is the identity (as noted above). If $n \geq 2$ then the algorithm splits the signal into its even and odd parts $f_e = f[::2]$ and $f_o = f[1::2]$, takes their DFT by calling fft recursively, and then recombines the resulting DFTs $\mathcal{D}f_e$ and $\mathcal{D}f_o$ using (6.8) to compute $\mathcal{D}f$. We also point out that Steps 11 and 12 simply extend $\mathcal{D}f_e$ and $\mathcal{D}f_o$ to functions on $\mathbb{Z}_n$ by periodicity, so that (6.8) can be applied (see Remark 6.12). We also recall (see Step 13) that $1j = \sqrt{-1}$ in Python.

We now carefully analyze the computational complexity of the FFT. In order to do so, it is important to consider how one chooses to count operations. Here, we will count an operation of addition, subtraction, multiplication or division of real numbers as a single operation. Applying functions like $\cos$ or $\sin$ to real numbers will also count as one operation. Operations on complex numbers take multiple real operations. For example, adding two complex numbers

$$(a + ib) + (c + id) = (a + b) + i(c + d)$$
Algorithm 6.1 The Fast Fourier Transform (FFT) in Python

1  import numpy as np

2  

3  def fft(f):
4      n = len(f)
5      k = np.arange(n)
6      if n == 1:
7          return f
8      else:
9          Dfe = fft(f[::2])
10         Dfo = fft(f[1::2])
11         Dfe = np.hstack((Dfe,Dfe))
12         Dfo = np.hstack((Dfo,Dfo))
13         return Dfe + np.exp(-2*np.pi*1j*k/n)*Dfo

requires two real additions (adding \(a + b\) and \(c + d\)). The complex number \(a + ib\) is stored in memory as a pair \((a, b)\), with the understanding that \(b\) is the imaginary part of the complex number, and \(a\) is the real part. So the addition operation between \((a + b)\) and \(i(c + d)\) is not a real operation and does not need to be computed, since the result is simply stored as \((a + b, c + d)\). The same can be said for the multiplication of \(i\) and \(c + d\); these numbers are not multiplied by the computer (how could they be?), so this does not count as an operation. The result of this discussion is that addition or subtraction of complex numbers takes two real operations.

Multiplication of complex numbers takes more than two real operations, since we must write

\[(a + ib)(c + id) = (ac - bd) + i(ad + bc).\]

So multiplying two complex numbers requires computing \(ac - bd\) and \(ad + bc\), which takes 6 real operations (4 multiplications and 2 additions). Note it is also possible to multiply numbers in polar coordinates with fewer operations, since

\[r_1e^{i\theta_1}r_2e^{-i\theta_2} = r_1r_2e^{i(\theta_1+\theta_2)}\]

requires only 2 real operations (one multiplication and one addition). However, adding or subtracting in polar coordinates is more expensive, since one must convert back to Cartesian coordinates \(a + ib\) first via Euler’s Formula. Thus, we will conduct our analysis assuming the complex numbers are stored in
Cartesian format, requiring 2 operations per addition or subtraction and 6 operations for a multiplication.

We define

\[ A_n = \text{Number of real operations taken by the FFT on } L^2(\mathbb{Z}_n). \]

We note that since \( D_1 \) is the identity, which requires no operations to compute, we have \( A_1 = 0 \). We can use Lemma 6.11 to obtain a recursion for \( A_n \) in terms of \( A_{n/2} \). Indeed, using (6.8) to compute \( Df(\ell) \) for \( \ell = 0, 1, \ldots, n - 1 \) requires computing \( D_{n/2} f_e \) and \( D_{n/2} f_o \), which takes \( 2A_{n/2} \) operations. Then we need to perform the complex addition and multiplication in (6.8), which takes \( 2 + 6 = 8 \) operations. This has to be done \( n \) times (for \( \ell = 0, 1, \ldots, n - 1 \)), yielding \( 8n \) operations. Thus, we find that

\[
(6.9) \quad A_n = 2A_{n/2} + 8n.
\]

Notice we did not allocate any computation time to the complex exponential \( e^{-2\pi i \ell/n} \). This quantity is independent of the signal \( f \) and can be computed once, offline, and the values can be stored in memory. Thus, we do not include this in the operation count of the algorithm.

The recursion (6.9) allows us to solve explicitly for \( A_n \), as shown in the following lemma.

**Lemma 6.13.** Let \( n = 2^k \) for a positive integer \( k \), and assume \( A_n \) satisfies (6.9) for \( n \geq 2 \) and \( A_1 = 0 \). Then we have

\[
(6.10) \quad A_n = 8n \log_2 n.
\]

**Proof.** We can iterate (6.9) again to obtain

\[
A_n = 2 \left( 2A_{n/2} + 8 \frac{n}{2} \right) + 8n = 2^2 A_{n/2} + 8n + 8n,
\]

and

\[
A_n = 2^2 \left( 2A_{n/2} + 8 \frac{n}{2^2} \right) + 8n + 8n = 2^3 A_{n/2^2} + 8n + 8n + 8n.
\]

Iterating this \( k \) times, by induction, we have that

\[
A_n = 2^k A_{n/2^k} + 8nk.
\]

Since \( n = 2^k \) we have \( A_{n/2^k} = A_1 = 0 \) and \( k = \log_2 n \), which yields

\[
A_n = 8nk = 8n \log_2 n.
\]
Lemma 6.13 shows that the FFT takes at most $8n \log_2 n$ operations to execute, which is an order of magnitude improvement over the naive implementation based on Definition 6.4 that takes $O(n^2)$ operations. It turns out this complexity can be improved to $5n \log_2 n$ by removing some redundant computations.

**Remark 6.14.** As in Remark 6.12, we consider writing (6.8) in different ways depending on whether $\ell \leq \frac{n}{2} - 1$ or not. Since 

$$e^{-2\pi i(\ell + \frac{n}{2})/n} = e^{-2\pi i\ell/n}e^{-\pi i} = -e^{-2\pi i\ell/n}$$

and $D_2 f_e(\ell), D_2 f_o$ are $\frac{n}{2}$ periodic, we can write (6.8) as

$$D_n f(\ell) = D_2 f_e(\ell) + e^{-2\pi i\ell/n}D_2 f_o(\ell), \text{ and}$$

$$D_n f(\ell + \frac{n}{2}) = D_2 f_e(\ell) - e^{-2\pi i\ell/n}D_2 f_o(\ell)$$

for $\ell = 0, 1, \ldots, \frac{n}{2} - 1$. Notice that there are now only $\frac{n}{2}$ complex multiplications to perform, since they are common between the two parts. We still need $n$ additions, so the number of real operations to combine $D_2 f_e(\ell)$ and $D_2 f_o$ into $D f$ is reduced from $8n$, as above, to $6\frac{n}{2} + 2n = 5n$. The recursion (6.9) becomes

$$A_n = 2A_{\frac{n}{2}} + 5n$$

and the complexity is thus $A_n = 5n \log_2 n$.

The computational complexity can be further improved to $4n \log_2 n$ by considering a 3-way split, where the odd terms are further split in half before the recursion. Exercise 6.15 explores this algorithm, called the split-radix FFT, which enjoyed its role as the fastest FFT for power-of-two $n$ for quite some time, until the relatively recent work [8] made some modifications to the algorithm and improved the complexity to $\frac{34}{9} n \log_2 n$. The split radix methods are variants of the general Cooley-Tukey FFT algorithm [4]. The Cooley-Tukey FFT can be applied to any $n$, not just powers of 2, although the constants in the computational complexity are worse for non powers of two.

**Exercise 6.15** (Split-radix FFT). Assume $n$ is a power of 2 and let $f \in L^2(\mathbb{Z}_n)$. Define $f_e \in L^2(\mathbb{Z}_{\frac{n}{2}})$, and $f_{o,1}, f_{o,2} \in L^2(\mathbb{Z}_{\frac{n}{4}})$ by 

$$f_e(k) = f(2k), \quad f_{o,1}(k) = f(4k + 1), \quad \text{and} \quad f_{o,2}(k) = f(4k + 3).$$

(i) Show that 

$$(6.11) \quad D_n f(\ell) = D_2 f_e(\ell) + e^{-2\pi i\ell/n}D_2 f_{o,1}(\ell) + e^{-2\pi i3\ell/n}D_2 f_{o,2}(\ell).$$
(ii) The FFT algorithm based on the 3-way split in (6.11) is called the split-radix FFT algorithm. As in Remark 6.14, there are a lot of redundant computations in (6.11), and these must be accounted for in order to realize the improved complexity of the split-radix FFT. Show that

\[
\mathcal{D}_n f(\ell) = \mathcal{D}_{n/2} f_e(\ell) + e^{-2\pi i \ell/n} \mathcal{D}_{n/4} f_{o,1}(\ell) + e^{-2\pi i 3\ell/n} \mathcal{D}_{n/4} f_{o,2}(\ell),
\]

\[
\mathcal{D}_n f(\ell + \frac{n}{2}) = \mathcal{D}_{n/2} f_e(\ell) - e^{-2\pi i \ell/n} \mathcal{D}_{n/4} f_{o,1}(\ell) + e^{-2\pi i 3\ell/n} \mathcal{D}_{n/4} f_{o,2}(\ell),
\]

\[
\mathcal{D}_n f(\ell + \frac{n}{4}) = \mathcal{D}_{n/4} f_e(\ell + \frac{n}{4}) - i(e^{-2\pi i \ell/n} \mathcal{D}_{n/4} f_{o,1}(\ell) - e^{-2\pi i 3\ell/n} \mathcal{D}_{n/4} f_{o,2}(\ell)),
\]

\[
\mathcal{D}_n f(\ell + \frac{3n}{4}) = \mathcal{D}_{n/4} f_e(\ell + \frac{n}{4}) + i(e^{-2\pi i \ell/n} \mathcal{D}_{n/4} f_{o,1}(\ell) - e^{-2\pi i 3\ell/n} \mathcal{D}_{n/4} f_{o,2}(\ell)),
\]

for \(0 \leq \ell \leq \frac{n}{4} - 1\). This gives all the outputs of \(\mathcal{D} f(\ell)\) and reduces the number of multiplications and additions required.

(iii) Explain how the observations in Part (ii) allow you to compute \(D_n f\) from \(D_{n/2} f_e, D_{n/4} f_{o,1}\) and \(D_{n/4} f_{o,2}\) using 6\(n\) real operations. [Note, multiplications with \(\pm 1\) or \(\pm i\) do not count, since they amount to negation of real or imaginary parts, which can be absorbed into the next operation by changing it from addition to subtraction or vice versa]

(iv) Show that part (iii) implies that the number of real operations taken by the split-radix FFT, denoted again as \(A_n\), satisfies the recursion

\[A_n = A_{n/2} + 2A_{n/4} + 6n.\]

Explain why \(A_1 = 0\) and \(A_2 = 4\). Use this to show that \(A_n \leq 4n \log_2 n\). [Hint: Define \(B_n = A_n - 4n \log_2 n\) and show that \(B_n\) satisfies

\[B_n = B_{n/2} + 2B_{n/4}\]

with \(B_1 = 0\) and \(B_2 = -4\). Use this to argue that \(B_n \leq 0\) for all power-of-two \(n\).] [Note: If one is more careful about redundant computations (there are additional multiplications with \(\pm 1\) or \(\pm i\) that can be skipped), then the complexity of the split-radix FFT algorithm is actually \(4n \log_2 n - 6n + 8\) real operations].

\[\triangle\]

Remark 6.16. Before concluding this section, we remark that the FFT algorithm can be easily extended to compute the inverse DFT. The analogous identity to (6.8) for \(\mathcal{D}_n^{-1}\) is

\[
(6.12) \quad \mathcal{D}_n^{-1} f(\ell) = \frac{1}{2} \mathcal{D}_{n/2}^{-1} f_e(\ell) + \frac{1}{2} e^{2\pi i \ell/n} \mathcal{D}_{n/2}^{-1} f_o(\ell).
\]
The proof of (6.12) is left to an exercise (below). The inverse FFT is formulated very similarly to the FFT, just using (6.12) in place of (6.8). As an alternative, one can always use the identity
\[ D_n^{-1} = \frac{1}{n} \overline{D_n f} \]
to compute the inverse DFT efficiently using a single forward FFT, two complex conjugation operations, and an elementwise division.

**Exercise 6.17.** Prove that (6.12) holds under the same assumptions as in Lemma 6.11. \[ \triangle \]

### 6.4 Parseval’s Identities

Recall that for a real-valued orthogonal matrix \( Q \), the inverse of \( Q \) is the transpose \( Q^T \). Since the DFT is an orthogonal change of coordinates, a similar property holds. Indeed, the following lemma shows that the IDFT \( D^{-1} \) is the adjoint of the DFT \( D \), up to the factor \( 1/n \).

**Lemma 6.18.** For each \( f, g \in L^2(\mathbb{Z}_n) \) we have
\[ \frac{1}{n} \langle Df, g \rangle = \langle f, D^{-1}g \rangle. \]

**Proof.** We compute
\[
\langle f, D^{-1}g \rangle = \sum_{k=0}^{n-1} f(k) \overline{D^{-1}g(k)} \\
= \frac{1}{n} \sum_{k=0}^{n-1} f(k) \sum_{\ell=0}^{n-1} g(\ell) \omega^{kl} \\
= \frac{1}{n} \sum_{\ell=0}^{n-1} g(\ell) \sum_{k=0}^{n-1} f(k) \omega^{-kl} \\
= \frac{1}{n} \sum_{\ell=0}^{n-1} g(\ell) Df(\ell) = \frac{1}{n} \langle Df, g \rangle,
\]
which completes the proof. \( \Box \)

An immediate consequence of Lemma 6.18 are Parseval’s Identities, which state that the DFT preserves the inner product on \( L^2(\mathbb{Z}_n) \), and hence also preserves the norm, up to the \( \frac{1}{n} \) factor.
Theorem 6.19 (Parseval’s Identities). Let \( f, g \in L^2(\mathbb{Z}_n) \). Then it holds that

(i) \( \langle f, g \rangle = \frac{1}{n} \langle Df, Dg \rangle \), and

(ii) \( \|f\|^2 = \frac{1}{n} \|Df\|^2 \).

Proof. To prove (i), we use Lemma 6.18 and Theorem 6.6 to write

\[
\langle f, g \rangle = \langle f, D^{-1}Dg \rangle = \frac{1}{n} \langle Df, Dg \rangle.
\]

To prove (ii), we take \( f = g \) in part (i).

Remark 6.20. Of course, a similar statement holds for the inverse transform \( D^{-1} \). Indeed, Lemma 6.18 and Theorem 6.6 imply

\[
\frac{1}{n} \langle f, g \rangle = \frac{1}{n} \langle DD^{-1}f, g \rangle = \langle D^{-1}f, D^{-1}g \rangle.
\]

Setting \( f = g \) yields \( \frac{1}{n} \|f\|^2 = \|D^{-1}f\|^2 \).

6.5 Convolution and the DFT

One of the most important properties of the DFT is that it turns convolutions into products. This property makes the DFT useful for solving partial differential equations, and for efficiently computing certain convolutions.

We first define the discrete convolution on the cyclic group \( \mathbb{Z}_n \).

Definition 6.21. The **discrete cyclic convolution** of \( f, g \in L^2(\mathbb{Z}_n) \), denoted \( f \ast g \), is the function in \( L^2(\mathbb{Z}_n) \) defined for each \( k \) by

\[
(f \ast g)(k) = \sum_{j=0}^{n-1} f(j)g(k-j).
\]

(6.13)

We note that the definition of the convolution makes use of the fact that \( \mathbb{Z}_n \) is a cyclic group when \( k-j \) falls outside of \( 0, 1, \ldots, n-1 \) (i.e., the values wrap around). We leave some basic properties of the convolution to an exercise.

Exercise 6.22. Let \( f, g, h \in L^2(\mathbb{Z}_n) \). Show that the following hold.

(i) \( f \ast g = g \ast f \);

(ii) \( f \ast (g \ast h) = (f \ast g) \ast h \);

(iii) \( f \ast (g + h) = f \ast g + f \ast h \).
The main result of this section concerns the DFT of a convolution.

**Lemma 6.23** (Convolution and the DFT). For \( f, g \in L^2(\mathbb{Z}_n) \) we have

\[
\mathcal{D}(f \ast g) = \mathcal{D}f \cdot \mathcal{D}g. 
\]

**(6.14)**

*Proof.* We compute

\[
\mathcal{D}(f \ast g)(\ell) = \sum_{k=0}^{n-1} (f \ast g)(k) \omega^{-k\ell} 
= \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} f(j) g(k-j) \omega^{-k\ell} 
= \sum_{j=0}^{n-1} f(j) \omega^{-j\ell} \left( \sum_{k=0}^{n-1} g(k-j) \omega^{-(k-j)\ell} \right). 
\]

Since \( \mathbb{Z}_n \) is cyclic, for any function \( h \in L^2(\mathbb{Z}_n) \) and any \( j \in \mathbb{Z}_n \) we have

\[
\sum_{k=0}^{n-1} h(k) = \sum_{k=0}^{n-1} h(k-j). 
\]

Thus, the term in brackets above is exactly \( \mathcal{D}g(\ell) \) and we have

\[
\mathcal{D}(f \ast g)(\ell) = \sum_{j=0}^{n-1} f(j) \omega^{-j\ell} \mathcal{D}g(\ell) = \mathcal{D}f(\ell) \mathcal{D}g(\ell). 
\]

*Remark 6.24.* Lemma 6.23 is arguably one of the most important properties of the DFT. It allow for efficient computation of convolutions via the FFT. Indeed, we can use (6.14) to write

\[
f \ast g = \mathcal{D}^{-1}(\mathcal{D}f \cdot \mathcal{D}g). 
\]

Using the FFT, we can compute the convolution with two forward and one inverse transformation, which takes \( O(n \log n) \) operations. Computing the convolution by the definition (6.13) takes \( O(n^2) \) operations.

Also, as we explore in Exercise 6.26, all discrete derivatives of \( f \) can be viewed as convolutions of \( f \) with a particular choice of \( g \). Thus, Lemma 6.23 turns differential equations, involving discrete derivatives, into algebraic equations which are easy to solve. This enables the use of the FFT for efficiently solving linear constant coefficient partial differential equations. We explore this further in Section 6.6.
Remark 6.25. Of course, a similar property holds for the inverse DFT $\mathcal{D}^{-1}$. To see this, we use the identity $\mathcal{D}^{-1}f = \frac{1}{n} \overline{Df}$ to obtain

$$\mathcal{D}^{-1}(f * g) = \frac{1}{n} \overline{Df * g} = \frac{1}{n} \overline{Df} \cdot \overline{Dg} = n \mathcal{D}^{-1}f \cdot \mathcal{D}^{-1}g.$$ 

Exercise 6.26. Discrete derivatives (difference quotients) can be interpreted as convolutions. Complete the following exercises.

(i) For $f \in L^2(\mathbb{Z}_n)$ define the backward difference

$$\nabla^- f(k) = f(k) - f(k - 1).$$

Find $g \in L^2(\mathbb{Z}_n)$ so that $\nabla^- = g * f$ and use this with Lemma 6.23 to show that $\mathcal{D}(\nabla^- f)(k) = (1 - \omega^{-k}) \mathcal{D}f(k)$, where $\omega = e^{2\pi i/n}$.

(ii) For $f \in L^2(\mathbb{Z}_n)$ define the forward difference

$$\nabla^+ f(k) = f(k + 1) - f(k).$$

Find $g \in L^2(\mathbb{Z}_n)$ so that $\nabla^+ = g * f$ and use this with Lemma 6.23 to show that $\mathcal{D}(\nabla^+ f)(k) = (\omega^k - 1) \mathcal{D}f(k)$.

(iii) For $f \in L^2(\mathbb{Z}_n)$ define the centered difference by

$$\nabla f(k) = \frac{1}{2}(\nabla^- f(k) + \nabla^+ f(k)) = \frac{1}{2}(f(k + 1) - f(k - 1)).$$

Use parts (i) and (ii) to show that

$$\mathcal{D}(\nabla f)(k) = \frac{1}{2}(\omega^k - \omega^{-k}) \mathcal{D}f(k) = i \sin(2\pi k/n) \mathcal{D}f(k).$$

(iv) For $f \in L^2(\mathbb{Z}_n)$, define the discrete Laplacian as

$$\Delta f(k) = \nabla^+ \nabla^- f(k) = f(k + 1) - 2f(k) + f(k - 1).$$

Use parts (i) and (ii) to show that

$$\mathcal{D}(\Delta f)(k) = (\omega^k + \omega^{-k} - 2) \mathcal{D}f(k) = 2(\cos(2\pi k/n) - 1) \mathcal{D}f(k).$$

Exercise 6.27. Consider the Poisson equation on $\mathbb{Z}_n$

$$(6.15) \quad -\Delta f(k) = g(k) \quad \text{for} \ k \in \mathbb{Z}_n.$$ 

The source term $g \in L^2(\mathbb{Z}_n)$ is given, and $f \in L^2(\mathbb{Z}_n)$ is the unknown we wish to solve for. The Laplacian $\Delta$ is defined in Exercise 6.26. Use Exercise 6.26 (iii) to derive a solution formula for $f(k)$. Is there a condition you need to place on $\mathcal{D}g$ for your solution formula to make sense? [Hint: Take the DFT of both sides of (6.15), solve for $\mathcal{D}f$, and then apply the inverse DFT $\mathcal{D}^{-1}$. Be careful not to divide by zero when you solve for $\mathcal{D}f$.]
6.6 Application: Signal denoising

Noise arises in most signal acquisition processes. For example, in a microphone, air molecules constantly bombard the diaphragm, causing random vibrations not associated with the sound one wishes to record, and electronic circuitry generates thermal noise due to the natural random motions of electrons inside conductors. Sensors on digital cameras are also prone to noise in the image capture process, for similar reasons, and this can be especially pronounced in low light situations. Noise can also be introduced after signal acquisition. For instance, noise can arise as the result of compression artifacts or corruption in the transmission of signals wirelessly. In Figure 6.3 shows a synthetic signal and a noisy version corrupted by additive Gaussian noise (at each sample point we add an independent Gaussian random variable to the signal). Many types of noise are well approximated by Gaussian noise due to the Central Limit Theorem in probability.

Let $f \in L^2(\mathbb{Z}_n)$ represent a noisy signal, like in Figure 6.3(b). While we are interested primarily in real-valued signals, the analysis is similar for complex-valued signals $f \in L^2(\mathbb{Z}_n)$, so we proceed in generality. The goal of signal denoising is to recover the clean signal in Figure 6.3(a), which amounts to removing the noise. In the sections below, we discuss Tikhonov regularization and Total Variation regularization for denoising.
6.6.1 Tikhonov regularization

Python Notebook: .ipynb

A widely used and successful method for signal and image denoising is the regularized variational approach. While there are many versions of this, we focus here on a simple case known as Tikhonov regularization, which denoises the signal by minimizing the function $E : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ defined by

\begin{equation}
E(u) = \sum_{k=0}^{n-1} |u(k) - f(k)|^2 + \lambda \sum_{k=0}^{n-1} |u(k) - u(k-1)|^2,
\end{equation}

where $\lambda \geq 0$ is a parameter. The first term in $E$ is called a data fidelity term, and it encourages the minimizer of $E$ (i.e., the denoised signal) to remain close to the noisy signal $f$. The second term aims to measure the amount of noise in the signal $u$, by summing squared differences of neighboring signal values. Clean signals should give much smaller values for this term, compared to noisy signals. There is a parameter $\lambda \geq 0$ in the function $E$ that controls the tradeoff between these two terms. For small $\lambda$ we expect the minimizer of $E$ to remain very close to $f$ and we will thus have very little denoising. For large $\lambda > 0$, we expect to get a very smooth signal with the noise removed, but we may not accurately reconstruct the clean signal. Finding a good value of $\lambda$ that removes noise while preserving important signal information is an important consideration in variational methods.

In this section, we show how to use the theory we have developed for the DFT to efficiently find the minimizer of $E$ in order to denoise signals. For this, it is useful to rewrite the Tikhonov regularized functional $E$. We recall first the definitions of discrete derivatives from Exercise 6.26. The backward difference $\nabla^- : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ is defined by

$\nabla^- u(k) = u(k) - u(k-1),$

while the forward difference is $\nabla^+ : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ is defined by

$\nabla^+ u(k) = u(k+1) - u(k).$

The Laplacian $\Delta : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ is the composition of the forward and backward differences, in either order, so $\Delta = \nabla^+ \nabla^- = \nabla^- \nabla^+$. In terms of the values of $u(k)$, the Laplacian is given by

$\Delta u(k) = u(k+1) - 2u(k) + u(k-1).$
Since our signals are one dimensional, we should think of the Laplacian as just the second derivative of $u$, and indeed, it is exactly the difference quotient approximation of $u''(x)$ for a smooth function $u$.

In terms of this new notation, we can write $E$ as

$$E(u) = \|u - f\|^2 + \lambda \|\nabla^- u\|^2.$$  

The denoising problem is to minimize $E$ over $u \in L^2(\mathbb{Z}_n)$, that is

$$\min_{u \in L^2(\mathbb{Z}_n)} E(u).$$

We now characterize minimizers of (6.16) as the solution of a differential equation known as the Euler-Lagrange equation.

**Theorem 6.28.** Let $\lambda \geq 0$ and $f \in L^2(\mathbb{Z}_n)$. Then there exists a unique solution $u \in L^2(\mathbb{Z}_n)$ of the optimization problem (6.18). Furthermore, the minimizer $u$ is also characterized as the unique solution of the Euler-Lagrange equation

$$u - \lambda \Delta u = f.$$  

Before proving the theorem, we need to record some properties of $\nabla^+$, $\nabla^-$ and $\Delta$. These properties are the discrete analog of integration by parts formulas.

**Proposition 6.29.** For all $u, v \in L^2(\mathbb{Z}_n)$ the following hold.

(i) $\langle \nabla^- u, v \rangle = -\langle u, \nabla^+ v \rangle$

(ii) $\langle \nabla^+ u, v \rangle = -\langle u, \nabla^- v \rangle$

(iii) $\langle \Delta u, v \rangle = \langle u, \Delta v \rangle$

**Proof.** We first prove (i). Let $u, v \in L^2(\mathbb{Z}_n)$ and compute

$$\langle \nabla^- u, v \rangle = \sum_{k=0}^{n-1} \nabla^- u(k)\overline{v(k)}$$

$$= \sum_{k=0}^{n-1} (u(k) - u(k - 1))\overline{v(k)}$$

$$= \sum_{k=0}^{n-1} u(k)\overline{v(k)} - \sum_{k=0}^{n-1} u(k - 1)\overline{v(k)}$$

$$= \sum_{k=0}^{n-1} u(k)\overline{v(k)} - \sum_{k=-1}^{n-2} u(k)\overline{v(k+1)}.$$
Now, since $\mathbb{Z}_n$ is cyclic, we have
\[
\sum_{k=-1}^{n-2} u(k)v(k+1) = \sum_{k=0}^{n-1} u(k)v(k+1)
\]
and so
\[
\langle \nabla^- u, v \rangle = \sum_{k=0}^{n-1} u(k)(v(k) - v(k+1)) = -\langle u, \nabla^+ v \rangle.
\]

To prove (ii), we use Part (i) to obtain
\[
\langle \nabla^+ u, v \rangle = \langle v, \nabla^+ u \rangle = -\langle \nabla^- v, u \rangle = -\langle u, \nabla^- v \rangle.
\]

To prove (iii), since $\Delta = \nabla^+ \nabla^-$ we can use parts (i) and (ii) to find that
\[
\langle \Delta u, v \rangle = \langle \nabla^+ \nabla^- u, v \rangle = -\langle \nabla^- u, \nabla^- v \rangle = \langle u, \nabla^+ \nabla^- v \rangle = \langle u, \Delta v \rangle,
\]
which completes the proof.

**Remark 6.30.** Notice in the proof of Proposition 6.29(iii) that we can take $u = v$ to obtain the identity
\[
\|\nabla^- u\|^2 = \langle \nabla^- u, \nabla^- u \rangle = -\langle \Delta u, u \rangle.
\]
Similarly, we can also write $\|\nabla^+ u\|^2 = -\langle \Delta u, u \rangle$.

**Proof of Theorem 6.28.** The proof is split into 3 parts.

1. We first show that the solution of (6.19) has a unique solution $u \in L^2(\mathbb{Z}_n)$. To see this, we first establish uniqueness. Suppose that $u$ and $v$ satisfy (6.19). Then by subtracting the equations satisfied by $u$ and $v$, we find that $w = u - v$ satisfies
\[
w - \lambda \Delta w = 0.
\]
We take the inner product with $w$ on both sides and use Remark 6.30 to obtain
\[
0 = \langle w - \lambda \Delta w, w \rangle = \langle w, w \rangle - \lambda \langle \Delta w, w \rangle = \|w\|^2 + \lambda \|\nabla^- w\|^2.
\]
Therefore $\|w\|^2 = 0$, and so $w = 0$. Therefore $u = v$ and so the solution, if it exists, is unique.
The existence of a solution to (6.19) now follows from linear algebra, since the equation \( u - \lambda \Delta u = f \) can be written as \( Au = f \) where \( A = I - \lambda \Delta \). The uniqueness proof above shows that the kernel of \( A \) is trivial, so \( A : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n) \) is one-to-one. By the rank-nullity theorem \( A \) is also onto, so for every \( f \in L^2(\mathbb{Z}_n) \) there exists \( u \in L^2(\mathbb{Z}_n) \) such that \( Au = f \). This completes the first step of the proof.

2. We now show that the unique solution of (6.19) solves the optimization problem (6.18). Let \( u \in L^2(\mathbb{Z}_n) \) be the solution of (6.19), which exists due to Part (i). To show that \( u \) solves the optimization problem (6.18), we need to show that for all \( w \in L^2(\mathbb{Z}_n) \) we have

\[
E(u) \leq E(w).
\]

Let \( w \in L^2(\mathbb{Z}_n) \), set \( v = w - u \), and define the function \( e : \mathbb{R} \to \mathbb{R} \) by

\[
e(t) = E(u + tv).
\]

Then we have

\[
E(w) - E(u) = e(1) - e(0) = \int_0^1 e'(t) \, dt.
\]

The goal of the proof is to show that \( e'(t) \geq 0 \), so that \( E(w) - E(u) \geq 0 \). To do this, we first need a simple algebraic identity. For any \( g, h \in L^2(\mathbb{Z}_n) \) we have

\[
\|g + h\|^2 = \langle g + h, g + h \rangle = \langle g, g \rangle + \langle h, g \rangle + \langle g, h \rangle + \langle h, h \rangle.
\]

Since \( \langle h, g \rangle = \overline{\langle g, h \rangle} \) and \( z + \overline{z} = 2 \text{Re} z \) for any \( z \in \mathbb{C} \), we have

\[
\|g + h\|^2 = \|g\|^2 + 2 \text{Re} \langle g, h \rangle + \|h\|^2.
\]

We now apply this to both terms in \( e(t) = E(u + tv) \) to obtain

\[
e(t) = E(u + tv) = \|u + tv - f\|^2 + \lambda \|\nabla^-(u + tv)\|^2
= \|u - f + tv\|^2 + \lambda \|\nabla^- u + t\nabla^- v\|^2
= \|u - f\|^2 + 2t \text{Re} \langle u - f, v \rangle + t^2 \|v\|^2
+ \lambda \|\nabla^- u\|^2 + 2\lambda t \text{Re} \langle \nabla^- u, \nabla^- v \rangle + \lambda t^2 \|\nabla^- v\|^2.
\]

By Proposition 6.29 we have

\[
\langle \nabla^- u, \nabla^- v \rangle = -\langle \nabla^+ \nabla^- u, v \rangle = -\langle \Delta u, v \rangle.
\]

Therefore, we can simplify the expression above to read

\[
e(t) = E(u) + 2t \text{Re} \langle u - \lambda \Delta u - f, v \rangle + t^2 \left( \|v\|^2 + \lambda \|\nabla^- v\|^2 \right).
\]
Differentiating in $t$ we have

\[(6.21) \quad e'(t) = 2 \text{Re} \langle u - \lambda \Delta u - f, v \rangle + 2t \left( \|v\|^2 + \lambda \|\nabla v\|^2 \right). \]

Since $u$ solves (6.19), the first term vanishes, and so $e'(t) \geq 0$ for $t \geq 0$, which completes the proof of this step.

3. Finally, we show that any solution of the optimization problem (6.18) must solve the Euler-Lagrange equation (6.19). Since solutions of (6.19) are unique, this shows that minimizers $E$ are also unique, and completes the proof.

Let $u \in L^2(\mathbb{Z}_n)$ be a solution to the optimization problem (6.18). Let $v \in L^2(\mathbb{Z}_n)$, to be determined later, and as in Part (ii) we define $e(t) = E(u + tv)$. Since $E(u) \leq E(w)$ for all $w \in L^2(\mathbb{Z}_n)$, we have $e(0) = E(u) \leq E(u + tv) = e(t)$. Therefore $e$ has a minimum at $t = 0$ and so $e'(0) = 0$. Using (6.21) we have

\[(6.22) \quad 0 = e'(0) = 2 \text{Re} \langle u - \lambda \Delta u - f, v \rangle. \]

Setting $v = u - \lambda \Delta u - f$ we obtain

\[0 = 2 \text{Re} \|u - \lambda \Delta u - f\|^2, \]

which shows that $u - \lambda \Delta u = f$, as desired.

**Remark 6.31.** A key step in the proof of Theorem (6.19) is differentiating the function $e(t)$, which amounts to computing

\[
\frac{d}{dt} E(u + tv) .
\]

This is called computing a *variation* of $E$ in the direction of $v$, and is a fundamental idea in the *calculus of variations*. In fact, this computation is part of the definition of the *Gateaux derivative* of $E$, and denoted $dE(u) : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$, and defined by

\[
dE(u) v := \left. \frac{d}{dt} \right|_{t=0} E(u + tv) .
\]

The Gateaux derivative is simply the directional derivative of $E$ in the direction $v$. The proof of Theorem 6.28 simply used the fact that minimizers $u$ of $E$ must satisfy $dE(u) v = 0$ for all $v \in L^2(\mathbb{Z}_n)$; that is, their directional derivatives in every direction are zero. It follows from (6.22) that

\[dE(u) v = e'(0) = 2 \text{Re} \langle u - \lambda \Delta u - f, v \rangle. \]

87
The gradient of $E$, denoted $\nabla E$, is defined as the mapping $\nabla E : L^2(\mathbb{Z}_n) \to L^2(\mathbb{Z}_n)$ satisfying

$$dE(u)v = \text{Re}\langle \nabla E(u), v \rangle.$$ 

In this case, the formula for the Gateaux derivative above shows that the gradient of $E$ is

$$\nabla E(u) = u - \lambda \Delta u - f.$$ 

Thus, the Euler-Lagrange equation (6.19) is simply the necessary condition $\nabla E(u) = 0$ satisfied by minimizers of $E$.

The solution of (6.19) satisfies a maximum principle, which says that its values cannot exceed the maximum and minimum values of the noisy signal $f$. This is a desirable property of denoising algorithms.

**Proposition 6.32** (Maximum Principle). Let $\lambda \geq 0$ and let $f \in L^2(\mathbb{Z}_n)$ be real-valued. Let $u \in L^2(\mathbb{Z}_n)$ be the solution of (6.19), which is also real-valued. Then we have

$$(6.23) \quad \min_{\mathbb{Z}_n} f \leq u \leq \max_{\mathbb{Z}_n} f.$$ 

**Proof.** The proof uses a maximum principle argument. Let $k \in \mathbb{Z}_n$ be a point at which $u$ attains its maximum value over $\mathbb{Z}_n$. Then $u(k) \geq u(j)$ for all $j \in \mathbb{Z}_n$, and so

$$\Delta u(k) = u(k+1) - 2u(k) + u(k-1) \leq u(k) - 2u(k) + u(k) = 0.$$ 

Inserting this into (6.19) we have

$$f(k) = u(k) - \lambda \Delta u(k) \geq u(k),$$

and so $\max_{\mathbb{Z}_n} u = u(k) \leq f(k) \leq \max_{\mathbb{Z}_n} f$.

Similarly, if $k \in \mathbb{Z}_n$ is a point where $u$ attains its minimum value over $\mathbb{Z}_n$ then $\Delta u(k) \geq 0$ and we find that $\min_{\mathbb{Z}_n} u = u(k) \geq \min_{\mathbb{Z}_n} f$. \hfill \Box

So far we have not made any connection between denoising and the DFT. We now show how to use the FFT to efficiently solve the Euler-Lagrange equation (6.19) to obtain the denoised signal. This is an example of the application of the FFT to numerically solving partial differential equations. We simply take the DFT on both sides of (6.19) and use linearity of the DFT to obtain

$$(6.24) \quad \mathcal{D}u - \lambda \mathcal{D}(\Delta u) = \mathcal{D}f.$$
We now use Exercise 6.26(iv) to obtain

$$\mathcal{D}(\Delta u)(k) = 2(\cos(2\pi k/n) - 1)\mathcal{D}u(k).$$

This is the main utility of the DFT for solving partial differential equations; it turns derivatives into multiplication. We now substitute this into (6.24) to obtain

$$\mathcal{D}u(k) - 2\lambda(\cos(2\pi k/n) - 1)\mathcal{D}u(k) = \mathcal{D}f(k).$$

We can solve algebraically for $$\mathcal{D}u(k)$$ now, yielding

$$\mathcal{D}u(k) = G_\lambda(k)\mathcal{D}f(k),$$

where

$$G_\lambda(k) = \frac{1}{1 + 2\lambda - 2\lambda \cos(2\pi k/n)}.$$

Taking the inverse transform on both sides we obtain

(6.25) $$u = \mathcal{D}^{-1}(G_\lambda \mathcal{D}f).$$

This gives a very efficient solution method, which solves the discrete differential equation (6.19) in $$O(n \log n)$$ operations using the FFT. Figure 6.4 shows examples of denoised images for various values of $$\lambda$$.

**Remark 6.33.** When using the DFT for signal denoising, the signal is implicitly extended periodically to $$\mathbb{Z}_n$$. This can introduce a sharp discontinuity in the signal, since $$f(0)$$ and $$f(n-1)$$ may not be similar, leading to boundary effects in the denoised signal. To eliminate this, we first take the even extension of the signal (see (6.58)) before applying the solution formula (6.25). The even extension extends a signal periodically without introducing discontinuities. As an alternative, one may use the Discrete Cosine Transform in place of the DFT in the solution formula (6.25). Section 6.8 covers the Discrete Cosine Transform.

The Fourier solution formula (6.25) also gives us insight into how Tikhonov regularization works to denoise a signal. Indeed, from (6.25) we see that each Fourier coefficient $$\mathcal{D}f(k)$$ is simply multiplied by $$G_\lambda(k)$$ before taking an inverse transformation to obtain $$u$$. This acts to attenuate certain frequencies, where $$G_\lambda$$ is small. In signal processing, this type of operation is called filtering. In Figure 6.5(a) we show the Tikhonov filters $$G_\lambda$$ for various values of $$\lambda$$. All the Tikhonov filters are low-pass filters, which means the lower frequencies get multiplied by $$G_\lambda \approx 1$$ and are unchanged, while the higher frequencies are
Figure 6.4: Example of Tikhonov denoising for $\lambda = 0, 1, 10, 50$, using the noisy signal from Figure 6.3.

attenuated. The amount of attenuation of high frequencies increases with $\lambda$. This attenuation is exactly how the noise is removed from the signal; noise typically contains very high frequency components.

We can also rewrite the formula (6.25) in the signal domain using the convolution property (Lemma 6.23). Indeed, defining $g_\lambda = D^{-1}G_\lambda$ so that $G_\lambda = D g_\lambda$ we have

$$u = D^{-1}(D g_\lambda \cdot D f) = g_\lambda * f.$$  

Thus, the denoised signal $u$ is merely the convolution of $g_\lambda$ with $f$. The function $g_\lambda$ is often called a *kernel*. Figure 6.5(b) shows the kernel $g_\lambda$ for various values of $\lambda$. The convolution to compute $u(k)$ should be thought of as a weighted average, weighted by $g_\lambda$, of the values of $f$ nearby $k$. Indeed, we have

$$u(k) = (g_\lambda * f)(k) = \sum_{j=0}^{n-1} g_\lambda(j)f(k-j).$$

As we increase $\lambda$ we can see in Figure 6.5(b) that the width of the kernels increases, meaning we are averaging over a wider neighborhood of the signal, yielding a smoother and less noisy signal.
Tikhonov regularization works well on signals that are smooth and do not have substantial high frequency content. Indeed, as we showed in Section 6.6.1, Tikhonov regularization is just applying a linear filter to the signal that attenuates the high frequencies. Tikhonov regularization does not work well on signals that have sharp changes, like bar codes, or in two or three dimensions, natural images. Here, any sharp change in the signal (say an edge in an image, or a stripe in a bar code) contributes substantially to the higher end of the spectrum, and Tikhonov regularization excessively blurs the edges. Figure 6.6 shows an example of this, using Tikhonov regularization on a piecewise constant signal, with additive Gaussian noise. The sharp transitions in the signal are lost after Tikhonov denoising.

A better approach to denoising is Total Variation regularization, which minimizes the energy

$$E(u) = \frac{1}{2} \sum_{k=0}^{n-1} |u(k) - f(k)|^2 + \lambda \sum_{k=0}^{n-1} |u(k) - u(k-1)|.$$  (6.26)

The second term above is call the Total Variation of the signal $u$. We note that
Figure 6.6: Example of (a) a noisy piecewise constant signal (e.g., a bar code) and (b) the results of Tikhonov regularized denoising with various values of $\lambda$. Tikhonov denoising does not preserve the sharp changes in the signal between regions, and instead overly smoothes these edges.

the only difference between (6.16) and (6.26) is that the Total Variation regularizer takes the sum of the absolute value of the differences $|u(k) - u(k-1)|$, instead of the squares $|u(k) - u(k-1)|^2$ that are used in Tikhonov regularization. Hence, Tikhonov regularization cares far more about minimizing the largest changes in the signal, which smoothes away the edges, while Total Variation does not place any preference over which size changes are minimized. The following exercise explores this.

**Exercise 6.34.** Assume that $f \in L^2(\mathbb{Z}_n)$ is real-valued and increasing; that is $f(k) \leq f(k+1)$ for $k = 0, \ldots, n - 2$.

(i) Show that

$$\sum_{k=1}^{n-1} |f(k) - f(k-1)| = f(n-1) - f(0).$$

Thus, the Total Variation norm does not care how the signal gets from $f(0)$ to $f(n-1)$, as long as the signal is monotone (i.e., not oscillating). Total Variation regularization assigns the same energy to a sharp jump as it does to a continuous ramp, so neither is preferred over the other.

(ii) Consider the Tikhonov regularizer

$$T = \sum_{k=1}^{n-1} |f(k) - f(k-1)|^2,$$
and suppose we fix \( f(0) = 0 \) and \( f(n - 1) = n - 1 \). As above, we assume \( f \) is increasing, so there are no oscillations. How does the Tikhonov energy \( T \) of a sharp jump, where \( f(k) = 0 \) for \( k \leq n/2 \) and \( f(k) = n - 1 \) for \( k > n/2 \), compare to a smooth ramp \( f(k) = k \)? Which ones does Tikhonov regularization prefer?

\( \triangle \)

For the rest of this section we will work with real-valued signals \( f : \mathbb{Z}_n \to \mathbb{R} \). The theory is similar, but needlessly more complicated, for complex valued signals. For this purpose, we define the space \( L^2(\mathbb{Z}_n; \mathbb{R}) \) of real-valued signals \( f : \mathbb{Z}_n \to \mathbb{R} \). Note that \( L^2(\mathbb{Z}_n; \mathbb{R}) \subset L^2(\mathbb{Z}_n) \) is a linear subspace \( L^2(\mathbb{Z}_n) \), so it inherits the inner-product structure and norm, etc.

We will proceed in generality, studying regularizers of the form

\[
(6.27) \quad \sum_{k=0}^{n-1} \Phi(u(k) - u(k - 1)) = \sum_{k=0}^{n-1} \Phi(\nabla^- u(k)),
\]

where \( \Phi : \mathbb{R} \to \mathbb{R} \) is a twice continuously differentiable, convex, and even function satisfying \( \Phi(0) = 0 \). We note that Tikhonov regularization corresponds to \( \Phi(t) = t^2 \), while Total Variation corresponds to \( \Phi(t) = |t| \). Convexity can be defined in many different, but equivalent, ways. Here, we take the definition that \( \Phi'' \geq 0 \), so that \( \Phi' \) is an increasing function. That is \( \Phi'(s) \leq \Phi'(t) \) whenever \( s \leq t \). Since \( \Phi \) is even we have \( \Phi'(0) = 0 \) and so \( \Phi \) is decreasing for \( t < 0 \) and increasing for \( t > 0 \). We also have that

\[
(6.28) \quad (\Phi'(t) - \Phi'(s))(t - s) \geq 0
\]

for all \( s, t \in \mathbb{R} \). This is an important monotonicity property used in the proofs in this section. To see why (6.28) holds, note that if \( t \geq s \), then \( t - s \geq 0 \) and \( \Phi'(t) - \Phi'(s) \geq 0 \) since \( \Phi' \) is increasing, so (6.28) holds. If \( t \leq s \) then \( \Phi'(t) - \Phi'(s) \leq 0 \) and \( t - s \leq 0 \), so (6.28) still holds.

**Remark 6.35.** The assumption that \( \Phi \) is continuously differentiable means that our theory does not apply directly to the Total Variation regularizer, where \( \Phi(t) = |t| \) is not differentiable at \( t = 0 \). However, as we’ll see below when we solve the Total Variation problem numerically, we will approximate it by \( \Phi(t) = \sqrt{t^2 + \varepsilon^2} \) for a small \( \varepsilon > 0 \), which is necessary for numerical stability of our solution scheme. This approximation is a smooth function to which all our theory applies.
In order to simplify notation, we write \( \Phi(\nabla^- u) : L^2(\mathbb{Z}_n; \mathbb{R}) \to L^2(\mathbb{Z}_n; \mathbb{R}) \) for the function
\[
\Phi(\nabla^- u)(k) = \Phi(\nabla^- u(k)).
\]
Then the regularizer above can be written more simply as \( \|\Phi(\nabla^- u)\|_1 \), where we recall the \( \ell_1 \)-norm is given by \( \|f\|_1 = \sum_{k=0}^{n-1} |f(k)| \). The regularized energy for denoising is then given by
\[
E_\Phi(u) = \frac{1}{2} \|u - f\|^2 + \lambda \|\Phi(\nabla^- u)\|_1,
\]
and the denoised signal is found by minimizing \( E_\Phi \). This section aims to study \( E_\Phi \) mathematically, to prove a minimizer exists, is unique, and to find the corresponding Euler-Lagrange equation. Then we will explore how to minimize \( E_\Phi \) numerically. The situation is far different compared to Tikhonov regularization, since here the Euler-Lagrange equation will be nonlinear. The techniques and tools in this section are thus somewhat different, and rely more on nonlinear and convex analysis.

We first prove the existence of a minimizer of \( (6.27) \).

**Lemma 6.36.** For any \( f \in L^2(\mathbb{Z}_n; \mathbb{R}) \) and \( \lambda \geq 0 \), there exists \( u \in L^2(\mathbb{Z}_n; \mathbb{R}) \) minimizing \( (6.29) \), i.e., \( E_\Phi(u) \leq E_\Phi(w) \) for all \( w \in L^2(\mathbb{Z}_n; \mathbb{R}) \). Furthermore, \( u \) satisfies
\[
\min_{\mathbb{Z}_n} f \leq u \leq \max_{\mathbb{Z}_n} f.
\]

**Proof.** Let \( f_{\text{max}} = \max_{\mathbb{Z}_n} f \) and \( f_{\text{min}} = \min_{\mathbb{Z}_n} f \). Define the truncation operator \( T : \mathbb{R} \to \mathbb{R} \) by
\[
T(x) = \begin{cases} 
  f_{\text{min}}, & \text{if } x < f_{\text{min}} \\
  x, & \text{if } f_{\text{min}} \leq x \leq f_{\text{max}} \\
  f_{\text{max}}, & \text{if } x > f_{\text{max}}.
\end{cases}
\]
The truncation operator is Lipschitz continuous with constant 1, that is
\[
|T(x) - T(y)| \leq |x - y|.
\]
To prove this, we may take \( x > y \) without loss of generality and write
\[
|T(x) - T(y)| = T(x) - T(y) = \int_y^x T'(s) \, ds \leq \max_{s \in \mathbb{R}} T'(s) |x - y|.
\]
Note that we can ignore the points where \( T \) is not differentiable, \( x = f_{\text{min}}, f_{\text{max}} \), since we can always split the integral to avoid them. Since at all points where \( T \)
is differentiable we have \( T'(x) = 0 \) or \( T'(x) = 1 \), it follows that \( \max_{s \in \mathbb{R}} T'(s) = 1 \), which establishes the claim.

Let \( u \in L^2(\mathbb{Z}_n; \mathbb{R}) \) and define \( w \in L^2(\mathbb{Z}_n; \mathbb{R}) \) by \( w(k) = T(u(k)) \), that is, by truncating the values of \( u \). We claim that

\[
E_\Phi(w) \leq E_\Phi(u).
\]

That is, \( E_\Phi \) decreases by truncation. To see this, note that for each \( k \), the Lipschitz property of \( T \) yields

\[
|w(k) - f(k)|^2 = |T(u(k)) - T(f(k))|^2 \leq |u(k) - f(k)|^2,
\]

and

\[
|T(u(k)) - T(u(k-1))| \leq |u(k) - u(k-1)|.
\]

Since \( \Phi \) is increasing for \( t \geq 0 \), so we can use the evenness of \( \Phi \) to write

\[
\Phi(w(k) - w(k-1)) = \Phi(|T(u(k)) - T(u(k-1))|) \\
\leq \Phi(|u(k) - u(k-1)|) \\
= \Phi(u(k) - u(k-1)).
\]

This establishes the claim \( (6.31) \).

Now we define the set

\[
M = \{ u \in L^2(\mathbb{Z}_n; \mathbb{R}) : f_{\text{min}} \leq u(k) \leq f_{\text{max}} \text{ for all } k \}.
\]

The function \( E_\Phi \) is continuous and the set \( M \) is closed and bounded (it’s a rectangle when \( L^2(\mathbb{Z}_n; \mathbb{R}) \) is viewed as \( \mathbb{R}^n \)), so \( E_\Phi \) attains its minimum value over \( M \) at some \( u \in M \). To see that \( u \) minimizes \( E_\Phi \) over \( L^2(\mathbb{Z}_n; \mathbb{R}) \), we take any other \( v \in L^2(\mathbb{Z}_n; \mathbb{R}) \) and define the truncation \( w(k) = T(v(k)) \). By \( (6.31) \) we have \( E_\Phi(w) \leq E_\Phi(v) \), and since the truncation \( w \) belongs to \( M \) we have \( E_\Phi(u) \leq E_\Phi(w) \), which completes the proof.

We now prove the solution is unique and characterize the Euler-Lagrange equation.

**Lemma 6.37.** Let \( f \in L^2(\mathbb{Z}_n; \mathbb{R}) \) and \( \lambda \geq 0 \). Then the minimizer \( u \in L^2(\mathbb{Z}_n; \mathbb{R}) \) of \( E_\Phi \) is unique and is characterized as the unique solution of the Euler-Lagrange equation

\[
u - \lambda \nabla^+ \Phi' \nabla^- u = f.
\]

95
Proof. Let \( u \in L^2(\mathbb{Z}_n; \mathbb{R}) \) be a minimizer of \( E_\Phi \), which exists due to Lemma 6.36. We first show that \( u \) satisfies (6.32). We take a variation of \( E_\Phi \), as in the proof of Theorem 6.28 and Remark 6.31. Let \( v \in L^2(\mathbb{Z}_n; \mathbb{R}) \) and define

\[
e(t) := E_\Phi(u + tv).
\]

Then, following a similar argument as in Theorem 6.28 we have

\[
e'(t) = \frac{d}{dt} \left( \frac{1}{2} \|u + tv - f\|^2 + \lambda \sum_{k=0}^{n-1} \Phi(\nabla^- u(k) + t \nabla^- v(k)) \right)
= \langle u - f, v \rangle + t^2 \|v\|^2 + \lambda \sum_{k=0}^{n-1} \Phi'(\nabla^- u(k) + t \nabla^- v(k)) \nabla^- v(k).
\]

Setting \( t = 0 \) and noting that \( e'(0) = 0 \) we have

\[
0 = e'(0) = \langle u - f, v \rangle + \lambda \sum_{k=0}^{n-1} \Phi'(\nabla^- u(k)) \nabla^- v(k)
= \langle u - f, v \rangle + \lambda \langle \Phi'(\nabla^- u), \nabla^- v \rangle
= \langle u - f, v \rangle - \lambda \langle \nabla^+ \Phi'(\nabla^- u), v \rangle
= \langle u - \lambda \nabla^+ \Phi'(\nabla^- u) - f, v \rangle,
\]

where we used Proposition 6.29 in the third step above. Setting \( v = u - \lambda \nabla^+ \Phi'(\nabla^- u) - f \) we find that

\[
\|u - \lambda \nabla^+ \Phi'(\nabla^- u) - f\|^2 = 0
\]

and so \( u \) satisfies (6.32).

The proof will be completed by showing that solutions of (6.32) are unique. Let \( u, v \in L^2(\mathbb{Z}_n; \mathbb{R}) \) be solutions of (6.32). Therefore

\[
u - \lambda \nabla^+ \Phi'(\nabla^- u) - (v - \lambda \nabla^+ \Phi'(\nabla^- u)) = 0,
\]

which can be simplified to read

\[
u - v - \lambda \nabla^+ \left( \Phi'(\nabla^- u) - \Phi'(\nabla^- v) \right) = 0.
\]

Now take the inner product with \( u - v \) on both sides to obtain

\[
\|u - v\|^2 - \lambda \langle \nabla^+ \left( \Phi'(\nabla^- u) - \Phi'(\nabla^- v) \right), u - v \rangle = 0.
\]

96
In the second term, we use the integration by parts formula from Proposition 6.29(ii) to obtain
\[ \|u - v\|^2 + \lambda (\Phi'(\nabla^- u) - \Phi'(\nabla^- v), \nabla^- u - \nabla^- v) = 0. \]
By (6.28), and the assumption that \( \lambda \geq 0 \), we have
\[ \lambda (\Phi'(\nabla^- u) - \Phi'(\nabla^- v), \nabla^- u - \nabla^- v) \geq 0. \]
It follows that \( \|u - v\|^2 = 0 \), and so \( u = v \), which completes the proof.

We now turn to the problem of numerically computing the denoised signal by solving the Euler-Lagrange equation (6.32). This equation is nonlinear, and we can no longer simply take the DFT on both sides to obtain a solution formula, as we did in Section 6.6.1. Here, we use gradient descent to solve (6.32) numerically. Recalling Remark 6.31 and examining the proof of Lemma 6.37 we see that the gradient of \( E_\Phi \) is given by
\[ \nabla E_\Phi(u) = u - \lambda \nabla^+ \Phi' (\nabla^- u) - f. \]
Gradient descent takes small steps in the direction of \( -\nabla E_\Phi \) until convergence.

We start with some initial guess for the denoised signal, \( u_0 \), and iterate
\[ u_{j+1} = u_j - dt (u_j - \lambda \nabla^+ \Phi' (\nabla^- u_j) - f), \tag{6.33} \]
where \( dt > 0 \) is a time step. A good choice for the initial iterate \( u_0 \) is, for example, the solution of Tikhonov denoising given in Section 6.6.1.

The choice of the time step is important. If the time step is taken too large, then the iteration (6.33) is unstable and non-convergent, and if it is chosen too small then the convergence can take prohibitively long. To see how to choose an appropriate time step, we use a Von Neumann analysis, which uses the DFT to analyze stability of numerical schemes. The Von Neumann analysis applies only to linear equations, so it is common in practice (though not rigorous, see below) to approximate the equation by a similar linear equation. To see how to do this, note that the linear Tikhonov setting is \( \Phi(t) = t^2 \), where \( \Phi'(t) = 2t \) and \( \Phi''(t) = 2 \). In general, we can make the approximation
\[ \nabla^+ \Phi' (\nabla^- u)(k) = \Phi' (\nabla^- u(k + 1)) - \Phi' (\nabla^- u(k)) \approx \Phi'' (\nabla^- u(k)) (\nabla^- u(k + 1) - \nabla^- u(k)) = \Phi'' (\nabla^- u(k)) \nabla^+ \nabla^- u(k) = \Phi'' (\nabla^- u(k)) \Delta u(k). \]
Thus, for the Von Neumann analysis we replace $\nabla^2 \Phi(\nabla^- u)$ with $C_\Phi \Delta u$, where $C_\Phi = \max_{t \in \mathbb{R}} |\Phi''(t)|$. We also set $f = 0$ for simplicity; the scheme needs to be stable in this case, and it turns out nonzero choices for $f$ do not affect stability considerations. This yields the simplified linear equation

$$u_{j+1} = u_j - dt (u_j - C_\Phi \lambda \Delta u_j).$$

We perform a Von Neumann analysis by taking the DFT of both sides and using Exercise 6.26 to obtain

$$\mathcal{D}u_{j+1}(k) = \mathcal{D}u_j(k) - dt (\mathcal{D}u_j(k) - 2C_\Phi \lambda (\cos(2\pi k/n) - 1)\mathcal{D}u_j(k))$$

$$= (1 - dt + 2C_\Phi \lambda dt (\cos(2\pi k/n) - 1)) \mathcal{D}u_j(k).$$

Hence, the Fourier modes are simply being multiplied by $\lambda_k$ at each iteration. In order to ensure stability, so the solution does not blow up, we need $|\lambda_k| \leq 1$. Using that $-1 \leq \cos(2\pi k/n) \leq 1$ we have

$$\lambda_k \leq 1 - dt \quad \text{and} \quad \lambda_k \geq 1 - dt (1 + 4C_\Phi \lambda).$$

Thus, to ensure $-1 \leq \lambda_k \leq 1$ for all $k$ we require that $1 - dt (1 + 4C_\Phi \lambda) \geq -1$ or

$$(6.34) \quad dt \leq \frac{2}{1 + 4C_\Phi \lambda}.$$ 

In numerical analysis of partial differential equations, this time step restriction is called a Courant-Friedrichs-Lewy, or CFL, condition.

Figure 6.7 shows the solution of Total Variation (TV) denoising compared to Tikhonov denoising. We chose $\lambda = 0.1$ for TV denoising and $\lambda = 20$ for Tikhonov denoising. These were chosen to yield similar amounts of denoising in the constant regions of the signal. As discussed in Remark 6.35, we approximate the function $\Phi(t) = |t|$ for TV denoising with the smooth approximation $\Phi_\varepsilon(t) = \sqrt{t^2 + \varepsilon^2}$. In this case

$$\Phi_\varepsilon'(t) = \frac{t}{\sqrt{t^2 + \varepsilon^2}}$$

and $0 \leq \Phi_\varepsilon''(t) \leq \frac{1}{\varepsilon}$. Therefore $C_\Phi = \frac{1}{\varepsilon}$, and the time step restriction from the Von Neumann analysis yields

$$dt \leq \frac{2}{1 + 4C_\Phi \lambda} = \frac{2\varepsilon}{\varepsilon + 4\lambda}.$$
Figure 6.7: Total Variation (TV) denoising versus Tikhonov denoising on the piecewise constant signal from Figure 6.6. Total Variation denoising better preserves edges (e.g., sharp changes) in the signal, and is superior at reconstructing piecewise constant signals. We used $\lambda = 0.1$ and $\varepsilon = 10^{-5}$ for TV denoising and $\lambda = 20$ for Tikhonov.

In the experiment we chose $\varepsilon = 10^{-5}$ and ran the gradient descent iterations (6.33) until $u_j$ satisfied (6.32) up to an error tolerance of $10^{-3}$, which took around 70,000 iterations. We can see that choosing $\varepsilon > 0$ is necessary to obtain a numerically stable computational method (though see Remark 6.40 below). There are other methods for minimizing energies with Total Variation regularizers that do not need to make this smooth approximation, for example, primal dual methods [3] and the Split-Bregman approach [7].

It is important to point out that the Von Neumann stability analysis for nonlinear equations like (6.33) is not mathematically rigorous. We made a heuristic simplification to obtain a linear equation, and to make this rigorous we would have to relate the solutions of the linearized equation to the original nonlinear equation (6.33) in some way, which is generally difficult. Nevertheless, it is very common to perform these heuristic stability analyses, since they usually give the correct CFL time step stability condition, and it is very easy to check in practice if the time step restriction is correct, and if it is tight.

In many cases we can say a great deal more if we analyze the gradient descent equation (6.33) using techniques from nonlinear and convex analysis. In Theorem 6.38 below, we give a rigorous proof of convergence of the gradient descent scheme (6.33) under a similar, but more restrictive, time step condition.

**Theorem 6.38.** Let $f \in L^2(\mathbb{Z}_n; \mathbb{R})$ and $\lambda \geq 0$. Let $u_j$ be the iterations of the
gradient descent scheme (6.33) and let \( u \) be the solution of (6.32). Assume that the time step \( dt \) in (6.33) satisfies

\[
 dt < \frac{2}{1 + 16C_\Phi^2 \lambda^2}.
\]

Then \( u_j \) converges to \( u \) as \( j \to \infty \), and the difference \( u_j - u \) satisfies

\[
 ||u_{j+1} - u||^2 \leq \mu ||u_j - u||^2
\]

where

\[
 \mu := (1 - dt)^2 + 16C_\Phi^2 dt^2 \lambda^2 < 1.
\]

**Remark 6.39.** Theorem 6.38 shows that whenever the time step restriction (6.35) holds, gradient descent (6.33) converges at the linear rate \( \mu \) to the solution \( u \) of (6.32). The time step restriction is not strict, and gradient descent may converge for larger time steps, and may exhibit a better convergence rate in practice. Indeed, one often uses the less restrictive Von Neumann time step (6.34) in practice.

**Proof.** We note that the gradient descent equation (6.33) can be written as

\[
 u_{j+1} = (1 - dt)u_j + dt \lambda \nabla^+ \Phi'(\nabla^- u_j) + dt f.
\]

The solution \( u \) of (6.32) satisfies

\[
 u = (1 - dt)u + dt \lambda \nabla^+ \Phi'(\nabla^- u) + dt f.
\]

Subtracting these equations and writing \( e_j = u_j - u \) we obtain

\[
 e_{j+1} = (1 - dt)e_j + dt \lambda \nabla^+ \left( \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u) \right).
\]

We take the squared norm on both sides to obtain

\[
 ||e_{j+1}||^2 = ||(1 - dt)e_j + dt \lambda \nabla^+ \left( \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u) \right)||^2
 = (1 - dt)^2 ||e_j||^2 + dt^2 \lambda^2 ||\nabla^+ \left( \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u) \right)||^2
 + 2dt(1 - dt) \lambda \langle \nabla^+ \left( \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u) \right), u_j - u \rangle
 = (1 - dt)^2 ||e_j||^2 + dt^2 \lambda^2 ||\nabla^+ \left( \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u) \right)||^2
 - 2dt(1 - dt) \lambda \langle \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u), \nabla^- u_j - \nabla^- u \rangle,
\]

where we used the integration by parts formula from Proposition 6.29(i) in the last line. By (6.28), the last term is negative or zero, so we can drop it to obtain the inequality

\[
 ||e_{j+1}||^2 \leq (1 - dt)^2 ||e_j||^2 + dt^2 \lambda^2 ||\nabla^+ \left( \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u) \right)||^2.
\]
We now note that for any $f \in L^2(\mathbb{Z}_n; \mathbb{R})$ we have

$$\|\nabla^+ f\|^2 \leq 4\|f\|^2.$$ 

To see this, we use the inequality $2ab \leq a^2 + b^2$ (obtained by expanding $(a - b)^2 \geq 0$) to compute

$$\|\nabla^+ f\|^2 = \sum_{k=0}^{n-1} (f(k+1) - f(k))^2$$

$$= \sum_{k=0}^{n-1} (f(k+1)^2 - 2f(k+1)f(k) + f(k)^2)$$

$$\leq 2 \sum_{k=0}^{n-1} (f(k+1)^2 + f(k)^2)$$

$$\leq 2(\|f\|^2 + \|f\|^2) = 4\|f\|^2.$$

Likewise we have $\|\nabla^- f\|^2 \leq 4\|f\|^2$. Using this estimate in (6.38) we have

$$(6.39) \quad \|e_{j+1}\|^2 \leq (1 - dt)^2\|e_j\|^2 + 4dt^2\lambda^2\|\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u)\|^2$$

Since

$$\Phi'(t) - \Phi'(s) = \int_s^t \Phi''(\tau) d\tau \leq C_\Phi(t - s)$$

for $t \geq s$ (recall $C_\Phi = \max \Phi''$), we have

$$|\Phi'(t) - \Phi'(s)| \leq C_\Phi|t - s|$$

for any $t, s \in \mathbb{R}$. Therefore

$$\|\Phi'(\nabla^- u_j) - \Phi'(\nabla^- u)\|^2 \leq C_\Phi\|\nabla^- u_j - \nabla^- u\|^2 = C_\Phi^2\|\nabla^- e_j\|^2 \leq 4C_\Phi\|e_j\|^2.$$

Inserting this into (6.39) we have

$$\|e_{j+1}\|^2 \leq (1 - dt)^2\|e_j\|^2 + 16C_\Phi^2dt^2\lambda^2\|e_j\|^2$$

$$\leq ((1 - dt)^2 + 16C_\Phi^2dt^2\lambda^2)\|e_j\|^2.$$

Thus, gradient descent converges when

$$\mu := (1 - dt)^2 + 16C_\Phi^2dt^2\lambda^2 < 1,$$

Since in this case we have $\|e_j\|^2 \leq \mu^j\|e_0\|^2 \to 0$ as $j \to \infty$. The proof is completed by checking that $\mu < 1$ whenever

$$\Box$$

101
Remark 6.40. It turns out it is possible to set $\varepsilon = 0$, i.e., use the true TV regularizer $\Phi(t) = |t|$, and still obtain a stable numerical method that results in good denoising results (and converges rapidly). In Figure 6.8 we show the denoising results with various values of $dt$ and $\varepsilon = 10^{-10}$. In this case the CFL condition gives a stability condition of $dt \leq 5 \times 10^{-10}$, and we consider much larger time steps of $dt = 0.01, 0.05, 0.1, 0.5$. In each case the gradient descent does not converge, but it does settle down to a steady state solution with some persisting oscillations. For $dt = 0.01$ the oscillations are quite small and could
be acceptable for a denoising result.

This is a remarkable phenomenon that does not hold for linear equations, where the Von Neumann analysis gives sharp CFL time step stability conditions that cannot be violated without finite time blow up of the solution. Here, there are nonlinear effects that prevent oscillations from growing beyond a certain amplitude. Even for larger time steps $dt$, the gradient descent iterations never grow unbounded, and instead the oscillations in the signal grow to a certain amplitude, dependent on the choice of $dt$, and then they remain that large and continue to oscillate. The scheme is nonconvergent, but if the oscillations are sufficiently small, they would be within the quantization error from saving a signal or image as 8-bit or 16-bit integers, which is commonly done in practice.

While there is not a rigorous mathematical understanding of this phenomenon, we can perform a rough heuristic argument that gives some good insight. Essentially, in the CFL condition (6.34) we took a very loose upper bound by setting $C_\Phi = \max \Phi''$. Recall we used the approximation

$$\nabla^+ \Phi'(\nabla^- u)(k) \approx \Phi''(\nabla^- u(k)) \Delta u(k)$$

to obtain a linear equation, but we replaced $\Phi''$ by $C_\Phi$. Instead, we can leave $\Phi''$ and treat the equation locally, so the equivalent linear equation becomes

$$u_{j+1} = u_j - dt \left( u_j - \lambda \Phi''(\nabla^- u)\Delta u_j \right).$$

We treat this equation locally, so $\Phi''$ is roughly constant, the CFL condition (6.34) becomes

$$(6.40) \quad dt \leq \frac{2}{1 + 4\Phi''(\nabla^- u)\lambda}.$$ 

Now, we flip the CFL condition around and write it as a condition on $\Phi''$, in the form

$$(6.41) \quad \Phi''(\nabla^- u) \leq \frac{1}{4\lambda} \left( \frac{2}{dt} - 1 \right).$$

From this, we can see that if we pick a time step $dt$, then the scheme will be stable locally wherever the gradient $\nabla^- u$ satisfies (6.41). Hence, some parts of the solution will be stable, while others will be unstable and have growing oscillations. But as the oscillations grow, the CFL condition (6.41) changes, leading to a nonlinear interplay between the gradient and time step stability condition.
For example, with the TV regularizer with \( \varepsilon > 0 \), i.e., \( \Phi(t) = \sqrt{t^2 + \varepsilon^2} \), we have
\[
\Phi''(t) = \frac{\varepsilon^2}{(t^2 + \varepsilon^2)^{3/2}} \leq \frac{\varepsilon^2}{t^3}.
\]
Therefore, the condition (6.41) holds whenever
\[
\frac{\varepsilon^2}{|\nabla^- u|^3} \leq \frac{1}{4\lambda} \left( \frac{2}{dt} - 1 \right),
\]
which can be rewritten as
\[\text{(6.42)} \quad |\nabla^- u|^3 \geq \frac{4\lambda \varepsilon^2 dt}{2 - dt}.\]
Hence, oscillations will grow until the gradient \( \nabla^- u \) is large enough so that (6.42) holds, and then the scheme becomes stable and those oscillations will stop growing. This gives a heuristic explanation for Figure 6.8. For more details on this phenomenon, we refer the reader to [1], where this phenomenon is described in more detail.

**Exercise 6.41.** In the proof of Theorem 6.38 we only used that \( \Phi \) is convex, namely \( \Phi'' \geq 0 \). In general, \( \Phi \) is often strongly convex, which means \( \Phi'' \geq c \Phi \) for some positive constant \( c \Phi > 0 \). In this exercise we assume \( \Phi \) is strongly convex with constant \( c_\Phi \).

(i) Show that for every \( t, s \in \mathbb{R} \) we have
\[
(\Phi'(t) - \Phi'(s))(t - s) \geq c_\Phi (t - s)^2.
\]
Use this to show that (in the context of the proof of Theorem 6.38) that
\[
\langle \Phi'(\nabla^- u_j) - \Phi'(\nabla^- u), \nabla^- u_j - \nabla^- u \rangle \geq c_\Phi \|\nabla^- e_j\|^2.
\]

(ii) Modify the proof of Theorem 6.38 using Part (i) instead of (6.28) to show that
\[
\|u_{j+1} - u\|^2 \leq (1 - dt)^2 \|u_j - u\|^2
\]
provided the time step is restricted so that
\[
dt \leq \frac{c_\Phi}{c_\Phi + 2C_\Phi \lambda}.
\]
(iii) Show that if $C_\Phi = c_\Phi$ (which means that $\Phi(t) = \frac{1}{2}C_\Phi t^2$) then the time step restriction becomes

$$dt \leq \frac{1}{1 + 2C_\Phi \lambda},$$

which is only slightly more restrictive than the Von Neumann condition (6.34).

\(\triangle\)

### 6.7 Multi-dimensional DFT

We now briefly study the Discrete Fourier Transform (DFT) in higher dimensions. Let

$$\mathbb{Z}^d_n = \mathbb{Z}_n \times \mathbb{Z}_n \times \cdots \times \mathbb{Z}_n \text{ \(d\) times},$$

where $d \geq 1$. Elements of $\mathbb{Z}^d_n$ are denoted by the same letters, i.e., $k, \ell$, as elements of $\mathbb{Z}_n$. In this case $k \in \mathbb{Z}^d_n$ has $d$ components $k = (k(1), k(2), \ldots, k(d))$. We denote the dot product of $k, \ell \in \mathbb{Z}^d_n$ by

$$k \cdot \ell = \sum_{j=1}^{d} k(j)\ell(j).$$

We denote by $L^2(\mathbb{Z}^d_n)$ the space of function $f : \mathbb{Z}^d_n \to \mathbb{C}$ equipped with the inner product

$$\langle f, g \rangle = \sum_{k \in \mathbb{Z}^d_n} f(k)\overline{g(k)}.$$  

We also have the induced norm $\|f\|^2 = \langle f, f \rangle$. The discrete cyclic convolution of $f, g \in L^2(\mathbb{Z}^d_n)$ is given by

$$(f * g)(k) = \sum_{\ell \in \mathbb{Z}^d_n} f(\ell)g(k - \ell).$$

Note that the sums above are short-hand for $d$ sums, and we have

$$\sum_{k \in \mathbb{Z}^d_n} = \sum_{k(1)=0}^{n-1} \sum_{k(2)=0}^{n-1} \cdots \sum_{k(d)=1}^{n-1}. $$

105
Definition 6.42. The (multi-dimensional) Discrete Fourier Transform (DFT) is the mapping $D : L^2(\mathbb{Z}_n^d) \to L^2(\mathbb{Z}_n^d)$ given by

\[(6.43)\quad Df(k) = \sum_{\ell \in \mathbb{Z}_n^d} f(\ell)e^{-2\pi ik\cdot\ell/n}.\]

The (multi-dimensional) Inverse Discrete Fourier Transform (IDFT) is the mapping $D^{-1} : L^2(\mathbb{Z}_n^d) \to L^2(\mathbb{Z}_n^d)$ given by

\[(6.44)\quad D^{-1}f(\ell) = \frac{1}{n^d} \sum_{k \in \mathbb{Z}_n^d} f(k)e^{2\pi ik\cdot\ell/n}.\]

We note that the same symbols $D$ and $D^{-1}$ are used for the one dimensional and higher dimensional DFTs. The value of $d$ will normally be clear from context.

It is important to point out that the multi-dimensional DFT can be viewed as applying $d$ one dimensional DFTs to the individual coordinates. Indeed, we consider the case of $d = 2$ where we can write

\[
Df(k) = Df(k(1), k(2)) \\
= \sum_{\ell(1)=0}^{n-1} \sum_{\ell(2)=0}^{n-1} f(\ell(1), \ell(2))e^{-2\pi i(k(1)\ell(1)+k(2)\ell(2))/n} \\
= \sum_{\ell(1)=0}^{n-1} e^{-2\pi ik(\ell(1))/n} \left( \sum_{\ell(2)=0}^{n-1} f(\ell(1), \ell(2))e^{-2\pi i\ell(2)/n} \right).
\]

The term in brackets above is the one dimensional DFT of $f$ in the second coordinate $\ell(2)$, and the outer sum computes the one dimensional DFT of the result in the first coordinate. A similar observation can be made for the case of $d \geq 3$. This means most properties of the one dimensional DFT carry over to the multi-dimensional case, and the FFT can be directly used to efficiently compute multi-dimensional DFTs.

We will briefly review the important properties of the multi-dimensional DFT. The proofs are very similar to the one dimensional setting, so we leave them to exercises.

Theorem 6.43. For every $f \in L^2(\mathbb{Z}_n^d)$ we have $f = DD^{-1}f = D^{-1}Df$. Furthermore, the following properties hold for each $f, g \in L^2(\mathbb{Z}_n^d)$.

(i) $\langle f, g \rangle = \frac{1}{n^d}\langle Df, Dg \rangle$,
(ii) \( \|f\|_2^2 = \frac{1}{n^d} \|Df\|_2^2 \),

(iii) \( D(f \ast g) = Df \cdot Dg \).

**Exercise 6.44.** Prove Theorem 6.43. \( \triangle \)

The discrete derivatives introduced in Section 6.6.1 can be extended to multi-dimensional versions. Let \( e_1, e_2, \ldots, e_d \in \mathbb{R}^d \) be the standard basis vectors in \( \mathbb{R}^d \). We define the forward difference in the \( j \)th direction, \( \nabla_j^+ : L^2(\mathbb{Z}_n^d) \to L^2(\mathbb{Z}_n^d) \), by

\[
\nabla_j^+ u(k) = u(k + e_j) - u(k).
\]

Similarly, the backward difference \( \nabla_j^- \) by

\[
\nabla_j^- u(k) = u(k) - u(k - e_j).
\]

The discrete Laplacian \( \Delta \) is defined by

\[
\Delta u = \sum_{j=1}^d \nabla_j^+ \nabla_j^- u.
\]

The analogous result to Proposition 6.29, concerning discrete integration by parts, holds in the multi-dimensional setting.

**Proposition 6.45.** For all \( u, v \in L^2(\mathbb{Z}_n^d) \) and \( j = 1, 2, \ldots, d \), the following hold.

(i) \( \langle \nabla_j^- u, v \rangle = -\langle u, \nabla_j^+ v \rangle \)

(ii) \( \langle \nabla_j^+ u, v \rangle = -\langle u, \nabla_j^- v \rangle \)

(iii) \( \langle \Delta u, v \rangle = \langle u, \Delta v \rangle \)

**Exercise 6.46.** Prove Proposition 6.45. \( \triangle \)

The DFTs of the multi-dimensional gradients and Laplacian are similar to their one dimensional counterparts. We leave the results to an exercise, which is the generalization of Exercise 6.26.

**Exercise 6.47.** Complete the following exercises.

(i) Show that \( D(\nabla_j^- f)(k) = (1 - \omega^{-k(j)})Df(k) \), where \( \omega = e^{2\pi i/n} \).

(ii) Show that \( D(\nabla_j^+ f)(k) = (\omega^{k(j)} - 1)Df(k) \).

(iii) Show that

\[
D(\Delta f)(k) = 2Df(k) \sum_{j=1}^d (\cos(2\pi k(j)/n) - 1).
\]

\( \triangle \)
6.7.1 Application: Image denoising

Python Notebook: .ipynb

We now consider an application of the multi-dimensional DFT to image denoising. This section follows closely Sections 6.6.1 and 6.6.2, so we only sketch the results here, leaving some of the proofs to exercises. While most images are in $d = 2$ or $d = 3$ dimensions, we proceed in generality here.

We consider the general regularized variational approach to denoising, as in Section 6.6.2, where we minimize the function

\[
E_{\Phi}(u) = \frac{1}{2} \|u - f\|^2 + \lambda \sum_{j=1}^d \|\Phi(\nabla_j u)\|_1,
\]

where $f \in L^2(\mathbb{Z}_n^d)$ is the noisy image and the denoised image is the minimizer $u$. The function $\Phi : \mathbb{R} \to \mathbb{R}$ is the regularizer, which we recall from Section 6.6.2 is assumed to be twice continuously differentiable, convex, and satisfies $\Phi(0) = 0$. We also recall that since these conditions imply $\Phi$ is nonnegative, we have

\[
\|\Phi(\nabla_j u)\|_1 = \sum_{k \in \mathbb{Z}_n^d} \Phi(\nabla_j u(k)).
\]

The choice of $\Phi(t) = \frac{1}{2} t^2$ leads to Tikhonov image denoising, while $\Phi(t) = |t|$ (or the regularized $\Phi(t) = \sqrt{t^2 + \varepsilon^2}$) leads to Total Variation (TV) regularization.

To compute the gradient of $E_{\Phi}$, we follow the notation in Remark 6.31 and the proof of Lemma 6.37 to obtain

\[
\left. \frac{d}{dt} \right|_{t=0} E_{\Phi}(u + tv) = \langle u - f, v \rangle + \lambda \sum_{j=1}^d \langle \Phi'(\nabla_j^- u), \nabla_j^- v \rangle,
\]

for any $v \in L^2(\mathbb{Z}_n^d)$.

**Exercise 6.48.** Prove that (6.46) holds. \(\triangle\)

We use Proposition 6.45 to obtain

\[
\left. \frac{d}{dt} \right|_{t=0} E_{\Phi}(u + tv) = \langle u - f, v \rangle - \lambda \sum_{j=1}^d \langle \nabla_j^+ \Phi'(\nabla_j^- u), v \rangle = \langle \nabla E_{\Phi}(u), v \rangle,
\]

where

\[
\nabla E_{\Phi}(u) = u - f - \lambda \sum_{j=1}^d \nabla_j^+ \Phi'(\nabla_j^- u).
\]
Therefore, the Euler-Lagrange equation $\nabla E_{\Phi}(u) = 0$ becomes

\[(6.47) \quad u - \lambda \sum_{j=1}^{d} \nabla_j^+ \Phi'(\nabla_j^- u) = f.\]

In the case of Tikhonov regularization, where $\Phi(t) = \frac{1}{2} t^2$, the equation becomes linear and we have

\[(6.48) \quad u - \lambda \Delta u = f.\]

For Tikhonov regularization, we solve the denoising equation (6.48) by taking the DFT on both sides and using Exercise 6.47 to obtain

$$\left(1 - 2\lambda \sum_{j=1}^{d} (\cos(2\pi k(j)/n) - 1) \right) \mathcal{D}u(k) = \mathcal{D}f(k).$$

Solving for $u$ we obtain

\[(6.49) \quad u = \mathcal{D}^{-1}(G_{\lambda} \cdot \mathcal{D}f) = g_{\lambda} * f,\]

where

$$G_{\lambda}(k) = \left(1 - 2\lambda \sum_{j=1}^{d} (\cos(2\pi k(j)/n) - 1) \right)^{-1}$$

and $G_{\lambda} = \mathcal{D}g_{\lambda}$.

For general nonlinear $\Phi$, we solve (6.47) with gradient descent, which iterates

\[(6.50) \quad u_{j+1} = u_j - dt \left( u_j - \lambda \sum_{m=1}^{d} \nabla_m^+ \Phi' (\nabla_m^- u_j) - f \right).\]

To determine the CFL stability condition for the time step $dt$, we follow a Von Neumann analysis for the simplified linear equation

$$u_{j+1} = u_j - dt \left( u_j - \lambda C_{\Phi} \sum_{m=1}^{d} \nabla_m^+ \nabla_m^- u_j - f \right),$$

which is equivalent, after dropping the forcing term $f$, to

\[(6.51) \quad u_{j+1} = (1 - dt)u_j + dt C_{\Phi} \lambda \Delta u_j.\]
Taking the multi-dimensional DFT on both sides and using Exercise 6.47 we find that
\[ D_{u_{j+1}}(k) = \left(1 - dt + 2dtC_\Phi \lambda \sum_{m=1}^{d} (\cos(2\pi k(m)/n) - 1)\right) D_{u_j}(k). \]

We need \(|\lambda_k| \leq 1\) for all \(k \in \mathbb{Z}_n^d\) to ensure stability. Note that
\[ 1 - dt(1 + 4dC_\Phi \lambda) \leq \lambda_k \leq 1. \]

Therefore, to ensure \(\lambda_k \geq -1\) we obtain the CFL condition
\[ \text{(6.52)} \quad dt \leq \frac{2}{1 + 4dC_\Phi \lambda}. \]

Figure 6.9 compares Tikhonov regularization to Total Variation regularization. Notice the edges are better preserved in TV denoising. Figure 6.10 shows the results of TV denoising with larger values for \(\lambda\), which favor the regularizer more heavily and lead to smoother images.

**Exercise 6.49.** Formulate and prove the analogous results to Lemma 6.36, Lemma 6.37, and Theorem 6.38 in the multi-dimensional setting. What is the corresponding time step restriction, analogous to (6.35)? \(\triangle\)

**Project 6.1 (TV Image Inpainting). Python Notebook: .ipynb**

A common problem in image processing is to fill in missing pieces of an image, which may have been corrupted by writing on an image of damage to a painting, for example. It may also be desirable to remove an object from an image, which can be treated by the same methods. The process of filling in missing parts of an image is called image inpainting. Figure 6.11 shows an example of part of the cameraman image that has been corrupted by writing "Math 5467" on the image, along with the result of TV regularized inpainting (from this project) to fill in the missing parts of the image.

For image inpainting, we assume we know the region of the image that is corrupted and needs to be inpainted. Let \(f \in L^2(\mathbb{Z}_n^d)\) denote the corrupted image, and let \(\Lambda \in \mathbb{Z}_n^d\) denote the set of uncorrupted (i.e., good) pixels. The TV regularized inpainting problem is to minimize
\[ \text{(6.53)} \quad E_\Phi(u) = \frac{1}{2} \|\delta_\Lambda \cdot (u - f)\|^2 + \lambda \sum_{j=1}^{d} \|\Phi(\nabla_j u)\|_1, \]
where $\delta_\Lambda(k) = 1$ if $k \in \Lambda$ and $\delta_\Lambda(k) = 0$ otherwise. Essentially we treat inpainting as a denoising problem where the image $f$ is known on only part of the domain.

Complete the following steps for this project, using the Python notebook linked above.
Figure 6.10: Examples of TV denoising with larger values of $\lambda$.

Figure 6.11: Example of a corrupted image and the result of TV inpainting.

(i) Show that

$$\nabla E_\Phi(u) = \delta_\lambda(u - f) - \lambda \sum_{j=1}^{d} \nabla_j^+ \Phi'(\nabla_j^- u).$$

(ii) Use gradient descent to minimize the TV inpainting functional to inpaint the corrupted image in Figure 6.11. Note that the CFL condition on the time step $dt$ in gradient descent is the same as in TV denoising.

Project 6.2 (TV Image Deblurring). Python Notebook: .ipynb

Another common task in image processing is image deblurring (i.e., deconvolution). An image blur is the convolution of an image $f$ with a blurring kernel $g$,

$$f_{\text{blur}} = g \ast f.$$
Figure 6.12: An example of an out-of-focus blurring of the cameraman image.

For example, an out-of-focus blur, the blurring kernel \( g \) is the characteristic function of a disk,

\[
g(k) = \begin{cases} 
1, & \text{if } \|k\| \leq r \\
0, & \text{otherwise.}
\end{cases}
\]

The radius \( r \) of the disk controls the amount of blur. Any convolution-based blurring operation can be computed with a DFT using the convolution property, as

\[
f_{\text{blur}} = D^{-1}(G \cdot Df),
\]

where \( G = D^{-1}g \) is the filter.

Complete the following steps for this project, using the Python notebook linked above.

(i) Try a naive deblurring based on inverting the filter \( G \). That is, try computing

\[
f_{\text{deblurred}} = D^{-1}(G^{-1} \cdot Df_{\text{blur}}).
\]

(ii) You should find the naive deblurring works well in a perfectly noise-free setting. Try adding noise after blurring, in the form

\[
f_{\text{blur}+\text{noise}} = f_{\text{blur}} + \eta
\]
where $\eta$ is independent Gaussian noise (each $\eta(k)$ is an Gaussian random variable with mean zero and standard deviation $\sigma > 0$). Then try the naive deblurring in this noisy setting, which is

$$f_{\text{deblurred}} = \mathcal{D}^{-1}(G^{-1} \cdot \mathcal{D}f_{\text{blur+noise}}).$$

What do you see? While the blurring operator is technically invertible, it has vanishingly small eigenvalues and taking the inverse is not stable in the presence of noise. Noise is always present in signal acquisition, and it is impossible to work in noise-free settings, like in Part (i), in practice.

(iii) In order to deblur in the presence of noise, we use Total Variation regularization, which minimizes the function

$$E_\Phi(u) = \frac{1}{2} \|g \ast u - f\|^2 + \lambda \sum_{j=1}^{d} \|\Phi(\nabla_j^- u)\|_1,$$

where $f = f_{\text{blur+noise}}$. Show that

$$\nabla E_\Phi(u) = g^T \ast (g \ast u - f) - \lambda \sum_{j=1}^{d} \nabla_j^+ \Phi'(\nabla_j^- u),$$

where $g^T := \mathcal{D}^{-1}G$. [Hint: Take a variation, and then use Parseval’s identity to simplify.]

(iv) Use gradient descent to minimize the TV regularized deblurring energy $E_\Phi$ and deblur the image. Note that the CFL time step restriction is the same as in TV denoising.

6.8 The Discrete Cosine and Sine Transforms

It is often useful in practical applications to avoid complex numbers and work with real-valued transformations. Indeed, many embedded systems do not have support for complex arithmetic, and due to the conjugate symmetry

$$\mathcal{D}f(\ell) = \overline{\mathcal{D}f(n-\ell)}$$

for real valued signals $f$, the FFT computes twice as many coefficients are are needed for real-valued signals, which is wasteful.
To motivate the Discrete Cosine and Sine Transforms, we recall the Fourier Inversion Theorem 6.6 yields

\[(6.55) \quad f(k) = \frac{1}{n} \sum_{\ell=0}^{n-1} Df(\ell)e^{2\pi ik\ell/n}.\]

Throughout this section we assume that \(f\) is real-valued, that is \(f(k) \in \mathbb{R}\) for all \(k\). In this case, the imaginary part of the right hand side above must be identically zero. The remaining real part of the right hand side will give us a representation formula for \(f\) in terms of only real-value quantities, and will lead us to the Discrete Cosine and Sine Transforms.

To do this, we substitute the definition of \(Df(\ell)\) into (6.55) to obtain

\[(6.56) \quad f(k) = \frac{1}{n} \sum_{\ell=0}^{n-1} \sum_{j=0}^{n-1} f(j) e^{-2\pi ij\ell/n} e^{2\pi ik\ell/n}.\]

We use Euler’s Formula to simplify the complex exponentials as follows

\[e^{-2\pi ij\ell/n} e^{2\pi ik\ell/n} = (\cos(2\pi j\ell/n) - i \sin(2\pi j\ell/n))(\cos(2\pi k\ell/n) + i \sin(2\pi k\ell/n))\]

\[= (\cos(2\pi j\ell/n) \cos(2\pi k\ell/n) + \sin(2\pi j\ell/n) \sin(2\pi k\ell/n))\]

\[+ i (\cos(2\pi j\ell/n) \sin(2\pi k\ell/n) - \sin(2\pi j\ell/n) \cos(2\pi k\ell/n)).\]

Since \(f(k)\) is real-valued, we can ignore the imaginary part above (it must vanish when summed in (6.56)), and substitute the real part into (6.56) to obtain

\[(6.57) \quad f(k) = \frac{1}{n} \sum_{\ell=0}^{n-1} \left( \sum_{j=0}^{n-1} f(j) \cos(2\pi j\ell/n) \right) \cos(2\pi k\ell/n)\]

\[+ \frac{1}{n} \sum_{\ell=0}^{n-1} \left( \sum_{j=0}^{n-1} f(j) \sin(2\pi j\ell/n) \right) \sin(2\pi k\ell/n).\]

This gives a representation of \(f\) in a basis of real-valued \(\cos\) and \(\sin\) functions.

To derive the Discrete Cosine Transform from this, we play a trick to remove the \(\sin\) terms. The trick is to take an even extension of the function \(f\) to a domain that is (nearly) twice as large. That is, starting with \(f: \mathbb{Z}_n \rightarrow \mathbb{R}\), we define the even extension \(f_e: \mathbb{Z}_{2(n-1)} \rightarrow \mathbb{R}\) by

\[(6.58) \quad f_e(k) = \begin{cases} f(k), & \text{if } 0 \leq k \leq n-1, \\ f(2(n-1) - k), & \text{if } n \leq k \leq 2(n-1) - 1. \end{cases}\]
The reason for using $\mathbb{Z}_{2(n-1)}$ instead of $\mathbb{Z}_{2n}$ is that we do not wish to duplicate the first and last points $f(0)$ and $f(n-1)$ when reflecting evenly. Figure 6.13 shows an example of the even extension $f_e$ of a signal $f \in L^2(\mathbb{Z}_{64})$. The figure also shows the odd extension, which is defined in Exercise 6.51.

We now apply the representation formula (6.57) to the even extension $f_e$, taking $2(n-1)$ in place of $n$, to obtain

\begin{equation}
\begin{aligned}
f_e(k) &= \frac{1}{2(n-1)} \sum_{\ell=0}^{2(n-1)-1} A_\ell \cos \left( \frac{\pi k \ell}{n-1} \right) \\
&\quad + \frac{1}{2(n-1)} \sum_{\ell=0}^{2(n-1)-1} B_\ell \sin \left( \frac{\pi k \ell}{n-1} \right),
\end{aligned}
\end{equation}

where

\begin{equation}
A_\ell = \sum_{j=0}^{2(n-1)-1} f_e(j) \cos \left( \frac{\pi j \ell}{n-1} \right) \quad \text{and} \quad B_\ell = \sum_{j=0}^{2(n-1)-1} f_e(j) \sin \left( \frac{\pi j \ell}{n-1} \right).
\end{equation}

Since we took the even extension of $f$, it turns out that $B_\ell = 0$. Indeed, we
compute
\[ B_\ell = \sum_{j=0}^{2(n-1)-1} f_\ell(j) \sin \left( \frac{\pi j \ell}{n-1} \right) \]
\[ = \sum_{j=0}^{n-1} f_\ell(j) \sin \left( \frac{\pi j \ell}{n-1} \right) + \sum_{j=n}^{2(n-1)-1} f_\ell(j) \sin \left( \frac{\pi j \ell}{n-1} \right) \]
\[ = \sum_{j=0}^{n-1} f(j) \sin \left( \frac{\pi j \ell}{n-1} \right) + \sum_{j=n}^{2(n-1)-1} f(2(n-1) - j) \sin \left( \frac{\pi j \ell}{n-1} \right) \]
\[ = \sum_{j=0}^{n-1} f(j) \sin \left( \frac{\pi j \ell}{n-1} \right) + \sum_{k=1}^{n-2} f(k) \sin \left( \frac{\pi(2(n-1) - k) \ell}{n-1} \right), \]
where we made the change of variables \( k = 2(n-1) - j \) in the last line. We now note that
\[ \sin \left( \frac{\pi(2(n-1) - k) \ell}{n-1} \right) = \sin \left( \pi \ell - \frac{\pi k \ell}{n-1} \right) = -\sin \left( \frac{\pi k \ell}{n-1} \right). \]
Substituting this above we obtain
\[ B_\ell = \sum_{j=0}^{n-1} f(j) \sin \left( \frac{\pi j \ell}{n-1} \right) - \sum_{k=1}^{n-2} f(k) \sin \left( \frac{\pi k \ell}{n-1} \right) \]
\[ = f(0) \sin(0) + f(n-1) \sin(\pi \ell) = 0. \]

We can in fact make a similar argument to simplify \( A_\ell \). Following the argument above yields
\[ A_\ell = \sum_{j=0}^{n-1} f(j) \cos \left( \frac{\pi j \ell}{n-1} \right) + \sum_{k=1}^{n-2} f(k) \cos \left( \frac{\pi(2(n-1) - k) \ell}{n-1} \right). \]
We now use the fact that
\[ \cos \left( \frac{\pi(2(n-1) - k) \ell}{n-1} \right) = \cos \left( 2\pi \ell - \frac{\pi k \ell}{n-1} \right) = \cos \left( \frac{\pi k \ell}{n-1} \right) \]
to obtain

\begin{equation}
(6.61) \quad A_\ell = \sum_{j=0}^{n-1} f(j) \cos \left( \frac{\pi j \ell}{n-1} \right) + \sum_{k=1}^{n-2} f(k) \cos \left( \frac{\pi k \ell}{n-1} \right)
= f(0) \cos(0) + f(n-1) \cos(\pi \ell) + 2 \sum_{j=1}^{n-2} f(j) \cos \left( \frac{\pi j \ell}{n-1} \right),
\end{equation}

Now, we aim to substitute \( B_\ell = 0 \) and the expression for \( A_\ell \) into (6.59) to get
the Discrete Cosine Transform. However, the nonzero part of the expression in (6.59) still needs some simplification. We compute

\begin{equation}
\sum_{\ell=0}^{2(n-1)-1} A_\ell \cos \left( \frac{\pi k \ell}{n-1} \right) = \sum_{\ell=0}^{n-1} A_\ell \cos \left( \frac{\pi k \ell}{n-1} \right) + \sum_{\ell=n}^{2(n-1)-1} A_\ell \cos \left( \frac{\pi k \ell}{n-1} \right)
= \sum_{\ell=0}^{n-1} A_\ell \cos \left( \frac{\pi k \ell}{n-1} \right) + \sum_{j=1}^{n-2} A_{2(n-1)-j} \cos \left( \frac{\pi kj}{n-1} \right)
= A_0 + (-1)^k A_{n-1} + \sum_{\ell=1}^{n-2} (A_\ell + A_{2(n-1)-\ell}) \cos \left( \frac{\pi k \ell}{n-1} \right).
\end{equation}

By Exercise 6.50, below, we finally arrive at

\begin{equation}
(6.62) \quad \sum_{\ell=0}^{2(n-1)-1} A_\ell \cos \left( \frac{\pi k \ell}{n-1} \right) = A_0 + (-1)^k A_{n-1} + 2 \sum_{\ell=1}^{n-2} A_\ell \cos \left( \frac{\pi k \ell}{n-1} \right).
\end{equation}

Exercise 6.50. Show that \( A_\ell = A_{2(n-1)-\ell} \). \( \triangle \)

Substituting \( B_\ell = 0 \), (6.61) and (6.62) into (6.59) and restricting to \( 0 \leq k \leq n-1 \), where \( f_\ell(k) = f(k) \), we obtain

\begin{equation}
(6.63) \quad f(k) = \frac{1}{2(n-1)} (A_0 + (-1)^k A_{n-1}) + \frac{1}{n-1} \sum_{\ell=1}^{n-2} A_\ell \cos \left( \frac{\pi k \ell}{n-1} \right),
\end{equation}

where \( A_\ell \) is given by

\begin{equation}
A_\ell = f(0) + (-1)^\ell f(n-1) + 2 \sum_{k=1}^{n-2} f(k) \cos \left( \frac{\pi k \ell}{n-1} \right).
\end{equation}
The Discrete Cosine Transform (DCT) is the transformation that maps a real valued function \( f : \mathbb{Z}_n \rightarrow \mathbb{R} \) to the coefficients \( A_0, A_1, \ldots, A_{n-1} \). The inverse DCT is the mapping that takes the coefficients \( A_0, \ldots, A_{n-1} \) and reconstructs the function \( f(k) \) via (6.63). Just like with the DFT (see Section 6.3), there is a fast recursive algorithm for computing the DCT using only \( O(n \log n) \) operations. The DCT is the fundamental tool behind image and audio compression algorithms, like mp3 and jpeg. We leave the Discrete Sine Transform to an exercise.

**Exercise 6.51.** Derive the Discrete Sine Transform (DST) reconstruction formula

\[
(6.65) \quad f(k) = \frac{1}{n+1} \sum_{\ell=0}^{n-1} B_\ell \sin \left( \frac{\pi (k+1)(\ell+1)}{n+1} \right),
\]

where

\[
B_\ell = 2 \sum_{k=0}^{n-1} f(k) \sin \left( \frac{\pi (k+1)(\ell+1)}{n+1} \right),
\]

by following the arguments in this section, except for replacing the even extension \( f_e \) with the odd extension \( f_o : \mathbb{Z}_{2(n+1)} \rightarrow \mathbb{R} \) defined by

\[
(6.66) \quad f_o(k) = \begin{cases} 
0, & \text{if } k = 0 \\
 f(k - 1), & \text{if } 1 \leq k \leq n, \\
0, & \text{if } k = n + 1 \\
- f(2(n+1) - 1 - k), & \text{if } n + 2 \leq k \leq 2(n+1) - 1.
\end{cases}
\]

\[\triangle\]

### 6.8.1 DCT-based image compression

**Python Notebook: .ipynb**

We briefly explore an application of the Discrete Cosine Transform to image compression. The idea of signal or image compression with the DCT is to compute the DCT coefficients \( A_\ell \), as per (6.64), and threshold the smallest coefficients to zero. The nonzero coefficients are stored and the image or signal is decompressed by taking the inverse DCT, as per (6.63). In the case of image compression, we use the two dimensional DCT, which is computed row-wise and then column-wise, similar to the multi-dimensional DFT introduced in Section 6.7. Also, as in Section 3.4 on PCA-based image compression, we split
the image up into $8 \times 8$ blocks and perform the DCT separately on each block. This works because small blocks in images are usually very simple and can be well-approximated by only a few DCT coefficients. Figure 6.14 shows the DCT coefficients of the cameraman image on $8 \times 8$ blocks.

We show the results of the decompressed cameraman images at different compression ratios in Figure 6.15 for the PCA-based compression from Section 3.4 and the DCT-based compression. We show the plots of PSNR vs compression ratio in Figure 6.16. DCT-based compression is generally superior at low and moderate compression ratios. This is because we keep only the most significant DCT modes on each block, and these may differ from block to block, whereas in PCA-based compression we used the top $k$ PCA coefficients over the whole image, which are good for all blocks on average, but might yield poor quality reconstructions on some blocks. The PCA-based compression does work better than DCT at very high compression ratios, reflecting the fact that when we use only a very small number of basis images for the blocks, a PCA-based basis is better adapted to the particular image (in fact, it is learned from the image, while DCT is not).

### 6.9 The Sampling Theorem

In these notes we generally deal with discrete signals, and the assumption is that they have been sampled from a continuous signal at a particular sampling
Figure 6.15: Comparison of PCA-based and DCT-based image compression on the cameraman image at different compression ratios. Left is original, center is PCA, and right is DCT.
frequency. In audio processing, the continuous signal is exactly the displacement of the diaphragm on the microphone, which encodes the sound waves striking its surface. The main question we will address in this section is how fast do we need to sample a continuous signal in order to capture all the important information in the signal. We will find that for a sufficiently fast sampling rate, the signal can be reconstructed perfectly from its samples, provided it is band-limited (which means it has a maximum frequency). We will also show how to reconstruct the signal from its samples with a simple convolution operation. This theory is referred to as the Sampling Theorem. Often then name Shannon-Nyquist Sampling Theorem is used, since it was proved by Claude Shannon in a 1949 paper, and some ideas, in particular the critical sampling rate, were due to work by Harry Nyquist previously. However, many other researchers had discovered the result independently, some before Shannon did, so it has become more common to use the term “The Sampling Theorem”.

The standard version of the sampling theorem says that if a signal $u : \mathbb{R} \to \mathbb{R}$ contains no frequencies greater than $\sigma_{\text{max}}$, then $u$ can be perfectly reconstructed from its evenly spaced samples provided the sampling frequency
is greater than $2\sigma_{\text{max}}$. The critical frequency $2\sigma_{\text{max}}$ is called the Nyquist rate. Furthermore, the signal $u$ can be reconstructed from its samples via the Sinc Interpolation formula

$$\tag{6.67} u(t) = \sum_{j=-\infty}^{\infty} u(jh) \text{sinc} \left( \frac{t - jh}{h} \right),$$

where $h$ is the sampling period and $\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$ is the normalized Sinc function (defining $\text{sinc}(0) = 1$). The sampling frequency is $\frac{1}{h}$, so the Nyquist rate condition for the Sampling Theorem is that $\frac{1}{h} > 2\sigma_{\text{max}}$, or $h < \frac{1}{2\sigma_{\text{max}}}$. At sampling intervals $h > \frac{1}{2\sigma_{\text{max}}}$ aliasing occurs, where frequencies that are too high to be represented at the sampling rate are aliased to lower frequencies, creating artifacts that make reconstruction of the signal from its samples impossible.

The proof of the version of the Sampling Theorem described above requires the continuous Fourier Transform, which we do not cover in these notes. We will instead prove a similar version of the Sampling Theorem for periodic signals, which can be done with the DFT theory developed in these notes and some very basic facts about Fourier series. In particular, we assume our signal $u : \mathbb{R} \to \mathbb{R}$ is periodic with period 1, and has no frequency larger than $\sigma_{\text{max}}$, where $\sigma_{\text{max}}$ is a positive integer. This means that the signal $u$ has the Fourier Series representation

$$\tag{6.68} u(t) = \sum_{k=-\sigma_{\text{max}}}^{\sigma_{\text{max}}} c_k e^{2\pi i k t}.$$

The complex numbers $c_k$ are the Fourier Series Coefficients of $u$ and we can think of the Fourier Series as the limit as $n \to \infty$ of the DFT. Since we do not study the Fourier Series in these notes, we will take the representation (6.68) as a starting place, and investigate whether we can reconstruct $u$ from its evenly spaced samples $u(jh)$ for a sampling period $h > 0$ and $j \in \mathbb{Z}$. Naturally this boils down to determining whether we can deduce the Fourier Series Coefficients $c_k$ from the samples $u(jh)$. The answer is positive precisely when the sampling frequency exceeds the Nyquist rate. We furthermore obtain a reconstruction formula similar to Sinc Interpolation, as shown in the following result.

**Theorem 6.52.** Suppose that $u$ is given by (6.68) and let $h = 1/n$ for $n \in \mathbb{N}$ with $n > 2\sigma_{\text{max}}$. Assume also that $n$ is odd. Then $u(t)$ can be reconstructed
Figure 6.17: Depiction of the Sinc-like kernel $S(t) = \frac{\text{sinc}(t)}{\text{sinc}(ht)}$ for $n = 21$ and $h = 1/21$. The kernel is periodic with period $n = 21$.

from its evenly spaced samples $u(jh)$ and furthermore we have

$$(6.69) \quad u(t) = \sum_{j=0}^{n-1} u(jh) S\left( \frac{t - jh}{h} \right),$$

where $S(t)$ is given by

$$S(t) = \frac{\text{sinc}(t)}{\text{sinc}(ht)}.$$

**Remark 6.53.** Technically $S(t)$ is undefined when $ht$ is a nonzero integer, since $\text{sinc}(ht) = 0$. We define $S$ by continuity in this case (see Exercise 6.54). We also note that the assumption that $n$ is odd in Theorem 6.52 is only for convenience, and a similar result holds for even $n$. We show a depiction of $S(t)$ in Figure 6.17.

**Proof.** Define the sampling function $f : \mathbb{Z} \to \mathbb{C}$ by $f(j) = u(jh) = u(j/n)$. Since $u$ is 1-periodic, $f$ is $n$-periodic on $\mathbb{Z}$, so we may think of $f$ as a function in $L^2(\mathbb{Z}_n)$. By (6.68) we have

$$f(j) = \sum_{k=-\sigma_{\text{max}}}^{\sigma_{\text{max}}} c_k e^{2\pi i k j/n} = \sum_{k=-\sigma_{\text{max}}}^{\sigma_{\text{max}}} c_k \omega^{kj},$$

where $\omega = e^{2\pi i/n}$. Recalling the orthogonality of the exponential functions $u_\ell(j) = \omega^{j\ell}$ (see Lemma 6.3), a natural way to try and obtain the coefficients
$c_k$ from the sampled function $f(j)$ is to compute the inner product

$$\langle f, u_\ell \rangle = \sum_{j=0}^{n-1} f(j) \omega^{-j\ell} = \sum_{j=0}^{n-1} \sigma_{\max} \sum_{k=-\sigma_{\max}}^{\sigma_{\max}} c_k \omega^{j(k-\ell)} = \sum_{k=-\sigma_{\max}}^{\sigma_{\max}} c_k \sum_{j=0}^{n-1} \omega^{j(k-\ell)}.$$ 

As in the proof of Lemma 6.3, it is tempting to say that the right hand side is $nc\ell$. However, there are in fact contributions from all $k$ such that $k = \ell$ modulo $n$, that is $k = \ell + np$ for some integer $p$, since in that case $\omega^{j(k-\ell)} = \omega^{jnp} = 1$ since $\omega^n = 1$. This reflects the aliasing of high frequencies when the sampling rate is insufficiently fast. Hence, we have

$$\langle f, u_\ell \rangle = n \sum_{k=-\sigma_{\max}}^{\sigma_{\max}} c_k \delta_{\{k = \ell \bmod n\}},$$

where $\delta_{\{k = \ell \bmod n\}} = 1$ when $k = \ell$ modulo $n$, and zero otherwise. In order to determine the coefficient $c_k$, we need the sum above to have exactly one term, namely $c_\ell$. This is the case whenever $\sigma_{\max} < \frac{n}{2}$, or $n - 1 \geq 2\sigma_{\max}$, so that if $k = \ell + np$ and $|k| \leq \sigma_{\max}$ we must have $p = 0$. In this case we have $\langle f, u_\ell \rangle = nc\ell$, and by (6.68) and the fact that $h = 1/n$ we have

$$u(t) = \frac{1}{n} \sum_{k=-\sigma_{\max}}^{\sigma_{\max}} \langle f, u_k \rangle e^{2\pi ikt}.$$

Since $\langle f, u_k \rangle = 0$ for $\sigma_{\max} < |k| \leq \frac{n-1}{2}$, we can expand the range of $k$ in the sum to range from $k = -\frac{n-1}{2}$ to $k = \frac{n-1}{2}$ to obtain

$$u(t) = \frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} \langle f, u_k \rangle e^{2\pi ikt} = \frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} \sum_{j=0}^{n-1} f(j) e^{-2\pi ijk/n} e^{2\pi ikt} = \sum_{j=0}^{n-1} u(jh) \left( \frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} e^{2\pi i(k-jh)} \right).$$

The proof is completed by invoking Exercise 6.54, below.

\[\square\]
Exercise 6.54. Let $n$ be odd. Show that for $t \not\in \mathbb{Z}$ we have
\[
\frac{1}{n} \sum_{k=-\frac{n-1}{2}}^{\frac{n-1}{2}} e^{\frac{2\pi i kt}{n}} = \frac{\text{sinc}(nt)}{\text{sinc}(t)}.
\]

What happens when $t \in \mathbb{Z}$? \hfill $\triangle$

7 The Discrete Wavelet Transform

The Discrete Fourier Transform (DFT) basis functions $u_\ell(k) = e^{\frac{2\pi i kt}{n}}$ are completely localized in the frequency domain. That is $\mathcal{D} u_\ell = \delta_\ell$ has only one nonzero entry. This allows us to decompose a signal or image very precisely into different frequency components, which is one of the most useful properties of the DFT in practice. This comes at the expense of complete delocalization in the time (or for images, space) domain; i.e., the function $u_\ell$ is nonzero at all grid points. Figure 7.1 shows the real part of $u_\ell$ and its DFT to illustrate the dichotomy between time and frequency localization. This delocalization means we are measuring frequencies across the entire signal, and not locally in time.

In practical applications of the DFT, we address this issue of delocalization of the basis functions by breaking the signal or image up into small blocks,
and applying the Fourier transform on each block separately. This is done, for example, in Section 3.4 when using PCA for image compression, and Section 6.8.1 when using the Discrete Cosine Transform (DCT) for image compression. See, for example, Figure 6.14 for an illustration of how the DCT is applied locally to an image to sidestep the delocalization of the DCT basis.

When viewing the FFT or DCT on blocks as a transformation of the entire image, we are essentially applying a windowed version of the FFT, using a window function that is exactly equal to 1 on each block, and zero elsewhere (the window function multiplies the Fourier basis functions, as in Figure 7.2). This discontinuous window function, which is a rather crude attempt to localize the DFT basis, leads to the blocking artifacts observed in block-based image compression (see Figure 6.15). Figure 7.2 shows the DFT of a windowed version of the Fourier basis function from Figure 7.1, which shows that localizing in the time domain leads to delocalization in the frequency domain. In fact, there is a fundamental limit to how much a function and its DFT can both be localized. The uncertainty principle [5] states that

$$\|f\|_0 \|\mathcal{D}f\|_0 \geq n,$$

where $\|f\|_0$ is the number of nonzero values of $f$. This says that it is impossible for both $f$ and $\mathcal{D}f$ to both be localized (i.e., have mostly zero entries). This bound is saturated by the Fourier basis functions $u_\ell$, which satisfy $\|u_\ell\|_0 = n$ and $\|\mathcal{D}u_\ell\|_0 = 1$.

Wavelets provide a principled approach to designing a frequency transformation whose basis functions are localized, as much as is possible given the uncertainty principle, in both the time and frequency domains. In contrast with the DFT, the wavelet transform decomposes an image into its frequency components at multiple scales, allowing fine scale details to be distinguished from large ones. Like the DFT, the Wavelet transform has proven to be very useful for image denosing and compression, and some nonlinear variations on the wavelet transform (e.g., the scattering transform [2]) are currently used in deep learning for feature extraction.

This section offers a brief introduction to the Wavelet Transform. We focus mostly on Haar Wavelets, and introduce the main ideas with concrete examples, before giving the more abstract definitions. We also consider some applications to image denoising, compression, and classification.

### 7.1 The 1D Haar Wavelet

The Wavelet Transform is based on repeatedly decomposing a signal into a low frequency part, called the approximation coefficients, and a high frequency
Figure 7.2: A plot the real part of a Fourier basis function and its Discrete Fourier Transform (DFT). The function \( v (\ell = 4, n = 128) \) is completely delocalized (most values are nonzero), while its DFT is highly localized (all values vanish except one).

part, called the *detail coefficients*. This is best illustrated at first with an example. Consider the following length \( n = 8 \) signal

**Signal:** \((7,5,6,3,2,5,4,1)\)

The first level of the Haar Wavelet Transformation breaks up the signal into blocks of size 2, so \((7,5), (6,3), (2,5), \) and \((4,1)\). For each block \((a,b)\), the approximation coefficient is the sum \(a+b\), while the detail coefficient is the difference \(b-a\). Thus, a one-level Haar Wavelet Transformation yields

**Approximation Coeff:** \((12,9,7,5)\) and **Detail Coeff:** \((-2,-3,3,-3)\)

Of course, the original signal can be easily reconstructed from the approximation and detail coefficients, so it is clear the transform is invertible.

The approximation and detail coefficients are normally placed together in an array, with the approximation coefficients coming first, so the one-level Haar Wavelet transform of our signal is

**1-level Haar Wavelet Transform:** \((12,9,7,5,-2,-3,3,-3)\)

The Wavelet Transformation can be continued to further levels by applying the same procedure to the *approximation coefficients* only. The approximation coefficients \((12,9,7,5)\) are split into blocks \((12,9)\) and \((7,5)\), and the approximation coefficients are \((21,12)\) while the detail coefficients are \((-3,-2)\). The 2-level Haar Wavelet Transform is then
2-level Haar Wavelet Transform: \((21, 12, -3, -2, -2, -3, 3, -3)\)

We can now perform one final Wavelet Transformation on the remaining approximation coefficients \((21, 12)\), yielding the 3-level transform

3-level Haar Wavelet Transform: \((33, -9, -3, -2, -2, -3, 3, -3)\)

At this point, we have only a single approximation coefficient remaining and cannot perform any further iterations of the Wavelet Transform. Since each level of the Wavelet Transform is invertible, the whole process is, regardless of how many levels are used.

We can view the 3-level Haar Wavelet Transformation as multiplication of the signal \(f = (7, 5, 6, 3, 2, 5, 4, 1)\) by the matrix

\[
W = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-1 & -1 & -1 & 1 & 1 & 1 & 1 & 0 \\
-1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 \\
-1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1
\end{bmatrix}
\]

The rows can be interpreted as the basis functions for the Haar Wavelet Transformation. Notice that the rows are mutually orthogonal, so the Wavelet Transformation is an orthogonal change of coordinates, like the DFT.

In general, we can define a one-level Haar Wavelet Transform of a signal \(f \in L^2(\mathbb{Z}_n)\) of length \(n = 2^k\) as the signal \(W_1 f\) given by

\[
W_1 f(j) = \begin{cases} 
    f(2j + 1) + f(2j), & \text{if } 0 \leq j \leq \frac{n}{2} - 1 \\
    f(2j - n + 1) - f(2j - n), & \text{if } \frac{n}{2} \leq j \leq n - 1.
\end{cases}
\]

The first line in the definition of \(W_1 f\) corresponds to the approximation coefficients, while the second line is the detail coefficients. The second level of the Haar Wavelet Transformation acts only on the approximation coefficients, so we have

\[
W_2 f(j) = \begin{cases} 
    W_1 f(2j + 1) + W_1 f(2j), & \text{if } 0 \leq j \leq \frac{n}{4} - 1 \\
    W_1 f(2j - \frac{n}{2} + 1) - f(2j - \frac{n}{2}), & \text{if } \frac{n}{4} \leq j \leq \frac{n}{2} - 1 \\
    W_1 f(j), & \text{if } \frac{n}{2} \leq j \leq n - 1
\end{cases}
\]

129
Algorithm 7.1 The Haar Wavelet Transformation in Python

```python
import numpy as np

def haar_wavelet(f, depth):
    g = np.zeros_like(f)
    n2 = len(f)>>1
    g[:n2] = f[::2] + f[1::2] #Approximation coeff
    g[n2:] = f[1::2] - f[::2] #Detail coeff
    if depth >= 2:
        g[:n2] = haar_wavelet(g[:n2], depth-1)
    return g
```

In general, the $\ell$th-level Haar Wavelet Transformation is

\[
W_\ell f(j) = \begin{cases} 
W_{\ell-1} f(2j + 1) + W_{\ell-1} f(2j), & \text{if } 0 \leq j \leq \frac{n}{2^\ell} - 1 \\
W_{\ell-1} f(2j - \frac{n}{2^\ell} + 1) - f(2j - \frac{n}{2^\ell}), & \text{if } \frac{n}{2^\ell} \leq j \leq \frac{n}{2^\ell} - 1 \\
W_{\ell-1} f(j), & \text{if } \frac{n}{2^\ell} \leq j \leq n - 1
\end{cases}
\]

The Haar Wavelet Transform is simple to implement recursively in a programming language like Python. Algorithm 7.1 gives the Python code for a 1D Haar Wavelet Transform of a signal whose length is a power of 2 (for other length signals, it is common to pad with zeros or use a reflection to increase the signal length). The inverse transform is given in Algorithm 7.1. The structure of the Wavelet Transform is very similar to the FFT discussed in Section 6.3, except that only half of the remaining signal is processed at each level, instead of the entire signal. For a signal of length $n = 2^k$, the maximum depth (i.e., number of levels) of the Wavelet Transform is $k$, and at each step the length of the signal to be processed reduces by half. Thus, the number of operations for the first level is $2n$, the second level is $2^2n$, the third level is $2^3n$, and so on. After $k$ levels the number of operations is

\[
2n \left( 1 + \frac{1}{2} + \cdots + \frac{1}{2^{k-1}} \right) \leq 4n.
\]

Thus, the Wavelet Transform takes a linear number of operations $O(n)$, in contrast with the FFT which takes $O(n \log_2 n)$ operations.
import numpy as np

def inverse_haar_wavelet(f, depth):
    if depth == 0:
        return f
    else:
        n2 = len(f) >> 1
        h = inverse_haar_wavelet(f[:n2], depth-1)
        g = np.zeros_like(f)
        g[1::2] = (h + f[n2:]) / 2
        g[::2] = (h - f[n2:]) / 2
        return g

Algorithm 7.2 The Inverse Haar Wavelet Transformation in Python

Figure 7.3: One level of the Haar Wavelet Transformation applied to a noisy signal. Notice that much of the noise appears in the detail coefficients. Wavelet based denoising and compression algorithms are based on thresholding the detail coefficients.

7.2 2D Haar Wavelet Transform

The 2D Haar Wavelet Transformation acts on images in a similar way to the 1D version acting on signals. We assume we have an image of size $n \times n$, where $n = 2^k$. One level of the 2D Haar wavelet transformation breaks up the image into $2 \times 2$ blocks and computes 1 approximation coefficient and 3 detail coefficients corresponding to horizontal, vertical, and diagonal details. The formulas for the approximation and detail coefficients can be found by
applying the 1D Haar Transformation to the rows of the image, and then afterwards to the columns, in a similar way that the multi-dimensional DFT can be decomposed into 1D DFTs of rows and columns (see Section 6.7). For a $4 \times 4$ patch of the image

\[
\begin{bmatrix}
a & b \\
c & d \\
\end{bmatrix}
\]

the approximation, vertical, horizontal, and diagonal detail coefficients, denoted $A, V, H, D$, respectively, are given by

\[
\begin{align*}
A &= a + b + c + d \\
H &= -a - b + c + d \\
V &= -a + b - c + d \\
D &= a - b - c + d.
\end{align*}
\]

The inverse transform is simple to obtain; indeed, we have

\[
\begin{align*}
a &= \frac{1}{4}(A - H - V + D) \\
b &= \frac{1}{4}(A - H + V - D) \\
c &= \frac{1}{4}(A + H - V - D) \\
d &= \frac{1}{4}(A + H + V + D).
\end{align*}
\]

Thus, one level of the 2D Haar Wavelet Transformation is clearly invertible. Further levels of the transform are obtained by applying the transform again to the approximation image. Figure 7.4 shows the 2D Haar Wavelet Transformation applied to the cameraman image at levels 1 and 2, and 9, which is the largest possible depth for a $512 \times 512$ image.

### 7.3 Wavelet denoising and compression

**Python Notebook:** .ipynb

We now give some brief applications of the Haar Wavelet Transformation to image denoising and compression. In both settings, the strategy is to threshold the detail coefficients to zero. We use a basic hard thresholding here, and the results can be greatly improved with a more sophisticated form of wavelet coefficient shrinkage. Figure 7.6 shows wavelet denoising of a noisy cameraman
Figure 7.4: The Haar Wavelet Transformation of levels 1, 2, and 9 on the cameraman image. The approximation image is placed in the upper left corner, the horizontal detail in the lower left, the vertical detail in the upper right, and the diagonal detail in the lower right.

Figure 7.5: Example of using a 2-level Haar Wavelet Transformation to denoise an image. Detail coefficients with absolute value less than 0.5 were set to zero.

Figure 7.6 shows the results of Haar Wavelet based image compression at different compression ratios. Here, we used a 3-level Haar Wavelet Transformation, and we set the detail coefficients smaller than a given threshold to zero.
Figure 7.6: Example of using a 3-level Haar Wavelet Transformation to compress an image. Detail coefficients with absolute value less than 0.25 were set to zero in (a), 0.5 in (b), and 1 in (c).

We see the compressed images appear to lose resolution as the compression ratio increases. In Figure 7.7 we give a comparison of DCT-based compression (from Section 6.8.1) to wavelet-based image compression. The main difference is that DCT-based compression exhibits regular blocking artifacts on an 8 × 8 grid, while wavelet-based method show fewer blocking artifacts. Wavelet-based compression exhibits more low resolution artifacts.

7.4 Wavelet-based image classification

8 Machine Learning

8.1 Introduction and basic algorithms

8.2 Neural Networks

8.3 Convolutional Neural Networks

References

Figure 7.7: Comparison of DCT-based compression and Haar Wavelet compression on the cameraman image at a compression ratio of 32:1.


