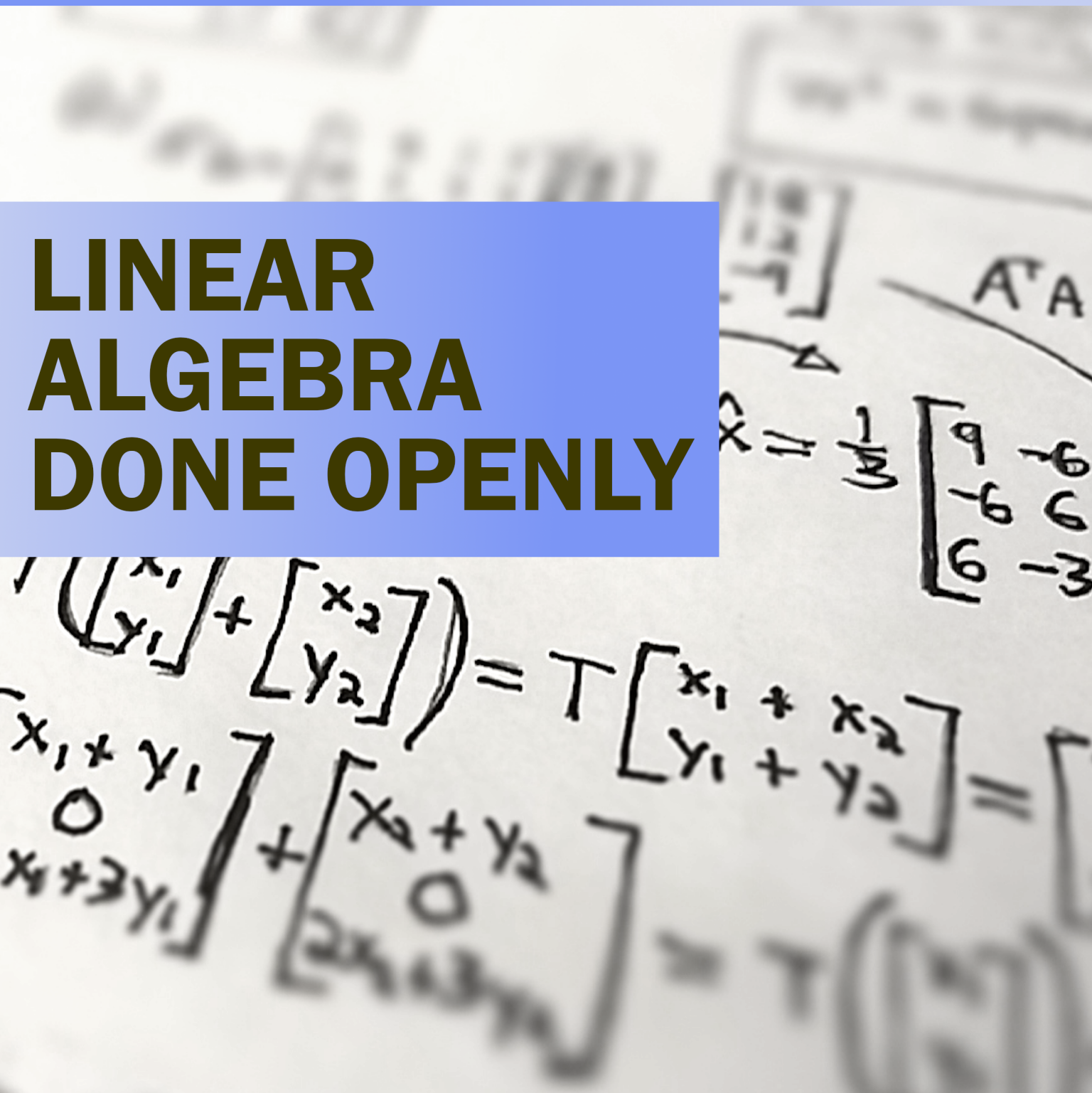
Andrew Misseldine

LINEAR ALGEBRA DONE OPENLY



Linear Algebra Done Openly: Applied Edition

Andrew Misseldine

October 12, 2026

Contents

Preface

1 The Algebra and Geometry of Vectors **3**
1.1 Vectors as Data . . . 4
1.2 Vector Operations . . . 10
1.2.1 Vector Examples . . . 10
1.2.2 Linear Combinations . . . 13
1.3 Inner Products . . . 16
1.3.1 The Dot Product . . . 16
1.3.2 Applications of Inner Products . . . 16
1.4 Norms . . . 22
1.4.1 The Norm of a Vector . . . 22
1.4.2 The Distance Function . . . 23
1.4.3 Statistical Measurements . . . 24
1.5 Clustering . . . 29
1.5.1 The Nearest Neighbor Problem . . . 29
1.5.2 The Clustering Problem . . . 29
1.5.3 Applications of Clustering . . . 31
1.6 The k-means Algorithm . . . 35

2 Linear Systems **45**
2.1 Linear Systems . . . 46
2.1.1 Linear Systems . . . 46
2.1.2 Number of Solutions to a Linear System . . . 47
2.1.3 3-Dimensional and Homogeneous Linear Systems . . . 48
2.2 Augmented Matrices . . . 53
2.2.1 The Elementary Row Operations . . . 53
2.2.2 Augmented Matrices . . . 54
2.2.3 Echelon Forms . . . 55
2.3 Reduction of Linear Systems . . . 64
2.3.1 Gaussian Elimination . . . 64
2.3.2 Gauss-Jordan Elimination . . . 66
2.4 Applications of Linear Systems . . . 72
2.4.1 Data Fitting . . . 72
2.4.2 Conservation Laws . . . 74
2.5 Vector Equations . . . 79
2.5.1 Equations and Linear Combinations . . . 79
2.5.2 The Span of Vectors . . . 82
2.6 Matrix Equations . . . 86
2.6.1 Matrix-Vector Multiplication . . . 86
2.6.2 Column Space . . . 88
2.6.3 Matrix Transformations . . . 89
2.7 Linear Independence . . . 93
2.7.1 Linear Independence of Vectors . . . 93
2.7.2 Dimension . . . 95

2.7.3 Basis for Column Space . . . 95
2.8 Solution Sets of Linear Systems . . . 99
2.8.1 The Null Space . . . 99
2.8.2 Basis for Null Space . . . 100
2.8.3 Particular Solutions to Linear Systems . . . 101
2.9 Orthogonality . . . 103
2.9.1 Orthogonal Vectors . . . 103
2.9.2 Orthogonal Sets . . . 105
2.9.3 Orthogonal Projections . . . 105

3 The Algebra and Geometry of Matrices 113
3.1 Matrix Operations . . . 114
3.1.1 Linear Combinations of Matrices . . . 114
3.1.2 Matrix Multiplication . . . 115
3.1.3 The Transpose of a Matrix . . . 117
3.1.4 The Trace of a Matrix . . . 118
3.2 Matrix Applications . . . 121
3.2.1 Graphs and Adjacency Matrices . . . 121
3.2.2 Selectors . . . 122
3.2.3 Vector Convolution . . . 123
3.3 Matrix Inverses . . . 128
3.3.1 Nonsingular Matrices . . . 128
3.3.2 The Inversion Algorithm . . . 129
3.3.3 Left Inverses . . . 129
3.4 Linear Transformations . . . 134
3.4.1 Linear Transformations . . . 134
3.4.2 The Standard Matrix of a Linear Transformation . . . 135
3.4.3 Composition of Linear Transformations . . . 136
3.5 Linear Transformations on $\mathbb{R}^2$ . . . 138
3.5.1 Stretches and Reflections . . . 138
3.5.2 Rotations . . . 141
3.5.3 Affine Transformations . . . 142
3.6 The Gram-Schmidt Algorithm . . . 147
3.6.1 The Gram-Schmidt Algorithm . . . 147
3.6.2 Isometries . . . 148
3.6.3 The QR Factorization . . . 150
3.7 Linear Dynamical Systems . . . 154
3.7.1 Dynamical Systems . . . 154
3.7.2 Linear Dynamical Systems . . . 154
3.7.3 Control . . . 156
3.8 Applications of Linear Dynamical Systems . . . 159
3.8.1 Population Growth . . . 159
3.8.2 Infectious Spread . . . 161
3.8.3 Directed Graphs and Logisitics . . . 162

4 Least Squares 169
4.1 The Least Squares Problem . . . 170
4.1.1 Normal Equations . . . 170
4.1.2 Applications of Least-Squares . . . 172
4.2 Data Fitting . . . 176
4.3 Regression . . . 182
4.3.1 Affine Regression . . . 182
4.3.2 Auto-regressive Time Series . . . 184
4.3.3 Periodic Regression . . . 185
4.4 Validation . . . 190
4.4.1 Out-of-Sample Validation . . . 190
4.4.2 Cross-Validation . . . 193

4.5 Feature Engineering . . . 197
4.5.1 Linear Features . . . 197
4.5.2 Boolean Features . . . 199
4.5.3 Nonlinear Features . . . 200
4.6 Classification . . . 204
4.6.1 Classifiers . . . 204
4.6.2 Least Squares Classifiers . . . 205
4.6.3 Multi-class Classifiers . . . 206
4.7 Multi-Objective Least Squares . . . 209
4.7.1 Multi-Objectives . . . 209
4.7.2 Applications of Multi-Objective Least Squares . . . 210
4.8 Regularization . . . 216
4.9 Constrained Least Squares Problem . . . 219
4.10 Control of Linear Dynamical Systems . . . 226
4.10.1 Linear Quadratic Control . . . 226
4.10.2 State Feedback Control . . . 229

5 Eigenvalues **237**
5.1 Eigenvalues and Eigenvectors . . . 238
5.1.1 Eigenvalues . . . 238
5.1.2 Eigenvectors . . . 239
5.1.3 Eigenvalues of Triangular Matrices . . . 240
5.1.4 The Characteristic Polynomial . . . 241
5.2 Diagonalization . . . 245
5.2.1 Similar Matrices . . . 245
5.2.2 Diagonalizable Matrices . . . 245
5.2.3 Eigenbases . . . 246
5.2.4 Stable Solutions . . . 248
5.3 Orthogonal Diagonalization . . . 257
5.3.1 Orthogonally Diagonalizable Matrices . . . 257
5.3.2 The Spectral Theorem . . . 258
5.3.3 Spectral Decomposition . . . 259
5.4 Singular Value Decomposition . . . 262
5.4.1 Singular Values . . . 262
5.5 Dimension Reduction . . . 267
5.5.1 Singular Reduction . . . 267
5.5.2 Principal Component Analysis . . . 270

Appendix A Python Primer **277**
A.1 Variables . . . 277
A.2 Loops and Conditionals . . . 280
A.3 Functions . . . 283

Appendix B The Substitution and Elimination Methods **285**
B.1 The Substitution Method . . . 285
B.2 The Elimination Method . . . 286

Appendix C Further Topics on Orthogonality **289**
C.1 Angles Between Vectors . . . 289
C.1 Inner Product Spaces . . . 289
C.2 Idempotent Matrices . . . 290
C.2 Orthogonal Decomposition . . . 292
C.1 Orthogonal Complements . . . 292
C.2 Orthogonal Decompositions . . . 293
C.3 Best Approximation . . . 293

Appendix D Error-Detecting and Correcting Codes **297**
D.1 Binary Symmetric Channels 297
D.2 The Hamming Norm 299
D.3 Linear Codes 300
D.4 Parity-Check and Generator Matrices 301
D.5 Decoding 303

Appendix E Solutions to Select Exercises **307**

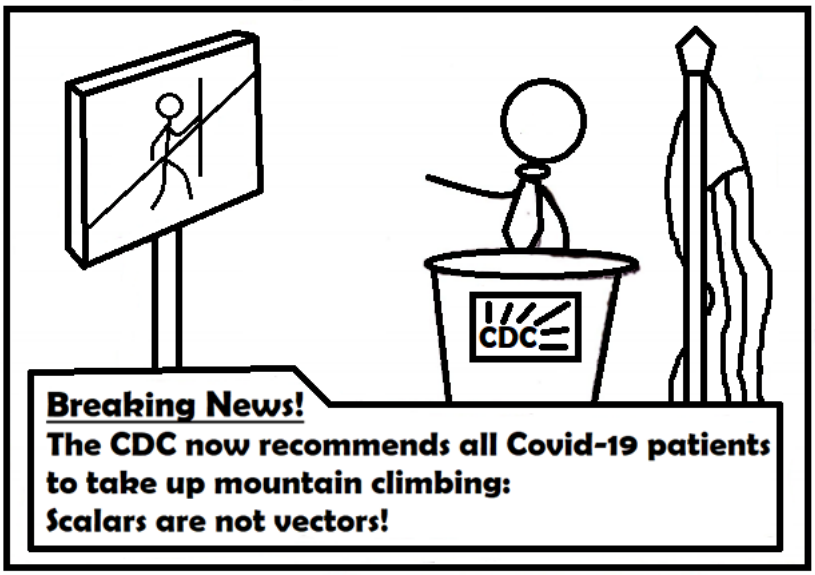
Preface

List of Contributors

Kory Adams
Abby Allen
Will Allen
Kaden Allred
Blaise Amisi
Samuel Andersen
Rodrigo Armaza
Luke Armstrong
Jordan Ashe
Caroline Ashton
Alicyn Astle
Sherie Ayag
Dillon Bales
Shelby Bartlett
Tyler Bayn
Landry Benimana
Andrew Biskey
Carson Blickenstaff
Karson Blomquist
Alexis Borell
Sage Bosgieter
Spencer Burton
Nicholas Bybee
Adam Carling
Braden Carlson
Owen Chadwick
Hailey Checketts
Skyler Clark
Mariah Clayson
Kacie Dastrup
Jacob Davenport
David de Lera Santo
Charles Del Quandro
Manoj Singh Dhami
Cameron Dix
Addyelle Dizon
Kaylee Dockter
Jakob Dowdle
Zayd Dridi
Riley Drishinski
Joshua Edgel
Kaden Empey
Bostyn Farr
Candance Fehr
Courtney Flanigan
Jaren Frandsen
Lara Garíca Pájaro
C.J. Giacoletto
Phillip Goins
Jaimie Goldberg
Caleb Goodrich
Robert Gordon
Jun Hanvey
Jonah Hoff
Jordan Griffith
Kaylee Hall
Kirsten Hall
Malcolm Hanks
Elisabeth Hansen
Thayne Hansen
Josh Hardman
Hadlee Haroldsen
Yingjie He
Bridger Hildreth
Mitchell Holdstock
Christopher Houston
Ethan Hoyer
Im Dohyeon
David Jackson
Jacob Jensen
Kerri Jensen
Rebekah Jensen
Cadence Johnson
Sofia Jones
Benita Kakama
Cyrus Kaveh
Efe Kaya
Jacob Kuhn
Maria Langford
Tyler Lauber
Louaivasa Laulu
Laura Lee
Lee Woojin
Yucheng Long
Jaxton Maez
Dareen Mahmoud
Blake Mangola
JJ Marsigli
Joshua Mathew
Adam Maxwell
Andrew Maxwell
Ryan McMurray
Raina Mickelsen
Kellen Nankervis
Hunter Nelson
Ian Mc Farlane
Ellie McReaken
Kalvin Mudrow
Mun Jungyong
Dallin Nelson
Hunter Nelson
Jacob Newey
Christopher Newton
Anthony Nguyen
Yinglong Niu
Gordon Ochsner
Ethan Olcott
Mateo Osorio Delhonte
Benjamin Palmer
Brittany Palmer
Isaac Palmer
Seth Palmer
Darby Parise
Bret Pendleton
Colin Reid
Chase Riggs
Zach Rogers
Hyrum Rohrbach
Rindy Roos
Hamza Samha
Henry Sanderson
Lucas Shaner
Hannah Simonson
Joseph Spendlove
Bram Stults
Noah Swenson
Tommaso Taddei
Ashley Taylor
Toby Tebbs
CJ Torgerson
Jaden Torgerson
Daven Triplett
Runtian Tu
Gage VanDyke
Allyson Vest
Grayson Walker
Gregory Walsh
Sarah Walters
Adym Warhurst
Leon Weingartner
Kordell Welch
Shawn Werber
Max Whipple
Sarah Wilcox
Andrew Wilks
Walt Williams
Chance Witt
Kyle Wood
Kennedy Worthington
Jiazheng Yan
Shuting Yang
Jianhe Yu
Mattie Zeigler
Yanjiao Liu
Makenley Zimmerman
Yifan Zhu
Heming Zu
Mitchell Zufelt

Chapter 1

The Algebra and Geometry of Vectors



Vectors are the fundamental building blocks of Linear Algebra. Vector spaces, matrices, linear systems, linear transformations, etc. are all made up from vectors. In the most naïve sense, vectors are lists of quantities, an array of numbers. The most common representation of vectors that we have seen before or will see going forward are just that, an array of numbers. This is exactly what a column vector or row vector is. The debate is around what a vector is, just how we should write them: vertically or horizontally. This perspective is not completely without merit since all vector spaces are isomorphic to a space of column vectors.

But if that is all a vector is, then Linear Algebra would not be the rich discipline of Mathematics that it is. Note that a point in a geometry, such as the Euclidean geometry $\mathbb{R}^2$ or $\mathbb{R}^3$, is characterized by its coordinates, and are not the coordinates, for example $(1, 2) \in \mathbb{R}^2$ or $(3, -1, 2) \in \mathbb{R}^3$, just a list of numbers? Coordinate geometry endows quantitative arrays with geometry meaning. The student of Linear Algebra would be given a disservice if their learning of vectors was divorced from this Analytic Geometry. And they will not be, as the geometric connections of vectors will be discussed in this chapter and beyond. We must come to see that vectors are not just arrays of numbers. They are geometric arrays of numbers.

More importantly, as the name "Linear Algebra" suggests (the book is not named "Linear *Geometry* Done Openly") is that vectors are algebraic arrays of numbers. Linear Algebra demands that we do algebra on vectors. What makes a vector a vector is not numbers in a list, but in the fact that we can add vectors together and we can scale vectors in a way that matches the geometric interpretation discussed above. Is it not a marvel that we can meaningfully discuss the sum of two arrows. How could an arrow plus an arrow possibly mean anything useful, but it does. It is another arrow! Is it meaningful to add together two trees, two bicycles, two backpacks, or two billboards? Of course not, as those are not vectors.

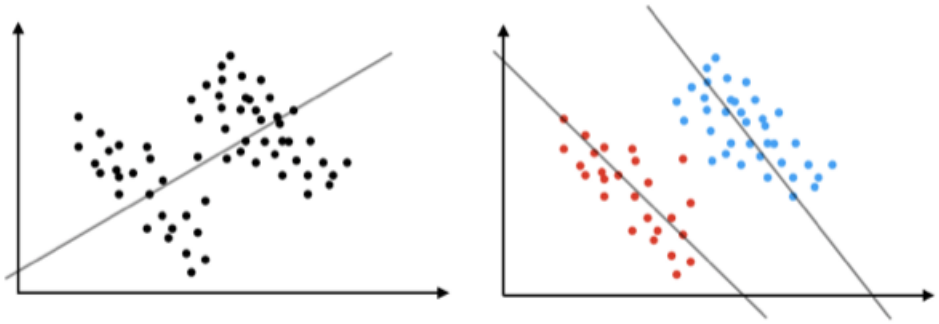
"Mathematics is the art of reducing any problem to linear algebra." – William Stein

1.1 Vectors as Data

In this section, we introduce the vectors as an array of numbers, but this simple idea has numerous applicability. We introduce basics of vectors, namely vocabulary and notation, but with an emphasis on how vectors are natural objects in the computer and data sciences.

Example 1.1.1. In August 2021, in the midst of the COVID-19 worldwide pandemic, data[i] out of Israel stated that people who had been vaccinated against COVID-19 were more likely to be hospitalized for this disease than those who were not vaccinated. Strange, right? Wouldn't you expect that the vaccinated are more protected? Does this mean that the vaccine was ineffective or even dangerous? That is how some people interpreted this data and used it to support vaccine hesitancy. A closer look at the data actually tells a different story.

Severe hospitalization was strongly correlated with age, e.g. those over 50 were 20 times more likely to be hospitalized than those under 50 and those over 90 years were 1600 times more likely to be hospitalized than a teenager. As such, vaccination rates were also strongly correlated with age, given that the elderly population knew they were more at risk and were more likely to need a vaccine. In Israel, nearly 80% of the population was vaccinated and over 90% of the population over 50 years was vaccinated. While the vaccine was perhaps the most effective shield from COVID-19, it was not perfect as vaccinated individuals had been able to contract the virus, at much lower rates that we will see shortly. When you stratify the data by age (the confounding variable for both vaccination rates and hospitalizations), we see that all age groups have a negative correlation between vaccination and hospitalization (from which we can correctly infer that the COVID-19 vaccine was effective at combating the virus). This is in contrast to the original unstratified data which suggested a positive correlation between vaccination and hospitalization (from which one might incorrectly infer that the COVID-19 vaccine was ineffective at combating the virus). This is an example of *Simpson's Paradox*.[ii]



■

The previous example introduces some interesting topics from data science for which linear algebra can prove useful. Let us consider two such questions. How do we know the variables are correlated? In this example, as there are two variables in play, we can draw the *line of best fit* (or the so-called *regression line*). The slope of this line quantifies the correlation of the variables, but the distances the data points are off the line also states how reliable this regression is. In the first graphic above, we see that the two variables (black) are positively correlated but the points do not fit on the line very well. In the second graphic, we see that both the red data and the blue fit much better on their respective regressions but now the correlations are negative. This leads to the second question: how did we know to separate the data into two subsets? This is known as *clustering*, and the problem is essentially the best way of drawing a line between the two clusters. After the clustering, the question about correlation has a very different answer. As this problem only consists of only two variables, it is easy to graph the data and intuitively inspect the data to answer these questions. Clearly, this becomes a much more difficult problem when the data sets are larger and the number of variables involved increases. In this book we will see how linear algebra can be used to solve these

[i] https://www.covid-datascience.com/post/israeli-data-how-can-efficacy-vs-severe-disease-be-strong-when-60-of-hospitalized-are-vaccinated

[ii] https://towardsdatascience.com/simpsons-paradox-and-interpreting-data-6a0443516765

two problems among others. Linear algebra is essentially the study of vectors.

Definition 1.1.2. A **vector** is a finite, ordered list of numbers. Commonly, a vector is denoted as a vertical array of said numbers delimited by brackets or parentheses, e.g.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad \text{or} \quad \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}.$$

These vertical arrays can be very inconvenient for typesetting purposes (look at all that wasted white space), and so vectors are commonly written horizontally too, e.g.

$$[1,\ 2,\ 3,\ 4] \qquad \text{or} \qquad (1,\ 2,\ 3,\ 4).$$

The numbers in a vector are called its **entries** (or **coordinates** or **components**). In the horizontal case, one uses commas to separate entries in a vector, but these are omitted in the vertical case. The number of entries in a vector is called its **dimension** (or **length** or **shape**). Hence, a vector with n entries is called an **n-dimensional vector** or simply an **n-vector**.

A **scalar** is a single real number. Hence, a vector is an array of scalars. Generally, vectors will be denoted with boldface font, e.g. $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$, and the scalars will have no such bolding. The entries in a vector, which are scalars, will typically be denoted by the same alphabetic character without the boldface font and a subscript denoting the entry's index, e.g. u_1, v_2, w_4.[iii]

The set of real numbers is denoted $\mathbb{R}$ and the set of all n-vectors is denoted $\mathbb{R}^n$.

Definition 1.1.3. Let $\boldsymbol{e}_i$ denote the n-vector[iv] all of whose entries are 0 except the ith entry which is 1. Let $\mathbf{0}$ denote the **zero vector**, which is the n-vector which consists of only zeros. Let $\mathbf{1}$ denote the **ones vectors**, which is the n-vector which consists of only ones. For example, in $\mathbb{R}^3$, we have

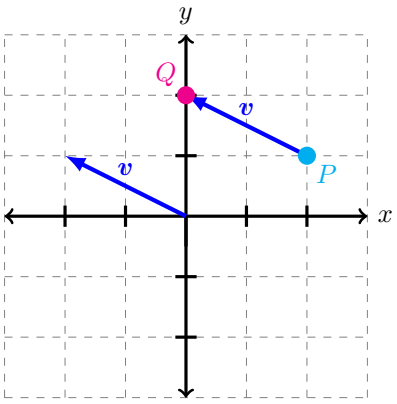
$$\boldsymbol{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{1} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Example 1.1.4. Vectors are commonly used to address position in space with respect to some coordinate system. Imagine (x, y)-coordinates in the plane (aka $\mathbb{R}^2$) or (x, y, z)-coordinates in 3-space (aka $\mathbb{R}^3$). Similar coordinate systems also apply to higher dimensional settings likes $\mathbb{R}^4$ or $\mathbb{R}^5$.

Vectors are similarly used to represent change of position. For example, consider the points illustrated below. The displacement from the point $P = (2, 1)$ to the point $Q = (0, 2)$. This displacement can be encoded by the vector $\boldsymbol{v} = (0 - 2, 2 - 1) = (-2, 1)$. This type of displacement vector is typically denoted by an arrow and this is the most common usage of vectors in geometry and physical science. In addition to displacement, they are used for velocity, acceleration, and force, among other important vector quantities.

[iii] As boldface font is difficult to hand-write, sometimes an arrow is drawn over a vector quantity instead, e.g. $\vec{u}$, $\vec{v}$, $\vec{w}$. This is not universal notation though. Different authors may use other notation and some do not use any notation to separate vector and scalar quantities. For beginners this lack of distinction can be somewhat awkward.

[iv] We sometimes use the same notation, e.g. $\boldsymbol{e}_1, \mathbf{0}, \mathbf{1}$, to denote different vectors. For example, in $\mathbb{R}^2$ we have $\boldsymbol{e}_1 = (1, 0)$, but in $\mathbb{R}^3$ we have $\boldsymbol{e}_1 = (1, 0, 0)$. In computer science, it is called *overloading*. It is potentially confusing or ambiguous but context should always make it clear which exact vectors we are referring to.



The location of an arrow in the plane does not change the amount it displaces. As such, vectors are often drawn in *standard position* meaning that their tail is placed on the origin of the geometry. Then its head points to a unique point in the plane. Identifying a displacement arrow with this geometric point shows that these two seemingly different usages of vectors are actually the same, that is, these two representations of vectors are *isomorphic*. It can be shown that all representations of vectors are isomorphic, which suggests how versatile and useful they are.

■

Example 1.1.5. A **portfolio** (or **profile**) is a list of quantities that describe an individual, client, vendor, etc. that are potentially unique yet essential for that individual. This could be a medical record, such as age, weight, height, blood pressure, etc. This could be the number of shares owned across several corporations. Note that a coordinate vector, as in Example 1.1.4, is a profile of a point in the geometry. We list a few examples of portfolio vectors to illustrates their usefulness.

A *bill of materials* is a vector that gives the amounts of n resources required to create a product or carry out a task, like the ingredients of a recipe.

Another example includes listing the number of times a critical category occurs, called the *frequency*, or the percentage of the whole that occurs, called the *relative frequency*, such as the distributions of grades in a linear algebra course or the number of occurrences of keywords in a document. A frequency vector is commonly illustrated as a histogram or bar graph, and a relative frequency vector is commonly illustrated as a pie chart.

A *pixel* is a 3-vector which is used to represent color in computer graphics. The three entries of the vector are real numbers on the interval $[0, 1]$ that encode what percentage of red, green, and blue (RGB) to blend together. Note that $\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3, \mathbf{0}, \mathbf{1}$ denotes the colors red, green, blue, black, and white, respectively.

Entries of a portfolio could be zero or negative, where positive versus negative could represent direction. For example, a *cash flow* vector could represent a list of accounts among clients and vendors. A positive entry means the client owes the firm, while a negative entry means the firm owes a vendor.

Entries might be restricted to integers or nonnegative real numbers. They could be restricted to a value between 0 and 1 inclusive, such as a percentage or relative frequency. They could be restricted to just 0 or 1, where 1 represents the presence of a feature and 0 represents its absence. Such a variable type is called a **boolean**. Sometimes 0 and 1 are replaced with false and true, respectively, although the usage of 0 and 1 foreshadows possible arithmetic usage. ■

Definition 1.1.6. Given two vectors $\boldsymbol{x}, \boldsymbol{y}$, we say that $\boldsymbol{x}$ is a **slice** (or **subvector**) of $\boldsymbol{y}$, if each entry x_i in $\boldsymbol{x}$ is contained in $\boldsymbol{y}$ in the same order as they appear in $\boldsymbol{x}$ (although gaps between x_i and x_{i+1} in $\boldsymbol{y}$ is allowed). The subvector $\boldsymbol{x}_{r:s}$ of $\boldsymbol{x}$ is defined as

$$\boldsymbol{x}_{r:s} = (x_r, x_{r+1}, \ldots, r_s),$$

for acceptable **index range** $[r : s]$.

Example 1.1.7. A **time series** (or **signal**) is an n-vector $\boldsymbol{t}$ where t_i (called a **sample**) denotes the value of a specific quantity at timestamp i. This vector could record the hourly temperature (rainfall, barometric pressure, etc.) at a specific geographic location, such as a city. An audio signal can be represented as a

vector whose entries are the acoustic pressure on short equally spaced time interval (typically 48000 or 44100 per second). It could be the dollar value of a stock every ten minutes (every hour, day, minute-by-minute, etc.). Such vectors are often visualized as *line graphs*.

Alternatively, the vector could be the difference between the current value and the previous sample in the series. This interpretation, like the example of stock value, could be thought of as the discrete *derivative* of these time series as it measures how the time series is changing over time. ∎

In addition to scalars, we allow the entries of a vector to be vectors themselves, e.g. $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$, and is called a **block vector**. In this case, we understand the array of vectors to be the *concatenation* of all vectors in the list in the order they are provided. The following definition provides an important example of block vectors.

$$\boldsymbol{x} = \begin{bmatrix} (1,2) \\ (3,4,5) \\ 6 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}.$$

Definition 1.1.8. An m-vector consisting of only n-vectors is called an $m \times n$-**matrix**. Instead of representing it as a long vector, matrices are drawn as rectangular arrays. As mentioned above, an array of vectors is itself a vector. Hence, an $m \times n$-matrix can be viewed as an mn-vector.

$$\begin{bmatrix} (1,2,3) \\ (4,5,6) \\ (7,8,9) \\ (10,11,12) \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Example 1.1.9. A digital photo is an $m \times n$-matrix of *pixels*. As each pixel is a 3-vector, a photo can be represented as a single $3mn$-vector. A digital movie is an array of photos and a vector of sounds (an audio time series). Concatenating all of these vectors together forms a single vector, which is quite large, that represents the movie. This example in addition to the previously mentioned examples (as well as several omitted examples) show how useful vectors can be in computer and data science. ∎

Python Programming

In Python, use the `NumPy` package to implement vectors.

```
import numpy as np
```

Python has two common ways to store collections of numbers: lists and `NumPy` arrays. Lists are built-in, flexible, but slower for numerical work. A list is written between two brackets and elements are separated by commas. `NumPy` arrays instead are fast, efficient, and designed for numerical computation. They are declared using `np.array(<a list>)`. We will encode vectors as 1-dimensional `NumPy` arrays.

To retrieve the ith entry (from the beginning) from a vector $\boldsymbol{x}$, write `x[i]`. To retrieve the ith entry from the end of $\boldsymbol{x}$, write `x[-i]`. Hence, `x[-1]` is always the last entry of the vector regardless of length. Be warned, in Python an n-vector's index starts at 0 and end at $n-1$. Note that `x[i:j]` will call the subvector of $\boldsymbol{x}$ containing entries x_i up to but not including x_j, that is, the jth entry is omitted but all entries up to j are included. If either i or j are omitted, it is understood to mean that $i = 0$ or $j = -1$, respectively. Lastly, if S is a subset of indices within the range of $\boldsymbol{x}$, then `x[S]` will call the slice of $\boldsymbol{x}$ associated to S.

Block vectors can be created using the `concatenate` command. The all zeros or all ones n-vector is formed by `np.zeros(n)` and `np.ones(n)`, respectively, where entries will be casted as floating points.

Example 1.1.10. The code:

```
import numpy as np


x = np.array([1,2,3,4])    # create a np.array, aka a vector
print(x)
print(x[0],x[3])           # retrieve specific entries from a vector
print(x[1:2])              # retrieve a slice of the vector
print(x[1:])               # omitting the last bound implies slice all the way to the end
subset = [0,2,3]           # create a list
print(x[subset])           # retrieve the slice associated to a subset of indices

x[0] = -5                  # individual entries of the vector can be redefined
x[1] = 6
print(x)

x[-1] = 0                  # negative indices retrieve from the end of the vector
x[-4] = 10
print(x,"\n")

x = np.array([1,2])
y = np.array([3,4,5])
 # puts together any number of np.arrays of the same shape, in the order they appear
z = np.concatenate((x,y,[6,8,9,10]))
print(z)
 # create an all-zeros or all-ones np.array, where the input is the shape of the np.array
print(np.zeros(5),np.ones(3))
```

produces the output:

```
[1 2 3 4]
1 4
[2]
[2 3 4]
[1 3 4]
[-5 6 3 4]
[10 6 3 0]

[ 1 2 3 4 5 6 8 9 10]
[0.  0.  0.  0.  0.]  [1.  1.  1.]
```

■

Exercises

(Go to Solutions)

For Exercises 1-3, determine the length of the vector described.

1. The list of favorite colors of each student in a class of 25, using RGB values.
2. The list of daily rental fees, house service fees, cleaning fees, and other miscellaneous fees (four values, not combined) charged to each room in the hotel, which has 7 floors and 15 rooms per floor.
3. The list of calories, fat, carbs, and protein for each meal: breakfast, lunch, dinner, and other snacks, throughout a given day by an individual on a diet.

We say that a vector is **sparse** if it only has a few nonzero entries. For Exercises 4-11, briefly describe what a sparse vector means in the given context.

4. $\boldsymbol{x}$ is a medical profile for a patient containing n medical quantities, say age, weight, height, blood pressure, etc.

♠ 5. $\boldsymbol{x}$ is a bill of materials for a project containing n materials needed.

♠ 6. $\boldsymbol{x}$ is a vector recording the frequencies of n categories observed in raw data, for example, keywords in a document.

♠ 7. $\boldsymbol{x}$ is the daily cash flow of a business over n days.

♠ 8. $\boldsymbol{x}$ is a Boolean vector which records the presence of n features.

♠ 9. $\boldsymbol{x}$ is the daily high temperature, in $F°$, at a certain location over n days.

♠ 10. $\boldsymbol{x}$ is an audio signal with n samples.

11. $\boldsymbol{x}$ is a digital photo with n pixels.

12. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/data.ipynb

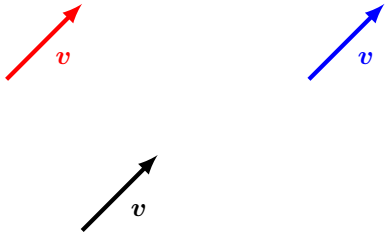
"When bad men combine, the good must associate; else they will fall one by one, an unpitied sacrifice in a contemptible struggle." – Edmund Burke

1.2 Vector Operations

In this section, we learn about vector and vector spaces. This revolves around algebraic operations of vectors, namely vector addition and scalar multiplication. We discuss properties of these vector operations.

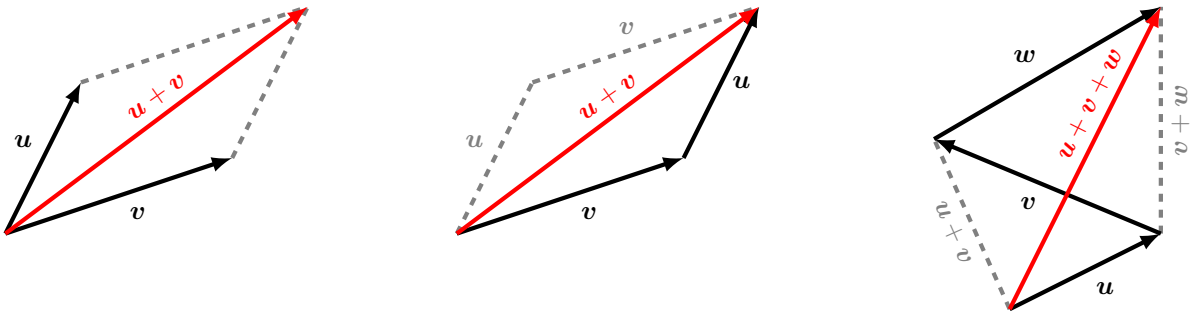
1.2.1 Vector Examples

Example 1.2.1. In physics, a vector represents a mathematical quantity with both magnitude and direction, such as a force applied to an object. Vectors are thus represented as arrows pointing in the given direction and whose length represents the magnitude of the vector. Physical vectors are also free to move about in space, that is, this exact location does not matter so long as their length nor direction does not change. Therefore, the three arrows illustrated above



represent that same vector $\boldsymbol{v}$. Hence, the arrow is determined by the relative displacement of the head of the arrow from its tails. These vectors can be added together by the so-called *parallelogram rule*: begin by placing the tails of the two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ together, form a parallelogram whose parallel sides correspond to $\boldsymbol{u}$ with a copy of $\boldsymbol{u}$ and $\boldsymbol{v}$ with a copy of $\boldsymbol{v}$; the sum $\boldsymbol{u}+\boldsymbol{v}$ is the unique vector which is the diagonal of the parallelogram whose tail agrees with the tail of $\boldsymbol{u}$ and $\boldsymbol{v}$, as displayed below left.

The second diagram shows that arrow addition is commutative, as the two paths from the tail of $\boldsymbol{u}+\boldsymbol{v}$ to head produce the same vector. In particular, the black, solid path comprises $\boldsymbol{v}+\boldsymbol{u}$ while the gray, dashed path comprises $\boldsymbol{u}+\boldsymbol{v}$, and both paths have the same starting and ending points. The third diagram likewise demonstrates how three or more vectors are added together and why arrow addition is associative, as the path following $(\boldsymbol{u}+\boldsymbol{v})$ then $\boldsymbol{w}$ produces the same vector as the path following $\boldsymbol{u}$ then $(\boldsymbol{v}+\boldsymbol{w})$.



The Parallelogram Rule $\boldsymbol{u}+\boldsymbol{v}=\boldsymbol{v}+\boldsymbol{u}$ $(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w}=\boldsymbol{u}+(\boldsymbol{v}+\boldsymbol{w})$

The zero vector $\mathbf{0}$ would be simply a point in space, that is, the vector with no magnitude nor direction. Notice that adjoining a point to the head or tail will neither lengthen the arrow nor turn it. Thus, $\boldsymbol{v}+\mathbf{0}=\boldsymbol{v}$. Additionally, given a vector $\boldsymbol{v}$, $-\boldsymbol{v}$ is defined as the arrow with the same length but pointing in the opposite direction. It holds that $\boldsymbol{v}+(-\boldsymbol{v})=(-\boldsymbol{v})+\boldsymbol{v}=\mathbf{0}$.



We define scalar multiplication of arrows by multiplying the length of the arrow by the scalar (and switching directions if the scalar is negative). Clearly, $1\boldsymbol{v}=\boldsymbol{v}$, since the length of the arrow is unchanged, and $c(d\boldsymbol{u})=(cd)\boldsymbol{u}$, since stretching a vector by a factor of d then stretching by a factor of c as the net affect of

stretching the arrow by a factor of cd. It is left as an exercise of the reader to prove that two distributive laws. Therefore, the set of arrow in space forms a vector space under these operations. ∎

As illustrated in the previous example, physical vectors, aka arrows, such as force or velocity, can be added and scaled and these two operations have physical interpretations. We also saw in Example 1.1.4 that physical vectors put in standard position correspond with points in space, whose coordinates form an array of numbers, that is, arrows can be represented as arrays. If we can add and scale the arrows, ought we not to also be able to add and scale their array counterparts? Of course, we can. Vectors are more than just arrays of numbers. All vectors are infused with algebraic operations, namely: **vector addition** and **scalar multiplication**. We will see that these algebraic operations of combining vectors as arrays (below) is dramatically easier than geometric operations of combining vectors as arrows (above).

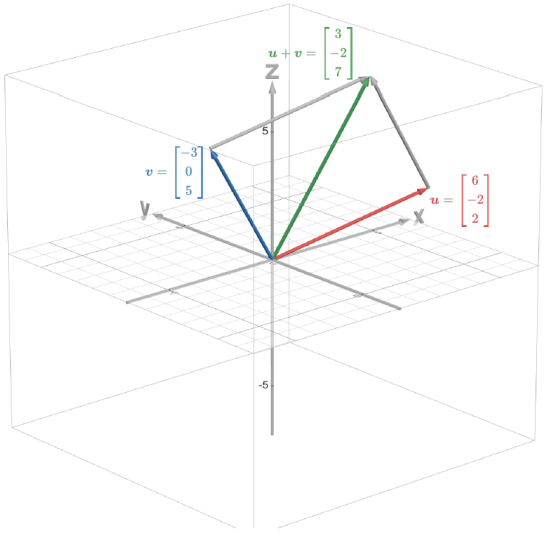
Addition of column vectors is component-wise, that is, scalars in the corresponding positions are added together:

$$\boldsymbol{u}+\boldsymbol{v}=\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}+\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}=\begin{bmatrix} u_1+v_1 \\ u_2+v_2 \\ \vdots \\ u_n+v_n \end{bmatrix}. \tag{1.2.1}$$

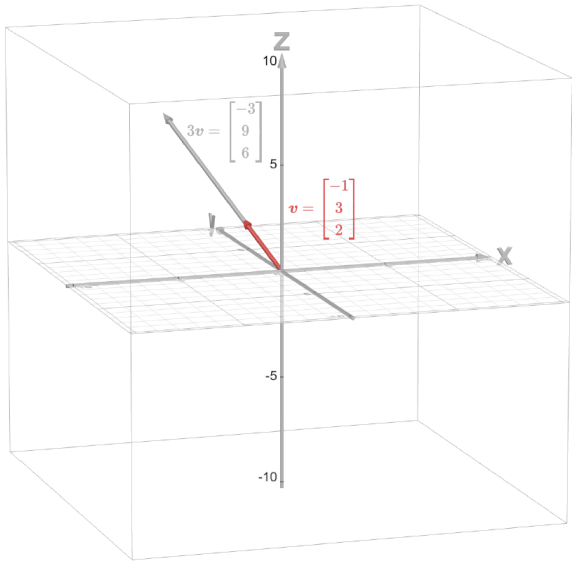
For scalar multiplication, every component is multiplied by the scalar, that is:

$$c\boldsymbol{x}=c\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}=\begin{bmatrix} cx_1 \\ cx_2 \\ \vdots \\ cx_n \end{bmatrix}. \tag{1.2.2}$$

Example 1.2.2. Let $\boldsymbol{u}=\begin{bmatrix} 6 \\ -2 \\ 2 \end{bmatrix}, \boldsymbol{v}=\begin{bmatrix} -3 \\ 0 \\ 5 \end{bmatrix} \in \mathbb{R}^3$. Then



$$\boldsymbol{u}+\boldsymbol{v}=\begin{bmatrix} 6 \\ -2 \\ 2 \end{bmatrix}+\begin{bmatrix} -3 \\ 0 \\ 5 \end{bmatrix}=\begin{bmatrix} 6-3 \\ -2+0 \\ 2+5 \end{bmatrix}=\begin{bmatrix} 3 \\ -2 \\ 7 \end{bmatrix}.$$



If $\boldsymbol{v} = \begin{bmatrix} -1 \\ 3 \\ 2 \end{bmatrix} \in \mathbb{R}^3$, then

$$3\boldsymbol{v} = 3\begin{bmatrix} -1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} -3 \\ 9 \\ 6 \end{bmatrix}.$$ ■

Example 1.2.3. Adding together two portfolios combines together all of the corresponding values. Adding together two bills of materials would produce the bill of materials for completing both tasks together. Adding together two histograms would produce the combined frequencies for each category when the two groups are joined. If $\boldsymbol{s}$ is a portfolio of shares, then on a given day a trader might buy (positive) or sell (negative) some number of shares on a given day. If the trader records the number of buys and sells as a *trade vector* $\boldsymbol{b}$, then $\boldsymbol{s} + \boldsymbol{b}$ would be the new portfolio after a day of trading.

Scaling a portfolio by scalar c would be replication of that client by a factor of c. For example, for a bill of materials $\boldsymbol{x}$, $c\boldsymbol{x}$ would be the bill of repeating the task c times. ■

Example 1.2.4. If two time series $\boldsymbol{s}$, $\boldsymbol{t}$ cover the same intervals of time, then $\boldsymbol{s}+\boldsymbol{t}$ would be the time series on the same time intervals representing the combined quantities. For example, if $\boldsymbol{s}$ and $\boldsymbol{t}$ are both audio files of the same length, then $\boldsymbol{s} + \boldsymbol{t}$ would be the audio file where both sounds are heard together. Likewise, $c\boldsymbol{t}$ would be the same audio file with the volume changed by a factor of c. ■

Not all vector sums are necessarily meaningful. First of all, only two n-vectors can be added together. If one vector is shorter than the other, what do we do with the extra entries which have no counterpart? Also, entries in a vector often have some measuring units attached to them, and it never makes sense to add together different units, e.g. what is foot plus hour? Unit conversion may resolve this issue, but oftentimes it will not. Even when the dimension and units associated to a vector sum are compatible, some ranges could be restricted that prohibit a sum. For example, if two boolean entries are added both equaling 1, then the sum would be two which is not a boolean value.[i] Lastly, even in the absent of arithmetic prohibition like we have seen, if the sum of two medical records expects us to add together the heights of two patients, this would result in given the height when one patient stands on the head of the other. While there is no meaningful medical data from this sum, we will still allow for such not meaningful calculations as we will see that some meaningless calculations could gain important meaning as the context shifts.

Theorem 1.2.5. *For any vectors $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ and any scalars $c, d \in \mathbb{R}$,*

(i) ***Additive Commutativity:*** $\boldsymbol{u} + \boldsymbol{v} = \boldsymbol{v} + \boldsymbol{u}$

(ii) ***Additive Associativity:*** $(\boldsymbol{u} + \boldsymbol{v}) + \boldsymbol{w} = \boldsymbol{u} + (\boldsymbol{v} + \boldsymbol{w})$

(iii) ***Additive Identity:*** $\boldsymbol{u} + \boldsymbol{0} = \boldsymbol{0} + \boldsymbol{u} = \boldsymbol{u}$

(iv) ***Additive Inverse:*** $\boldsymbol{u}+(-\boldsymbol{u}) = (-\boldsymbol{u})+\boldsymbol{u} = \boldsymbol{0}$

[i] There is a simple remedy to this problem. Addition on booleans is replaced with the $\vee$ operator: $0 \vee 0 = 0$, $0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1$. The operator $\vee$ is just the "or" operator in programming.

(v) ***Multiplicative Identity:*** $1\boldsymbol{u} = \boldsymbol{u}$

(vi) ***Multiplicative Associativity:*** $c(d\boldsymbol{u}) = (cd)\boldsymbol{u}$

(vii) ***Left Distributivity:*** $c(\boldsymbol{u} + \boldsymbol{v}) = c\boldsymbol{u} + c\boldsymbol{v}$

(viii) ***Right Distributivity:*** $(c + d)\boldsymbol{u} = c\boldsymbol{u} + d\boldsymbol{u}$

1.2.2 Linear Combinations

Definition 1.2.6. Given vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n \in \mathbb{R}^m$ and scalars $c_1, c_2, \ldots, c_n \in \mathbb{R}$, the vector $\boldsymbol{x}$ given as

$$\boldsymbol{x} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \ldots + c_n\boldsymbol{v}_n$$

is called a **linear combination** of $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ with **coefficients** $c_1, c_2, \ldots, c_n$. We say that the linear combination is an **affine** if $c_1 + c_2 + \ldots + c_n = 1$. Further, we say that an affine combination is **convex** if additionally $c_i \geq 0$ for all i.

Finally, a **vector space** is a set of vectors for which all possible linear combinations of vectors wherein are likewise contained wherein. In other words, if V is a vector space such that $\boldsymbol{u}, \boldsymbol{v} \in V$, then $a\boldsymbol{u} + b\boldsymbol{v} \in V$ for all scalars $a, b \in \mathbb{R}$.

A linear combination is the most general way of combine a set of vectors using the vector operations of addition and scalar multiplication. In one regard, every vector vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ is a linear combination with respect to the standard unit vectors $\boldsymbol{e}_i$, that is,

$$\boldsymbol{x} = x_1\boldsymbol{e}_1 + x_2\boldsymbol{e}_2 + \ldots + x_n\boldsymbol{e}_n.$$

Less standard linear combinations will also prove useful.

Example 1.2.7. The convex combination of vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ using the coefficients $x_i = \frac{1}{n}$ for all i gives the **average vector** $\overline{\boldsymbol{x}}$:

$$\overline{\boldsymbol{x}} = \frac{1}{n}\boldsymbol{v}_1 + \frac{1}{n}\boldsymbol{v}_2 + \ldots + \frac{1}{n}\boldsymbol{v}_n.$$

While it does not make sense to simply add together the profiles of a group of 9-year-old girls including height, weight, and daily intake of water, it is very meaningful to compute the average of these profile vectors.

In general, a convex combination can be seen as a mixture of the vectors. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two RGB color vectors. While $\boldsymbol{x} + \boldsymbol{y}$ is not a color, $0.5\boldsymbol{x} + 0.5\boldsymbol{y}$ is. In fact, this is the blend of the two colors. The color $0.25\boldsymbol{x} + 0.75\boldsymbol{y}$ would be the color which is 25% color $\boldsymbol{x}$ and 75% color $\boldsymbol{y}$. Hence, convex combinations should be viewed as *weighted averages* of vectors. ∎

Theorem 1.2.8. *The following properties hold for any $\mathbb{R}$-vector space V. Let $\boldsymbol{u} \in V$ and $c \in \mathbb{R}$.*

(i) The zero vector $\mathbf{0}$ in V is unique.

(ii) The additive inverse of $\boldsymbol{u}$ is unique.

(iii) $0\boldsymbol{u} = \mathbf{0}$.

(iv) $c\mathbf{0} = \mathbf{0}$.

(v) $-\boldsymbol{u} = (-1)\boldsymbol{u}$.

Python Programming

Addition, subtraction, and scaling of vectors is simple to implement in Python, so long as the two vectors have the same length, that is, the same number of entries. Simply use +, -, and *, respectively.

Example 1.2.9. The code:

```
import numpy as np

a = np.array([1,2])
b = np.array([3,4])
alpha = -0.5
beta = 1.5

c = alpha*a + beta*b
print(c)
```

produces the output:

```
[4.  5.]
```

■

Exercises

(Go to Solutions)

For Exercises 1-9, simplify the linear combination.

1. $3\begin{bmatrix}1\\2\end{bmatrix}-5\begin{bmatrix}0\\3\end{bmatrix}$

2. $2\begin{bmatrix}5\\-7\end{bmatrix}+\begin{bmatrix}-9\\6\end{bmatrix}+0\begin{bmatrix}3\\-23\end{bmatrix}$

3. $5\begin{bmatrix}1\\2\\4\end{bmatrix}+3\begin{bmatrix}0\\2\\1\end{bmatrix}+4\begin{bmatrix}1\\1\\1\end{bmatrix}$

♠ 4. $3\begin{bmatrix}-1\\0\\1\end{bmatrix}+2\begin{bmatrix}0\\0\\1\end{bmatrix}-2\begin{bmatrix}3\\4\\0\end{bmatrix}$

5. $\begin{bmatrix}3\\2\\1\end{bmatrix}+\begin{bmatrix}5\\8\\10\end{bmatrix}+\begin{bmatrix}0\\0\\5\end{bmatrix}$

6. $-1\begin{bmatrix}3\\0\\1\end{bmatrix}+\begin{bmatrix}1\\2\\4\end{bmatrix}+3\begin{bmatrix}2\\4\\1\end{bmatrix}$

7. $7\begin{bmatrix}3\\0\\2\end{bmatrix}+2\begin{bmatrix}8\\-3\\5\end{bmatrix}-3\begin{bmatrix}-2\\-2\\0\end{bmatrix}$

8. $8\begin{bmatrix}1\\4\\3\end{bmatrix}+2\begin{bmatrix}5\\0\\4\end{bmatrix}-4\begin{bmatrix}1\\0\\1\end{bmatrix}$

9. $\pi\begin{bmatrix}3\\-7\\\frac{3}{5}\end{bmatrix}+\frac{4}{3}\begin{bmatrix}4\\3\\7\end{bmatrix}$

10. $3\begin{bmatrix}4\\2\\5\end{bmatrix}+4\begin{bmatrix}2\\6\\1\end{bmatrix}-2\begin{bmatrix}1\\3\\5\end{bmatrix}+5\begin{bmatrix}1\\-1\\2\end{bmatrix}$

11. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/vector.ipynb

"Success isn't measured by money or power or social rank. Success is measured by your discipline and inner peace." – Mike Ditka

1.3 Inner Products

In this section, we introduce the notion of an inner product, often called the dot product. The inner product is a simple algebraic operation with many important applications, including developing geometric notions such as distance and length. A vector space equipped with an inner product is called, shockingly, an inner product space.

1.3.1 The Dot Product

Definition 1.3.1. We define the **dot product** $\cdot : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ of two vectors, $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ by the rule

$$\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u}^\top \boldsymbol{v} = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n.$$

Example 1.3.2. Given $\boldsymbol{u} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ and $\boldsymbol{v} = \begin{bmatrix} -5 \\ 0 \\ 3 \end{bmatrix}$, their dot product is

$$\boldsymbol{u} \cdot \boldsymbol{v} = 1(-5)) + 2(0) + 3(3) = -5 + 0 + 9 = \boxed{4}.$$ ∎

The dot product is also possible for block vectors, so long as vectors in each corresponding position have the same dimension. Hence,

$$\boldsymbol{u} \cdot \boldsymbol{v} = \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \vdots \\ \boldsymbol{u}_n \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_n \end{bmatrix} = \boldsymbol{u}_1 \cdot \boldsymbol{v}_1 + \boldsymbol{u}_2 \cdot \boldsymbol{v}_2 + \ldots + \boldsymbol{u}_n \cdot \boldsymbol{v}_n.$$

Note that for 1-vectors (which are really just scalars), the dot product is the same as the usual product of real numbers. Hence, the above formula generalizes the original.

Theorem 1.3.3. *Let $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$ and let $c \in \mathbb{R}$. Then*

(i) $\boldsymbol{u} \cdot (\boldsymbol{v} + \boldsymbol{w}) = \boldsymbol{u} \cdot \boldsymbol{v} + \boldsymbol{u} \cdot \boldsymbol{w}$;

(ii) $\boldsymbol{u} \cdot (c\boldsymbol{v}) = c(\boldsymbol{u} \cdot \boldsymbol{v})$;

(iii) $\boldsymbol{u} \cdot \boldsymbol{v} = \overline{\boldsymbol{v} \cdot \boldsymbol{u}}$;

(iv) $\boldsymbol{u} \cdot \boldsymbol{u} \geq 0$, *and* $\boldsymbol{u} \cdot \boldsymbol{u} = 0$ *if and only if* $\boldsymbol{u} = \boldsymbol{0}$.

1.3.2 Applications of Inner Products

Dot products are essential in many statistical calculations, particularly in measurements of *center*.

Example 1.3.4. Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ be an n-vector and consider the standard unit vector $\boldsymbol{e}_i$. Then

$$\boldsymbol{e}_i \cdot \boldsymbol{x} = x_i.$$

If $\boldsymbol{b} = \boldsymbol{e}_{i_1} + \boldsymbol{e}_{i_2} + \ldots + \boldsymbol{e}_{i_k}$, that is, $\boldsymbol{b}$ is some boolean vector, then

$$\boldsymbol{b} \cdot \boldsymbol{x} = x_{i_1} + x_{i_2} + \ldots + x_{i_k},$$

that is, the dot product of $\boldsymbol{b}$ and $\boldsymbol{x}$ will be that sum of entries of $\boldsymbol{x}$ which correspond to ones in $\boldsymbol{b}$. Hence,

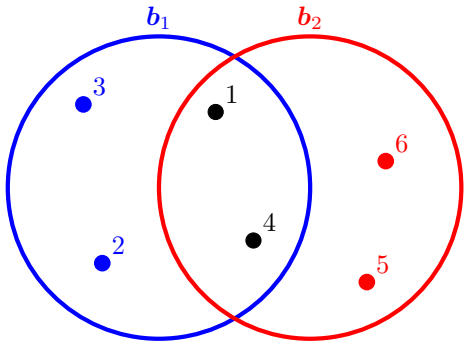
$$\mathbf{1} \cdot \boldsymbol{x} = x_1 + x_2 + \ldots + x_n.$$

■

Example 1.3.5. Generalizing Example 1.3.4, if $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are both boolean vectors, then $\boldsymbol{b}_1 \cdot \boldsymbol{b}_2$ will equal the number of ones that occur exactly in the same position between the two vectors. If these vectors represent the presence of certain features, then the dot product computes the number of co-occurrences between the two profiles. For example, consider the vectors $\boldsymbol{b}_1 = (1, 1, 1, 1, 0, 0)$ and $\boldsymbol{b}_2 = (1, 0, 0, 1, 1, 1)$. Hence,

$$\boldsymbol{b}_1 \cdot \boldsymbol{b}_2 = 1 + 0 + 0 + 1 + 0 + 0 = 2$$

counts the two overlaps between these vectors.



■

Example 1.3.6. Another application of boolean vectors and inner products might be for "masking" the data of another vector. That is, *data masking* is a data security technique used to protect sensitive or confidential information by replacing, encrypting, or otherwise disguising original data with fictional or altered data. The primary purpose of data masking is to prevent unauthorized access to sensitive information while maintaining the data's usability for legitimate purposes, such as software testing, analytics, or development. For example, when trying to diagnose a disease perhaps only part of a medical record is considered and usedto identify individuals with that disease. Suppose that the 6-vector $\boldsymbol{x} = (55.2, 101, 9, 0.9222, 20, 2)$ is a patient's medical record but only $\boldsymbol{x}_{[3:5]}$ is needed for the medical test. Thus, the boolean vector $\boldsymbol{b} = (0, 0, 1, 1, 1, 0)$ records the necessary slice. All other entries could be masked with some kind of value variance or data substitution. Note that $\boldsymbol{x} \cdot \boldsymbol{b} = 29.9222 = (\boldsymbol{x} + 100\boldsymbol{e}_1) \cdot \boldsymbol{b}$, that is, the inner product is not affected by masking varying the first entry. Conversely, $(\boldsymbol{x} - 100\boldsymbol{e}_3) \cdot \boldsymbol{b} = -70.0778 \neq 29.9222$. ■

Generalizing Example 1.3.4 in a different way, let $\boldsymbol{x}$ be any n-vector and let $\boldsymbol{w}$ be a such that $w_i \geq 0$ for all i and $w_1 + w_2 + \ldots + w_n = 1$. Then $\boldsymbol{w}$ is called a **probability vector** and

$$\boldsymbol{w} \cdot \boldsymbol{x} = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

is the *weighted average* of the entries of $\boldsymbol{x}$. Note that $\mathbf{1}/n$ is the probability vector where all outcomes are *equally likely* and gives rise to the arithmetic mean.

Definition 1.3.7. Let $\boldsymbol{x} \in \mathbb{R}^n$. Then define the **mean** (or **average**) of $\boldsymbol{x}$ as

$$\text{avg}(\boldsymbol{x}) = \frac{\mathbf{1} \cdot \boldsymbol{x}}{n} = \frac{1}{n}(x_1 + x_2 + \ldots + x_n).$$

Note that $\text{avg}(\boldsymbol{x}) \in \mathbb{R}$, a scalar, computes the arithmetic mean of the values of $\boldsymbol{x}$. When the vector $\boldsymbol{x}$ is clear from context, we often abbreviate the mean as $\mu = \text{avg}(\boldsymbol{x})$.

Let $\widetilde{\boldsymbol{x}} = \boldsymbol{x} - \mu\mathbf{1}$ denote the **de-meaned vector**. Note that

$$\text{avg}(\widetilde{\boldsymbol{x}}) = \frac{\mathbf{1}}{n} \cdot \left(\boldsymbol{x} - \left(\frac{\mathbf{1} \cdot \boldsymbol{x}}{n}\right)\mathbf{1}\right) = \frac{1}{n}\left(\mathbf{1} \cdot \boldsymbol{x} - \frac{(\mathbf{1} \cdot \boldsymbol{x})(\mathbf{1} \cdot 1)}{n}\right) = \frac{\mathbf{1} \cdot \boldsymbol{x} - \mathbf{1} \cdot \boldsymbol{x}}{n} = 0.$$

Therefore, $\widetilde{\boldsymbol{x}}$ is the vector formed by shifting all the entries in $\boldsymbol{x}$ so that the new mean is 0, that is, the mean has been centered at zero. Hence, $\widetilde{x}_i$ is exactly the amount that x_i varies from $\text{avg}(\boldsymbol{x})$.

Example 1.3.8. If $\boldsymbol{x} = (1, -2, 3, 2)$, then $\text{avg}(\boldsymbol{x}) = \dfrac{1-2+3+2}{4} = 1$ and $\widetilde{\boldsymbol{x}} = \boldsymbol{x} - \mathbf{1} = (0, -3, 2, 1)$. ∎

Suppose a quantity X can vary based upon the outcome of a random experiment. Then X is called a **random variable**. If X can only take on finitely many possibilities, say $\{x_1, x_2, \ldots, x_n\}$, then X is called a **discrete random variable** and can be encoded by the vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$. Let $\boldsymbol{p}$ be a probability vector associated to X, that is, if $\mathcal{P}(X = x_i)$ denotes the probability that X is assigned to x_i, then we have that $\mathcal{P}(X = x_i) = p_i$. Then the **expected value** (or **mean**) of the random variable X is $E(X) = \boldsymbol{p} \cdot \boldsymbol{x}$.

Example 1.3.9. Suppose that a bank is studying the transaction times of its tellers. The lengths of time spent on observed transactions, rounded to the nearest minute, are shown in the following frequency table.

Time	1	2	3	4	5	6	7	8	9	10	
Frequency	3	5	9	12	15	11	10	6	3	1	(Total = 75)

We can then consider the random variable X that a specific transaction will x minutes, e.g. $x = 1$, $x = 2$, $x = 3$, etc. The frequencies can be converted to probabilities by dividing each frequency by the total number of transactions to get the results in the following relative frequence table.

Time	1	2	3	4	5	6	7	8	9	10
Relative Frequency	0.04	0.07	0.12	0.16	0.2	0.15	0.13	0.08	0.04	0.01

If you were a customer at this bank, how long should you *expect* your transaction to be? The expected value is given by

$$\boldsymbol{X} \cdot \boldsymbol{p} = 1(0.04)+2(0.07)+3(0.12)+4(0.16)+5(0.20)+6(0.15)+7(0.13)+8(0.08)+9(0.04)+10(0.01) = 5.09$$

Thus, the average time a person can expect to spend with the bank teller is $\boxed{5.09 \text{ minutes}}$. ∎

Even if the assumptions on $\boldsymbol{w}$ are removed, that is, even if $\boldsymbol{w}$ is not a probability vector, the product $\boldsymbol{w} \cdot \boldsymbol{x}$ represents a *weighted sum* (or *score*) of the entries of $\boldsymbol{x}$, where w_i represents the *weight* associated to position i determined by its significance.

Example 1.3.10. Imagine a fictitious election for Math Club president consisting of 11 voters and three candidates:

- Candidate 1 (c_1) : The Algebra Party
- Candidate 2 (c_2) : The Trigonometry Party
- Candidate 3 (c_3) : The Geometry Party

Candidate	**1st Choice**	**2nd Choice**	**3rd Choice**
$\boldsymbol{c_1}$	5	0	6
$\boldsymbol{c_2}$	2	7	2
$\boldsymbol{c_3}$	4	4	3

Suppose a candidate gets two points for being a first choice and one point for being second choice. The candidate with the most points wins. This gives us the weight vector $\boldsymbol{w} = (2, 1, 0)$. We candidate has a vote

vector: $\boldsymbol{c}_1 = (5, 0, 6)$, $\boldsymbol{c}_2 = (2, 7, 2)$, and $\boldsymbol{c}_3 = (4, 4, 3)$. The score of candidate c_i is then the dot product $\boldsymbol{w} \cdot \boldsymbol{c}_i$:

$$c_1 : 10; \qquad c_2 : 11; \qquad \boxed{c_3 : 12}.$$

By this voting method, c_3, the Geometry Party, wins the election. ∎

Example 1.3.11. Let $\boldsymbol{p}$ be the vector listing the prices for commodities $1, 2, \ldots, n$, and let $\boldsymbol{q}$ be the number of sales of each of these n commodities, that is, $\boldsymbol{p}$ is the *price vector* and $\boldsymbol{q}$ is the *sales* (or *quantity*) *vector*. Then the **revenue** from these sales is the dot product $\boldsymbol{p} \cdot \boldsymbol{q}$. ∎

Example 1.3.12. Suppose a time interval is divided into n not-necessarily equal subintervals. Let $\boldsymbol{t}$ be the *time vector* where t_i is the length of the ith subinterval of time, and let $\boldsymbol{v}$ be the *velocity vector* where v_i is the constant velocity of a moving object during the ith time subinterval. Then the dot product $\boldsymbol{v} \cdot \boldsymbol{t}$ is the total distance traveled by this object. ∎

The last few examples show that the dot product is the linear algebraic version of an *integral*.

Python Programming

There are two easy ways to compute dot products in Python. Suppose that $\boldsymbol{x}$ and $\boldsymbol{y}$ are already established as vectors in Python. Then the dot product is computed by the command `np.inner(x,y)` or `x @ y`.

Example 1.3.13. The code:

```
import numpy as np

x = np.array([1,2,3])
y = np.array([0,-3,5])
print(x @ y, np.inner(x,y), "\n")


# Create two small boolean vectors.
bool_x = np.array([0, 1, 1, 1, 0, 1, 0, 0, 0])
bool_y = np.array([0, 1, 1, 0, 0, 0, 1, 0, 1])

# Taking the dot product allows you to see the number of enabled features common between
    them.
print("Common Features (small):", bool_x @ bool_y)

# Create two 1000-vectors of random boolean values.
large_bool_x = np.array([np.random.choice([0, 1]) for i in range(1000)])
large_bool_y = np.array([np.random.choice([0, 1]) for i in range(1000)])
# print("large_bool_x =", large_bool_x) #Uncomment if you want to see the vectors,
# print("large_bool_y =", large_bool_y) #but they are too big to fit here.

# A dot product is used once again, not any harder then on the much smaller vectors.
print("Common Features (large):", large_bool_x @ large_bool_y, '\n')
```

produces the output:

```
9 9

Common Features (small):  2
Common Features (large):  265
```

■

Exercises (Go to Solutions)

For Exercises 1-9, compute the given inner products.

1. $\begin{bmatrix} 1 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ -3 \end{bmatrix}$

♠ 2. $\left(3\begin{bmatrix} 1 \\ -2 \end{bmatrix}\right) \cdot \left(2\begin{bmatrix} 2 \\ -3 \end{bmatrix}\right)$

3. $\begin{bmatrix} 7 \\ 6 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ -1 \\ 5 \end{bmatrix}$

♠ 4. $\begin{bmatrix} 7 \\ 1 \\ -5 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}$

5. $\left(5\begin{bmatrix} 7 \\ 1 \\ -5 \end{bmatrix}\right) \cdot \left(2\begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}\right)$

♠ 6. $\left(\begin{bmatrix} 1 \\ -5 \\ 0 \end{bmatrix} + \begin{bmatrix} 3 \\ 2 \\ -3 \end{bmatrix}\right) \cdot \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$

♠ 7. $\left(2\begin{bmatrix} -1 \\ -1 \\ 3 \end{bmatrix}\right) \cdot \left(\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ -5 \\ 3 \end{bmatrix}\right)$

♠ 8. $\begin{bmatrix} 2 \\ 1 \\ -3 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 1 \\ -1 \\ 2 \end{bmatrix}$

9. $\left(2\begin{bmatrix} 7 \\ 1 \\ -3 \\ 2 \end{bmatrix}\right) \cdot \left(3\begin{bmatrix} 7 \\ 1 \\ -1 \\ 2 \end{bmatrix}\right)$

In physics, **work** W is the linear effort done by a force vector $\boldsymbol{F}$ applied over a distance vector $\boldsymbol{d}$. Intuitively, work is a measure of effort expended when moving an object by applying a force to it. Unlike velocity and force, which are vector quantities, work is a scalar.

Theorem. *If a constant force $\boldsymbol{F}$ is applied to an object and moves the object in a straight line a distance $\boldsymbol{d}$, then the work W performed by the force is*

$$W = \boldsymbol{F} \cdot \boldsymbol{d} = \|\boldsymbol{F}\|\|d\| \cos\theta \tag{1.3.1}$$

where θ is the angle between the force $\boldsymbol{F}$ and $\boldsymbol{d}$.

For Exercises 10-18, find the work done by the force. Recall that $\boldsymbol{i} := \boldsymbol{e}_1 = (1, 0)$ and $\boldsymbol{j} := \boldsymbol{e}_2 = (0, 1)$, a notation commonly used in physical applications.

10. $\boldsymbol{F} = 24\boldsymbol{i} + 7\boldsymbol{j}$, $\boldsymbol{d} = 30\boldsymbol{i} + 4\boldsymbol{j}$

11. $\boldsymbol{F} = -8\boldsymbol{i} + 17\boldsymbol{j}$, $\boldsymbol{d} = 6\boldsymbol{i} + 50\boldsymbol{j}$

12. $\boldsymbol{F} = 19\boldsymbol{j}$, $\boldsymbol{d} = 33\boldsymbol{i}$

13. A shipping clerk pushes a heavy package across the floor. He applies a force of 64 pounds in a downward direction, making an angle of 35° with the horizontal. The package is moved 25 feet.

14. A force $\boldsymbol{F} = 35\boldsymbol{i} - 12\boldsymbol{j}$ (in pounds) is used to push an object up a ramp. The resulting movement of the object is represented by the displacement vector $\boldsymbol{d} = 15\boldsymbol{i} + 4\boldsymbol{j}$ (in feet).

15. A package is pushed across a floor a distance of 70 feet by exerting a force of 46 pounds downward at an angle of 25° with the horizontal.

16. A package is pushed across a floor a distance of 90 feet by exerting a force of 41 pounds downward at an angle of 20° with the horizontal.

17. Joseph pulls Emma and Olivia in a wagon by exerting a force of 20 pounds on the handle at an angle of 30° with the horizontal. Joseph pulls the wagon a distance of 300 feet.

18. An automobile is pushed 110 feet down a level street by exerting a force of 87 pounds at an angle of 15° with the horizontal.

19. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/inner.ipynb

"When the norm is decency, other virtues can thrive: integrity, honesty, compassion, kindness, and trust."
– Raja Krishnamoorthi

1.4 Norms

With the inner product now in hand, we can utilize it to describe the norm of a vector, that is, the magnitude of the vector (or the length of the arrow). This gives us also a distance function between vectors which leads to numerous applications.

1.4.1 The Norm of a Vector

Definition 1.4.1. The **norm** of a vector $\boldsymbol{v} \in \mathbb{R}^n$ is the nonnegative scalar $\|\boldsymbol{v}\|$ defined by

$$\|\boldsymbol{v}\| = \sqrt{\boldsymbol{v} \cdot \boldsymbol{v}} = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}.$$

We say that $\boldsymbol{v}$ is a **unit vector** if $\|\boldsymbol{v}\| = 1$.

Geometrically, the norm of a vector $\boldsymbol{x}$ is the *length* of the arrow drawn in $\mathbb{R}^n$, that is, the norm is the *magnitude* of the vector.

It is useful to note that $\|\boldsymbol{v}\|^2 = \boldsymbol{v} \cdot \boldsymbol{v}$.

Theorem 1.4.2. *Let $\boldsymbol{u} \in \mathbb{R}^n$ and $c \in \mathbb{R}$. Then*

$$\|c\boldsymbol{v}\| = |c|\|\boldsymbol{v}\|$$

Example 1.4.3. Let $\boldsymbol{v} = \begin{bmatrix} 1 \\ 0 \\ 2 \\ -2 \end{bmatrix} \in \mathbb{R}^4$. Then the norm of $\boldsymbol{v}$ is

$$\|\boldsymbol{v}\| = \sqrt{\boldsymbol{v} \cdot \boldsymbol{v}} = \sqrt{1(1) + 0(0) + 2(2) - 2(-2)} = \sqrt{1 + 4 + 4} = \sqrt{9} = \boxed{3}.$$

Since the norm of $\boldsymbol{v}$ is 3, $\boldsymbol{v}$ is not a unit vector. On the other hand, let $\boldsymbol{u} = \frac{1}{3}\boldsymbol{v} = \begin{bmatrix} 1/3 \\ 0 \\ 2/3 \\ -2/3 \end{bmatrix}$. Then

$$\|\boldsymbol{u}\| = \left\| \frac{1}{3}\boldsymbol{v} \right\| = \frac{1}{3}\|\boldsymbol{v}\| = \frac{1}{3}(3) = \boxed{1}.$$

Therefore, $\boldsymbol{u}$ is a unit vector in the same direction as $\boldsymbol{v}$. ∎

The process of constructing a unit vector in the same direction as a given vector is called **normalization**. The normalization of any nonzero vector $\boldsymbol{v}$ is given by $\dfrac{1}{\|\boldsymbol{v}\|}\boldsymbol{v}$. The zero vector cannot be normalized. In particular, the zero vector does not point in any direction.

1.4.2 The Distance Function

Example 1.4.4. Recall that a digital picture can be represented as a matrix. Facial recognition software is artificial intelligence meant to, first, recognize human faces in a picture, and, second, match the face in a photo to a face in a database. The first task boils down to the AI being able to decide when a submatrix of the matrix is a face. The second task very much motivates the topic at hand. How does one know that two pictures of faces are a match? How close do two pictures have to be in order to declare a match? The word "close" here suggests a distance. While intuitively humans can say whether two faces are similar or not, AI has no intuition and must be taught how to do this. As such, facial recognition software measures the "distance" between to photo vectors to determine whether a match is found or not. ∎

Definition 1.4.5. For $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$, the **distance** between $\boldsymbol{u}$ and $\boldsymbol{v}$, denoted as $\text{dist}(\boldsymbol{u}, \boldsymbol{v})$, is the length of the vector $\boldsymbol{u} - \boldsymbol{v}$, that is,

$$\text{dist}(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\|.$$

Example 1.4.6. Let $\boldsymbol{u} = \begin{bmatrix} 7 \\ 1 \end{bmatrix}$ and $\boldsymbol{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$. Then

$$\text{dist}(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\| = \left\| \begin{bmatrix} 4 \\ -1 \end{bmatrix} \right\| = \sqrt{4^2 + (-1)^2} = \boxed{\sqrt{17}}.$$ ∎

The definition of distance that we presented above is actually known as the *Euclidean distance* between vectors, as this is exactly the distance formula from standard geometry, also known as *Euclidean geometry*. There are interestingly alternative notions of distance one can develop, such as the previous example about facial recognition. These distance functions essentially always are derived from some norm (the norm introduced above known as the *Euclidean norm*). While there is great merit in exploring non-Euclidean norms/distances, for the sake of simplicity, we will exclusively use the Euclidean norm/distance when discussing vectors.

Example 1.4.7. Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be two boolean n-vectors which list the presence (or absence) of n features. Then $\text{dist}(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\|$ would be the number of times one vector had a 0 and the other vector had a 1. In other words, the distance between the vectors measures the number of differences between the profiles.

Generalizing this example, let $\boldsymbol{u}$ and $\boldsymbol{v}$ be two n-vectors which record the measurements of features and not just recording their occurrences, that is, the entries are nonnegative real numbers instead of just booleans. For example, these vectors could be profiles of patients who enter a hospital: the first entry is body weight, the second age, the third chest pain (on a scale of one to ten), the fourth difficulty breathing (on a scale of one to ten), and the remaining entries are results of medical test performed. The distance between two medical records here measures, in some degree, how similar the two patients are. A medical scientist experimenting a new treatment may investigate the treatment's affect on patients based upon how "close" two patients are to each other. One might hypothesize that patients whose profiles will be close (the distance between them is small) will have a similar affect to the treatment.

The previous measurement of distance between these profiles can easily be biased. In the medical record, one entry could be the patient's weight and age, which could vary largely among humans, in some cases by hundreds of pounds and decades of age. This wide variance has important implications on patient's health. On the other hand, if a patient records a chest pain of 10 (the most severe), this is only 9 points away from a patient with 1 chest pain (the least severe). This distance would be measured less significant than two patients who differ in age by a decade or who differ in weight by 10 pounds. Similarly, if one entry in the medical record is a boolean that records 1 if diabetic and 0 otherwise, this difference would be considered small compared to a few centimeter difference in someone's height, which probably should not be the case.

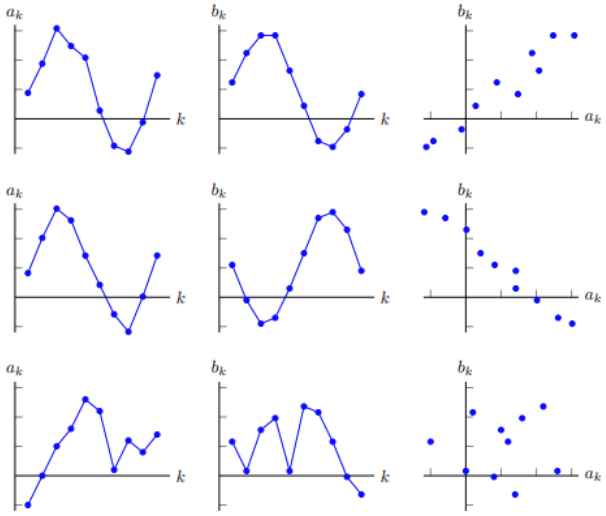
Hence, the Euclidean distance (just like any distance function) can measure the gap between two vectors but that gap can be biased if the ranges of possible vector entries are not standard, which may justify a non-Euclidean norm. ∎

1.4.3 Statistical Measurements

Definition 1.4.8. Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$. Then the **correlation coefficient** of $\boldsymbol{x}$ and $\boldsymbol{y}$, denoted ρ, is given as

$$\rho = \frac{\widetilde{\boldsymbol{x}} \cdot \widetilde{\boldsymbol{y}}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}.$$

Note that $\rho = \cos(\angle(\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{y}}))$[i], hence $-1 \leq \rho \leq 1$. As such, ρ is often expressed as a percentage. By the Cauchy-Schwarz Inequality,[ii] $\rho = 1$ if and only if $\widetilde{\boldsymbol{x}} = a\widetilde{\boldsymbol{y}}$ for some positive scalar a. Likewise, $\rho = -1$ if and only if $\widetilde{\boldsymbol{x}} = a\widetilde{\boldsymbol{y}}$ for some negative scalar a. Finally, $\rho = 0$ if and only if $\rho = 1$ if and only if $\widetilde{\boldsymbol{x}}$ and $\widetilde{\boldsymbol{y}}$ are orthogonal. Hence, the degree at which we say two vectors are correlated is measured by the angle between their de-meaned vectors. Why? When the vector is de-meaned, then the entries measure the displacement from the mean. When these two de-meaned vectors are close, it means the original vector deviate from their average similarly. This allows us to compare apples to apples.



Example 1.4.9. Three pairs of vectors $\boldsymbol{a}$, $\boldsymbol{b}$ of length 10, with correlation coefficients 0.968 (top), -0.988 (middle), and 0.004 (bottom). ∎

Definition 1.4.10. Let $\boldsymbol{x} \in \mathbb{R}^n$. Then the **root-mean-square** (RMS) of $\boldsymbol{x}$ is defined as

$$\text{rms}(\boldsymbol{x}) = \frac{\|\boldsymbol{x}\|}{\sqrt{n}} = \sqrt{\frac{x_1^2 + \ldots + x_n^2}{n}}.$$

The importance of RMS is revealed by Chebyshev's inequality.[iii] Note that if $k \leq \dfrac{\|\boldsymbol{x}\|^2}{a^2}$ then

$$\frac{k}{n} \leq \left(\frac{\text{rms}(\boldsymbol{x})}{a}\right)^2, \tag{1.4.1}$$

since $\dfrac{k}{n} \leq \dfrac{\|\boldsymbol{x}\|^2}{na^2} = \left(\dfrac{\|\boldsymbol{x}\|}{\sqrt{n}}\right)^2 \left(\dfrac{1}{a^2}\right) = \left(\dfrac{\text{rms}(\boldsymbol{x})}{a}\right)^2$. The left-hand side of the above modified Chebyshev's inequality is the percentage of those entries of $\boldsymbol{x}$ bounded below by a. To consider the right-hand side, suppose that $a = m * \text{rms}(\boldsymbol{x})$. Hence the right-hand side simplifies to $\dfrac{1}{m^2}$. Thus, the percentage of entries exceeding the RMS of the vector is bounded above by the reciprocal square of this factor. For example, there can be at most $\dfrac{1}{25} = 4\%$ of the entries that are larger than 5 times the RMS of the vector. For this reason, we can interpret $\text{rms}(\boldsymbol{x})$ as the typical value of the value (as not too many values can be larger than this one).

Note that $\text{rms}(\mathbf{1}) = 1$.

---

[i] The symbol $\angle(\boldsymbol{u}, \boldsymbol{v})$ denotes the angle between vectors $\boldsymbol{u}$ and $\boldsymbol{v}$. See Appendix C for further details

[ii] (Cauchy-Schwarz Inequality) For all $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$, $|\boldsymbol{u} \cdot \boldsymbol{v}| \leq \|\boldsymbol{u}\|\|\boldsymbol{v}\|$.

[iii] (Chebyshev's Inequality) Let $\boldsymbol{x} \in \mathbb{R}^n$ and $a > 0$. Suppose there are at least k entries of $\boldsymbol{x}$ which satisfy the inequality $|x_i| \geq a$. Then $\dfrac{\|\boldsymbol{x}\|^2}{a^2} \geq k$.

While RMS measures the average value of a vector, it in many ways behaves like a norm (as it is just a scalar multiple of the Euclidean norm). As such, $\text{rms}(\boldsymbol{x}-\boldsymbol{y})$ in many ways behaves likes a distance function. We should interpret $\text{rms}(\boldsymbol{x}-\boldsymbol{y})$ to measure the average error difference between the entries of the two vectors. For example, suppose that $\widehat{\boldsymbol{t}}$ is a time series, say the hourly temperatures forecasted for tomorrow. Tomorrow comes and the actual hourly temperatures are recorded as the vector $\boldsymbol{t}$. Then $\text{dist}(\boldsymbol{t}-\widehat{\boldsymbol{t}})$ would measure the error in the forecast, the so-called *prediction error*. On the other hand, $\text{rms}(\boldsymbol{t}-\widehat{\boldsymbol{t}})$, called the *RMS prediction error*, instead measures the average error for each hour in the time series. If this value is small, then we can say the prediction is good, and we say that RMS prediction error is a better measurement[iv] of the accuracy of the forecast, compared just to the prediction error, since the standard prediction error can easily be skewed by a few extremely good or bad predictions.

Definition 1.4.11. Let $\boldsymbol{x} \in \mathbb{R}^n$. The **standard deviation** of a vector is the RMS of $\widetilde{\boldsymbol{x}}$, that is,

$$\text{std}(\boldsymbol{x}) = \text{rms}(\widetilde{\boldsymbol{x}}) = \frac{\|\boldsymbol{x}-\mu\mathbf{1}\|}{\sqrt{n}} = \sqrt{\frac{(x_1-\mu)^2+(x_2-\mu)^2+\ldots+(x_n-\mu)}{n^{\text{v}}}}.$$

Standard deviation measures the typical amount that an entry deviates from the mean and is hence a very effective measurement of spread of a data set.

Theorem 1.4.12. *Let $\boldsymbol{x} \in \mathbb{R}^n$ and $c \in \mathbb{R}$. Then*

(i) $\text{rms}(\boldsymbol{x})^2 = \text{avg}(\boldsymbol{x})^2 + \text{std}(\boldsymbol{x})^2$; *(ii)* $\text{std}(\boldsymbol{x}+c\mathbf{1}) = \text{std}(\boldsymbol{x})$; *(iii)* $\text{std}(c\boldsymbol{x}) = |c|\,\text{std}(\boldsymbol{x})$.

The first condition says that the functions of root-mean-square, average, and standard deviation satisfy a Pythagorean relation. The last two conditions allow us to **standardize** a vector $\boldsymbol{x}$, sometimes called the *z-score*, as

$$\boldsymbol{z} = \frac{1}{\text{std}(\boldsymbol{x})}(\boldsymbol{x}-\text{avg}(\boldsymbol{x})\mathbf{1}).$$

Note that $\text{avg}(\boldsymbol{z}) = 0$ and $\text{std}(\boldsymbol{z}) = 1$.

[iv] What about $\text{avg}(\boldsymbol{t}-\widehat{\boldsymbol{t}})$? This would give an average of the errors in prediction. The difference here is the geometry. We could measure the distance between two points (x_1, y_1) and (x_2, y_2) as $|x_2-x_1|+|y_2-y_1|$, as this does form a distance function (the so-called 1-*norm*), but instead we elect to use the distance function $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$ (the so-called 2- or *Euclidean norm*). The reasons for this are geometric and apply to this discussion of RMS too. In other words, while $\text{avg}(\boldsymbol{x})$ is the *arithmetic mean* of $\boldsymbol{x}$, $\text{rms}(\boldsymbol{x})$ is the *Euclidean mean* of $\boldsymbol{x}$ with its various consequences to geometry.

Python Programming

It is simple enough to code a norm function on vectors. The following code, for example, could implement it:

```
import numpy as np

def vector_norm(vector):
    vector_sum = 0
    for i in x:
        vector_sum += i**2
    return np.sqrt(vector_sum)

x = np.array([1,2,5,12])
print(vector_norm(x))
```

The output would then be $\|\boldsymbol{x}\| = 13.19090595827292$, which is correct. On the other hand, the `norm` of a vector is already implemented within the sublibrary `linalg` in `numpy`, namely `np.linalg.norm(x)`. It is recommended this be used instead of your own implementation for two reasons. First, it saves you time when writing code and simplifies the code you are writing. Second, the preloaded procedures are generally going to be MUCH FASTER to run than those you write yourself. This is nothing against the reader, but there are advanced programming techniques utilized inside these black boxes, such as parallel processing, that go beyond the scope of this textbook. In generally, it is always a better idea to utilize the preloaded procedures if they are available.

With that said, there is no preloaded procedure in Python for the distance between two vectors. Thus, we just use the definition $\text{dist}(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\|$ to implement distance calculations very simply.

Example 1.4.13. The code:

```
import numpy as np

x = np.array([2,-1,2])
print(np.linalg.norm(x))            #compute the norm of x

u = np.array([1.8, 2.0, -3.7, 4.7])
v = np.array([0.6, 2.1, 1.9, -1.4])
w = np.array([2.0, 1.9, -4.0, 4.6])
print(np.linalg.norm(u-v))          #compute the distance between u and v
print(np.linalg.norm(u-w))          #compute the distance between u and w
print(np.linalg.norm(v-w))          #compute the distance between v and w
```

produces the output:

```
3.0
8.367795408588812
0.3872983346207417
8.532877591996735
```

■

Additionally, NumPy has preloaded functions for arithmetic mean and standard deviation, namely `np.mean(x)` and `np.std(x)`, respectively, where `x` is a vector. There is no preloaded procedure in Python for the root-mean-square. As such, a `lambda` function, a technique in Python to quickly code up simple, one-line procedures, is recommended. Here are two possible implementations:

```
rms = lambda v : np.linalg.norm(v)/np.sqrt(len(v))
```

or

```
rms = lambda v : np.sqrt(np.mean(v**2))
```

The first implementation follows directly from Definition 1.4.10. The second implementation, a little bit less intuitive at first, has two advantages. First, it does not utilize the package `linalg`, which could have complexity benefits on the code. Second, it suggests why RMS is called *root-mean-square*. Note that in Python when an arithmetic operation, namely: `+`, `-`, `*`, `/`, or `**`, which represent addition, subtraction, multiplication, division, and exponentiation, respectively, is used it is always performed component-wise. That is, v**2 means in Python to square each entry in the vector $\boldsymbol{v}$.

Example 1.4.14. The code:

```python
import numpy as np

rms = lambda v : np.sqrt(np.mean(v**2))

x = np.array([-2, 5, 0, 7, 10])          # Define a vector
print("Vector:", x)
print("Mean:", np.mean(x))               # Compute Average (Mean)
print("Standard Deviation:", np.std(x)) # Compute Standard Deviation
print("RMS:", rms(x))                    # Compute Root Mean Square (RMS)
```

produces the output:

```
Vector:  [-2 5 0 7 10]
Mean:  4.0
Standard Deviation:  4.427188724235731
RMS: 5.966573556070519
```

■

Exercises

(Go to Solutions)

For Exercises 1-6, find the norm of the vector.

1. $\begin{bmatrix} 3 \\ -1 \\ 5 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 7 \\ 1 \\ 5 \end{bmatrix}$

3. $\begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$

♠ 4. $\begin{bmatrix} 2 \\ 1 \\ -3 \\ 2 \end{bmatrix}$

5. $2\begin{bmatrix} 1 \\ 0 \\ -3 \end{bmatrix} - 3\begin{bmatrix} -2 \\ 1 \\ 3 \end{bmatrix}$

6. $\begin{bmatrix} 3 \\ -1 \\ 5 \end{bmatrix} + 3\begin{bmatrix} 1 \\ 0 \\ -3 \end{bmatrix} + \begin{bmatrix} -2 \\ 1 \\ 3 \end{bmatrix}$

For Exercises 7-12, find normalize each vector, that is, compute its normalization.

7. $\begin{bmatrix} 3 \\ 4 \end{bmatrix}$

8. $\begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$

9. $\begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$

10. $\begin{bmatrix} 4 \\ 0 \\ 3 \end{bmatrix}$

11. $\begin{bmatrix} 1 \\ 4 \\ -2 \end{bmatrix}$

12. $\begin{bmatrix} 3 \\ 7 \\ 12 \\ 8 \end{bmatrix}$

For Exercises 13-16, find the distance between the two vectors.

13. $\begin{bmatrix} 7 \\ 6 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ -1 \\ 5 \end{bmatrix}$

♠ 14. $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 5 \\ 2 \\ 2 \end{bmatrix}$

♠ 15. $\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ -5 \\ 3 \end{bmatrix}, 2\begin{bmatrix} 3 \\ -1 \\ -2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

16. $\left(\begin{bmatrix} 3 \\ -1 \\ -2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix}\right)\begin{bmatrix} -2 \\ 1 \\ -1 \end{bmatrix}, 2\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

For Exercises 17–18, compute $\text{avg}(\boldsymbol{x})$, $\text{rms}(\boldsymbol{x})$, and $\text{std}(\boldsymbol{x})$ for each vector $\boldsymbol{x}$ provided.

17. $\begin{bmatrix} 3 \\ 2 \\ -1 \end{bmatrix}$

18. $\begin{bmatrix} -2 \\ 0 \\ 4 \\ 5 \end{bmatrix}$

19. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/norm.ipynb

"You cannot eat a cluster of grapes at once, but it is very easy if you eat them one by one."
– Jacques Roumain

1.5 Clustering

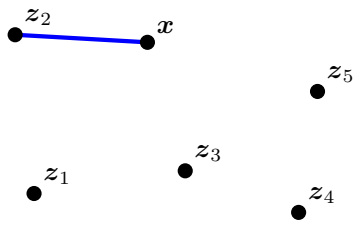
In this section, we approach a major application of linear algebra, known as the clustering problem. We develop the notions and ideas necessary to understand the clustering problem in this section, and in Section 1.6 we develop an effective algorithm to solve the clustering problem.

1.5.1 The Nearest Neighbor Problem

Definition 1.5.1. Let $\boldsymbol{z}_1, \ldots \boldsymbol{z}_m$ be a list of m-many n-vectors. Let $\boldsymbol{x}$ be another n-vector. We say that $\boldsymbol{z}_j$ is the *nearest neighbor* of $\boldsymbol{x}$ if

$$\|\boldsymbol{x} - \boldsymbol{z}_j\| \leq \|\boldsymbol{x} - \boldsymbol{z}_i\|$$

for all $1 \leq i \leq m$.

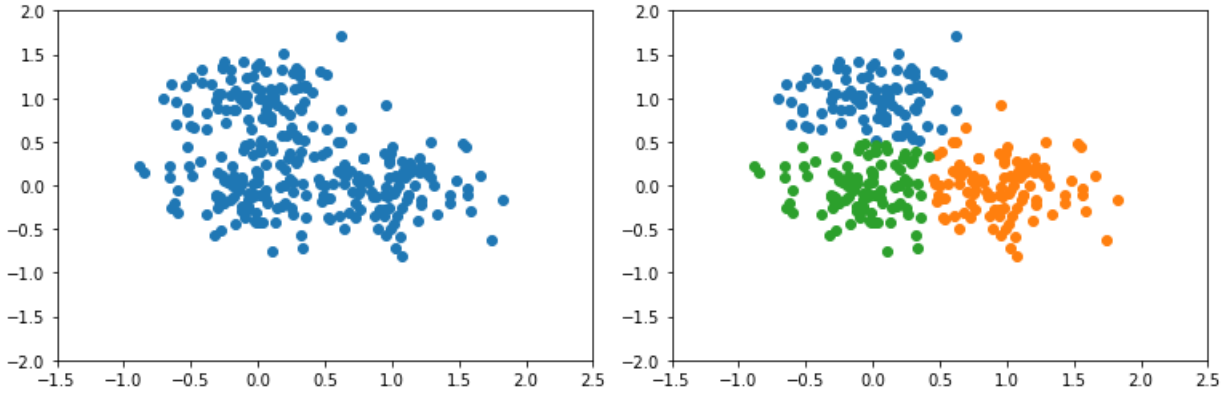


Therefore, the nearest neighbor to $\boldsymbol{x}$ is the minimal solution $\boldsymbol{z}_j$ to the objective $J = \|\boldsymbol{x} - \boldsymbol{z}_i\|$ as $1 \leq i \leq m$.

1.5.2 The Clustering Problem

Image we have a list of n-many n-vectors, say $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$. Suppose we want to sort these n vectors into k-many *groups* or *clusters* so that every vector in a cluster is *similar* to all the other vectors in its same group and *dissimilar* from the vectors in other groups. Clearly, $k \leq n$ and generally k is much, much smaller than n, e.g. $n = 300$, $k = 3$.

Example 1.5.2. The accompanying illustration displays twice a set of 300 points in the plane. On the left, no clustering is illustrated. On the right, the points can be clustered in the three groups shown, distinguished by color.



■

In the previous example, it is easy to see that the clustering is *good* and that $k = 3$ is a good choice of clusters. As most data sets will involve vectors with more than 3 entries, geometrically visualizing clustering will be extremely difficult (if not impossible). How can we measure if a clustering is *good* or not? Might that measurement depend on the number of clusters? How do we know the right number of clusters?

Definition 1.5.3. Let X be a set of vectors. Let $C_i \subseteq X$ for $1 \leq i \leq k$. If

(i) $C_1 \cup C_2 \cup \ldots \cup C_k = X$,

(ii) $C_i \cap C_j = \emptyset$ whenever $i \neq j$,

then we call $\{C_1, C_2, \ldots, C_k\}$ a **partition** of X. Each of the subsets of X contained in the partition is called a **cluster** (**part**, **class**, or **cell**).

In more simple terms, a partition of a set is a division of the elements so that every element belongs to one and only one part of that division. In particular, different cells never overlap. Let $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ be the set of these vectors which we wish to cluster. The Clustering Problem boils down to constructing a partition of X with k parts so that vectors in the same cell are *similar*. This last part is the hard part, of course, and we will revisit it in a moment.

Maybe it comes as no surprise the partition of an indexed set is itself a vector. Let $\{C_1, C_2, \ldots, C_k\}$ be a partition of $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$. Let $\boldsymbol{c}$ be an n-vector such that $c_i = j$ exactly when $\boldsymbol{x}_i \in C_j$. Then it holds that

$$C_j = \{\boldsymbol{x}_i \mid c_i = j\}.$$

Hence, the vector $\boldsymbol{c}$ encodes naturally the clustering associated to the partition of X. We will let $C_{\boldsymbol{x}_i}$ denote the cluster that contains $\boldsymbol{x}_i$, that is, $\boldsymbol{x}_i \in C_{\boldsymbol{x}_i} = C_{c_i}$.

Example 1.5.4. Let $X = \{1, 2, 3, 4, 5\}$. Then if $C_1 = \{2, 3, 4\}, C_2 = \{1\}, C_3 = \{5\}$, then

$$\{C_1, C_2, C_3\} = \{\{2, 3, 4\}, \{1\}, \{5\}\}$$

is a partition on X. The associated partition vector is given as $\boldsymbol{c} = (2, 1, 1, 1, 3)$. ∎

If two vectors are *similar* then they are *close* together, which is a measurement of distance. Hence, our goal will be to cluster vectors according to their Euclidean distance from each other. While there is a chance that this distance function could create bias, first, the algorithm we will develop to cluster treats distance function as a replaceable part, and second, even if biased, we will see that our algorithm will do extremely well at clustering related data.

Definition 1.5.5. Let $\{C_1, \ldots, C_k\}$ be a partition of $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$. We associate to each cluster C_j a vector $\boldsymbol{z}_j$, which we call the **cluster representative**. Hence, for the k clusters $C_1, C_2, \ldots, C_k$ we have their k cluster representatives $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_k$. Note that while the cluster representatives could be among the $\boldsymbol{x}_i$'s we do not require this.

Next, we define the **cluster objective** of this partition, denoted J^{clust}, by the rule

$$J^{\text{clust}} = \frac{\|\boldsymbol{x}_1 - \boldsymbol{z}_{c_1}\|^2 + \|\boldsymbol{x}_2 - \boldsymbol{z}_{c_2}\|^2 + \ldots + \|\boldsymbol{x}_n - \boldsymbol{z}_{c_n}\|^2}{n}.$$

Note that if we associate the set X with the block vector $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$ and introduce the block vector $\boldsymbol{Z} = (\boldsymbol{z}_{c_1}, \boldsymbol{z}_{c_2}, \ldots, \boldsymbol{z}_{c_n})$, then $J^{\text{clust}} = \text{rms}(\boldsymbol{X} - \boldsymbol{Z})^2$.

The clustering objective of a partition offers a metric to measure the efficacy of the partition. Note that $J^{\text{clust}} \geq 0$ and $J^{\text{clust}} = 0$ if and only if $\boldsymbol{x}_i = \boldsymbol{z}_{c_i}$ for all i. This latter case would only happen if $C_{\boldsymbol{x}_i} = \{\boldsymbol{x}_i\}$, that is, all classes are a single element (perhaps X has repetitions in the vectors). Generally, a perfect score of 0 will be unlikely for a partition, but the closer the clustering objective is to zero measures how tight the clusters are, and hence how good a clustering we have. After all, the RMS will measure here the typical distance from the representative of a cluster, which we want to be small.

The clustering objective also can be used to determine the right number of k. Sometimes the data set will suggest or require a fixed number of k, but in the absence of which we can treat k as a variable. As we increase k, J^{clust} decreases, that is, $\lim_{k \to n} J^{\text{clust}} = 0$. But the more groups we allow the more difficult the eventual calculation will be and perhaps the less useful the clustering becomes (setting $k = n$ is fairly

pointless). So with k relatively small compared to n, if there is a significant jump from k to $k+1$, then we use the larger number of groups. But if after several increments of k we gain very little improvement of J^{clust}, then we should resolve to use the k after the last significant jump.

Example 1.5.6. Suppose $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$. Suppose that the number of clusters is fixed at k and the cluster representatives $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k$ are fixed, but the cluster assignment vector $\boldsymbol{c} = (c_1, \ldots, c_n)$ is allowed to vary. How do we cluster the data to minimize J^{clust}? We will treat this clustering problem now as a nearest neighbor problem. For each $\boldsymbol{x}_i$, we set

$$c_i = \min_{j=1}^{k} \|\boldsymbol{x}_i - \boldsymbol{z}_j\|,$$

that is, $\boldsymbol{x}_i$ is assigned to C_j if and only if $\boldsymbol{z}_j$ is the closest representative to $\boldsymbol{x}_i$. To see how this optimizes J^{clust} suppose that all $\boldsymbol{x}_i$ have been sorted except for $\boldsymbol{x}_n$. Suppose that we decide (for no good reason) to assign $\boldsymbol{x}_n$ to C_1. Then

$$\begin{aligned} J^{\text{clust}} &= \frac{\left(\|\boldsymbol{x}_1 - \boldsymbol{z}_{c_1}\|^2 + \|\boldsymbol{x}_2 - \boldsymbol{z}_{c_2}\|^2 + \ldots + \|\boldsymbol{x}_{n-1} - \boldsymbol{z}_{c_{n-1}}\|^2\right) + \|\boldsymbol{x}_n - \boldsymbol{z}_1\|^2}{n} \\ &= \frac{\|\boldsymbol{x}_1 - \boldsymbol{z}_{c_1}\|^2 + \|\boldsymbol{x}_2 - \boldsymbol{z}_{c_2}\|^2 + \ldots + \|\boldsymbol{x}_{n-1} - \boldsymbol{z}_{c_{n-1}}\|^2}{n} + \frac{1}{n}\|\boldsymbol{x}_n - \boldsymbol{z}_1\|^2 \end{aligned}$$

Note that if we can lower $\|\boldsymbol{x}_n - \boldsymbol{z}_1\|$, then we lower the entire J^{clust}. This is accomplished by replacing $\boldsymbol{z}_1$ with a representative $\boldsymbol{z}_j$ that is closer to $\boldsymbol{x}_i$ (if necessary). Of course, there is no closer representative than the nearest neighbor. Therefore, assigning a vector to a group based upon the nearest representative offers the optimal clustering for fixed representatives. ∎

Example 1.5.7. Suppose $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$. Suppose that the number of clusters is fixed at k and the cluster assignment vector $\boldsymbol{c} = (c_1, \ldots, c_n)$ is this time fixed, but the cluster representatives $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k$ are allowed to vary. How do we choose the cluster representatives to minimize J^{clust}? This time we will separate J^{clust} according to the cluster a vector belongs to. Suppose that we decide (for no good reason) to assign $\boldsymbol{z}_j = \boldsymbol{x}_i$ where c_i is the first appearance of j in $\boldsymbol{c}$. Then let $J_j = \sum_{\boldsymbol{x}_i \in C_j} \|\boldsymbol{x}_i - \boldsymbol{z}_j\|^2$. Hence,

$$J^{\text{clust}} = \frac{1}{n}(J_1 + J_2 + \ldots + J_k).$$

If we adjust any $\boldsymbol{z}_j$, then it will only affect J_j in the above sum. So we want to choose $\boldsymbol{z}_j$ to minimize the distance from $\boldsymbol{z}_j$ to $\boldsymbol{x}_i \in C_j$ on average. But this is accomplished by assigning $\boldsymbol{z}_j$ to the average (mean) vector $\overline{\boldsymbol{x}}$ of C_j (see Example 1.2.7), that is,

$$\boldsymbol{z}_j = \frac{1}{|C_j|} \sum_{\boldsymbol{x}_i \in C_j} \boldsymbol{x}_i$$

Therefore, assigning a cluster representative based upon the mean of each cluster offers the optimal clustering for fixed clusters. ∎

In the general clustering problem, neither the cluster assignments nor the cluster representatives are fixed, but from the two above examples we can still derive an effective clustering algorithm known as the k-means algorithm discussed in Section 1.6.

1.5.3 Applications of Clustering

There are two natural applications of clustering. First, separating objects based upon their groups can be helpful because the different groups may have different wants or needs that should be address. Patient clustering would be organizing patients based upon their medical data to more accurate diagnose and treat their diseases. Separating customers based upon their shopping history creates market segments that may have more targeted marketing strategies compared to the general audience. Regions with similar weather patterns can be clustered into the same weather zone, or companies with similar financial and business attributes can be grouped into the same financial sectors.

Also, after an optimal clustering has been achieved, this clustering also applies to new data that enters the data set. That is, we can sort new vectors based upon which group they are most similar to. For example, if a social media platform advertises to its members based upon their market segment, then when a new user creates an account then the platform will want to gather information about the user to decide which cluster they belong to in order to share advertisements that the cluster generally likes. If a patient has a medical history similar to other patients that suffer from the same disease, a doctor could suggest medical tests to determine if the patient also has the disease.

Example 1.5.8. Suppose a data set has been clustered and a new vector joins the data, maybe a new customer joins an online service. Instead of re-clustering the whole data set, we can simply adjoin the new vector to a pre-existing group based upon which cluster representative is nearest to the new vector. Then we can make decisions for this new vector based upon the expectations from its group. Of course, if there is a large influx of new vectors over time, eventually we will need to re-cluster the data. The advantage here is that we do not have to re-cluster immediately but we can still benefit from the clustering while waiting for the periodic re-clustering. ∎

Example 1.5.9. Similar to the previous example, imagine we have a clustered data set and a new vector not belonging to data. We want to identify which cluster the vector belongs to but we do not plan to retain this new vector. For example, suppose that our data set is a collection of images of hand-written characters, e.g., a, b, c, d, etc. Our software is hand writing recognition software. After scanning a hand-written document (or a user simply writes on a touch screen), the software views each image and has to identify which character was written. This is done by determining which cluster the new hand-written symbol belongs to. Thus, the hand-written scribble is replaced with the closest character in the software's alphabet. This is a very primitive notion of machine learning. The software was feed hundreds or thousands of handwritten character images and asked to cluster them. Then the clusters are given labels as ASCII characters. The representatives are retained (and the original data can be retained or discarded based upon if we want more learning in the future) and used to identify new images. ∎

Example 1.5.10. Again, suppose a data set is already clustered and a new vector is entered, but this time suppose that some of the entries in the vector are blank (missing information). We can still determine which cluster representative is closest to a vector (using all the non-blank entries, of course). After the cluster is decided, we can fill in the missing entries using the corresponding entries from the group representative, that is, we fill in the blanks with the typical entry for the cluster. For example, imagine a customer is listening to music on a streaming service. There is a song that the customer has never listened to before and hence there is no data on whether she likes the song or not. Because she belongs to a cluster, we can look at the cluster representative which has a 1 in the entry corresponding to this song (meaning that the typical customer in this segment likes the song). Hence, the app can recommend the song to the customer since we can expect that she will like the song too like the rest of her group. ∎

Python Programming

A very famous data set known as *Iris*, due to Ronald A. Fisher, is a collection of measurements of iris flowers. Fisher's goal was to use the measurements to classify the flowers into one of three types: setosa, versicolor, and virginica. He collected 150 samples, 50 of each type. For each flower, Fisher made four measurements (in cm): sepal length, sepal width, petal length, and petal width. The following code can be used to load and view Fisher's *Iris* data set via Python:

```
import numpy as np
#scikit learn is a library for many machine learning applications in Python
from sklearn.datasets import load_iris

iris_names = load_iris()['target_names']
iris_types = load_iris()['target']

iris_feature = load_iris()['feature_names']
iris_data = np.array(load_iris()['data'])

print(iris_names)
print(iris_types)
print(iris_feature)
print(iris_data)
```

Example 1.5.11. Consider the first vectors from the *Iris* data set of each type: setosa $\boldsymbol{z}_1 = [5.1, 3.5, 1.4, 0.2]$, versicolor $\boldsymbol{z}_2 = [7, 3.2, 4.7, 1.4]$, and virginica $\boldsymbol{z}_3 = [6.3, 3.3, 6, 2.5]$. Consider also the randomly selected vector $\boldsymbol{x} = [5.5, 2.3, 4, 1.3]$. Which type does $\boldsymbol{x}$ belong to? A simple method to classify this vector is to find its nearest neighbor among the three cluster representatives. Note

$$\|\boldsymbol{x} - \boldsymbol{z}_1\| = 3.09354166, \qquad \|\boldsymbol{x} - \boldsymbol{z}_2\| = 1.88679623, \qquad \|\boldsymbol{x} - \boldsymbol{z}_3\| = 2.66082694.$$

Thus, the nearest neighbor to $\boldsymbol{x}$ is $\boldsymbol{z}_2$, which suggests that most likely $\boldsymbol{x}$ is a versicolor iris.

The following code implements the nearest neighbor algorithm in Python:

```
import numpy as np
from sklearn.datasets import load_iris

iris_data = np.array(load_iris()['data'])

z = np.array([iris_data[0], iris_data[50], iris_data[100]])
k = len(z)
x = np.array([5.5,2.3,4.,1.3])

distances = np.zeros(k) # initialize distances

for i in range(k):
    distances[i] = np.linalg.norm(x-z[i])

print(np.argmin(distances))
print(distances)
#Solution is z[1], that is x is likely a versicolor
```

■

Exercises

(Go to Solutions)

For Exercises 1-6, consider a set of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ which is clustered into k-many classes $C_1, C_2, \ldots, C_k$ with representatives $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_k$, respectively.

♠ 1. If each $\boldsymbol{x}_i$ has only nonnegative entries, explain why each representative $\boldsymbol{z}_j$ likewise has only nonnegative entries.

♠ 2. If each $\boldsymbol{x}_i$ is a frequency vector, that is, each vector records the number of occurrences of certain categories, give an interpretation of an arbitrary entry of a representative $\boldsymbol{z}_j$.

♠ 3. If each $\boldsymbol{x}_i$ is a relative frequency vector, that is, each vector records the proportion of occurrences of certain categories, explain why each representative $\boldsymbol{z}_j$ is likewise a relative frequency vector.

♠ 4. If each $\boldsymbol{x}_i$ has only Boolean entries, give an interpretation of an arbitrary entry of a representative $\boldsymbol{z}_j$.

♠ 5. Suppose some vectors $\boldsymbol{x}_{i_1}, \boldsymbol{x}_{i_2}, \ldots, \boldsymbol{x}_{i_m}$ must be assigned to specific classes, e.g., say $\boldsymbol{x}_1$ must always be assigned to C_1. Provide a practical situation where this pre-assignment might arise.

♠ 6. Suppose $k = 2$. Then the function $f(\boldsymbol{x}) = \|\boldsymbol{x} - \boldsymbol{z}_2\|^2 - \|\boldsymbol{x} - \boldsymbol{z}_1\|^2$ can be used to determine which group $\boldsymbol{x}$ belongs to. If $f(\boldsymbol{x}) > 0$, then $\|\boldsymbol{x} - \boldsymbol{z}_2\|^2 - \|\boldsymbol{x} - \boldsymbol{z}_1\|^2 > 0$, which implies that $\|\boldsymbol{x} - \boldsymbol{z}_2\|^2 > \|\boldsymbol{x} - \boldsymbol{z}_1\|^2$ or more simply $\text{dist}(\boldsymbol{x}, \boldsymbol{z}_2) > \text{dist}(\boldsymbol{x}, \boldsymbol{z}_1)$. Thus, if $f(\boldsymbol{x}) > 0$, then $\boldsymbol{x}$ is closer to $\boldsymbol{z}_1$ and belongs to group 1. Similarly, if $f(\boldsymbol{x}) < 0$, then $\boldsymbol{x}$ belongs to group 2. Using properties of the inner product, expand and simplify the formula for f by finding a vector $\boldsymbol{a}$ and scalar b such that $f(\boldsymbol{x}) = \boldsymbol{a} \cdot \boldsymbol{x} + b$.

7. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/cluster.ipynb

"Do all the good you can, by all the means you can, in all the ways you can, in all the places you can, at all the times you can, to all the people you can, as long as ever you can." – John Wesley

1.6 The k-means Algorithm

In this section, we discuss the k-means Algorithm, which solves the clustering problem by alternating between fixed representatives and fixed clusters. It is a fast and efficient clustering algorithm, but it depends on the initial representatives to determine how effectively it solves the clustering objectives.

We saw in Section 1.5 how to optimize a clustering problem when the representatives are fixed or when the clusters are fixed. In the former problem, vectors are assigned to clusters via the nearest neighbor algorithm, and in the latter problem, the representatives are chosen as the mean of each cluster. The general problem of minimizing J^{clust} is much more difficult[i] because the clusters and their representatives inform each other. How do we choose the best partition when the representatives are in flex? How do we choose the best representative of a cluster when the clustering keeps changing?

The ***k*-means algorithm** is an algorithm for solving the clustering problem, proposed independently by Stuart Lloyd and Hugo Steinhaus in 1957. It iterates between fixed representatives and fixed clustering. Each step in the process improves the objective J^{clust} and the process terminates when the representatives stop changing[ii].

Algorithm 1.6.1 (k-means Algorithm)**.**

Given *a list of n vectors $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ and an initial list of k cluster representatives $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_k$. Repeat the following two steps until sufficient convergence of J^{clust} is obtained.*

1. Partition the vectors into k clusters. *For each vector $i = 1, 2, \ldots, n$, assign c_i to be the index of the nearest representative $\boldsymbol{z}_j$ to $\boldsymbol{x}_i$.*
2. Update representatives. *For each cluster $j = 1, 2, \ldots, k$, assign $\boldsymbol{z}_j$ to be the mean of the vectors in C_j.*

Returns *the representatives $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_k$, the cluster assignment $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$, and the final clustering objective J^{clust}.*

To implement the k-means algorithm, we utilize four functions: `clustering_objective`, `update_clustering`, `update_reps`, and `Kmeans`.[iii]

The `clustering_objective` function computes J^{clust} for the current representatives $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k$ and current cluster assignment $\boldsymbol{c} = (c_1, \ldots, c_n)$. It takes three parameters: `data` which represents the vectors $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$, `clustering` which represents $\boldsymbol{c}$, and `reps` which represents $[\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k]$. It returns J^{clust}.

[i] It turns out that finding an *optimal solution* for the general problem will be very difficult, that is, it will be computationally complex, relative to the data set. So difficult that for even a reasonably sized data set, the time required to find the optimal clustering (and its corresponding representatives) becomes too expensive to be worth it. Do we just give up then? In situations like this, we often abandon an *optimal algorithm*, that is, an algorithm which guarantees the best solution. One such optimal algorithm would be *Brute Force*, that is, check every possible partition. The number of partitions for even a small set is quite large, and hence, like usual, the Brute Force Algorithm is impracticable to use. Our implementation of the Nearest Neighbor Algorithm in Section 1.5 is a Brute Force process as it checks all neighbors. Without any ordering to the neighbors, we have no better search for the nearest neighbor. Fortunately, the Nearest Neighbor Search is no more expensive than the number of neighbors, that is, the complexity is proportional to the data set and not too expensive.

For the sake of clustering, where Brute Force would be very expensive, we instead use a *heuristic algorithm*, that is, a process that produces a *sub-optimal solution*, a solution with a small error when compared to the optimal solution, relatively quickly. The amazing speed in solving the problem is so valuable that we are even okay with the small error attached to it. The k-means algorithm is a heurisitc algorithm.

[ii] If after a round of k-means the representatives $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_k$ do not alter, then the following round of nearest neighbor will not change the clustering, that is, all vectors $\boldsymbol{x}_i$ is remain nearest to the previous representative $\boldsymbol{z}_j$. The next round of mean calculations will result in the same representatives again as no vector changed its cluster.)

[iii] The code for these four procedures can be found here.

```
def clustering_objective(data, clustering, reps):
    J_obj = 0
    for i in range(len(data)):
        J_obj += np.linalg.norm(data[i] - reps[int(clustering[i])-1])**2
    return J_obj/len(data)
```

The `update_clustering` function computes the optimal clustering assignment $\boldsymbol{c}$, in order to overwrite the previous assignment vector, based upon the current representatives $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k$. It takes two parameters: `data` which represents the vectors $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$ and `reps` which represents $[\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k]$. It returns an updated clustering $\boldsymbol{c}$.

```
def update_cluster(data,reps):
    new_clustering = np.zeros(len(data))
    for i in range(len(data)):
        distance = np.zeros(len(reps))
        for j in range(len(reps)):
            distance[j] = np.linalg.norm(data[i] - reps[j])
        new_clustering[i] = np.argmin(distance)+1
    return new_clustering
```

The `update_reps` function computes the means of the clusters $C_1, C_2, \ldots, C_k$, in order to overwrite the previous representatives, based upon the current clustering $\boldsymbol{c}$. It takes two parameters: `data` which represents the vectors $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$ and `clustering` which represents $\boldsymbol{c}$. It returns $[\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_k]$.

```
def update_reps(data, clustering):
    new_reps = [];
    for j in range(int(max(clustering))):
        sum = np.zeros(len(data[0]))
        count = 0
        for i in range(len(data)):
            if clustering[i] == (j+1):
                sum = sum+data[i]
                count += 1
        if count > 0:
            new_reps.append(sum/count)
    return np.array(new_reps)
```

Finally, the `Kmeans` function implements the k-means algorithm from above using recursive calls to the other three functions. It takes two parameters:[iv] `data` which represents the vectors $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$ and `reps` which represents $[\boldsymbol{z}_1, \ldots, \boldsymbol{z}_j]$. It returns the final versions of each $[\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k]$, $\boldsymbol{c}$, and J^{clust}.

```
def Kmeans_alg(data, reps, show_steps=False):
    J_obj = []
    clustering = update_cluster(data, reps)
    J_obj.append(clustering_objective(data, clustering, reps))

    if show_steps:
        print("Iteration", len(J_obj)-1, ":\n (Update Cluster)")
        print("c = ", clustering)
        print(" Z = ", reps)
        print("J^clust = ", J_obj[-1], "\n")

    Stop = False
    while not(Stop):
        new_reps = update_reps(data, clustering)
        if show_steps:
            print("Iteration", len(J_obj), ":\n (Update Reps)")
            print("c = ", clustering)
            print(" Z = ", reps)
            print("J^clust = ", clustering_objective(data, clustering, new_reps), "\n")
        clustering = update_cluster(data, new_reps)
        J_obj.append(clustering_objective(data, clustering, new_reps))
        if show_steps:
            print("(Update Cluster)")
            print("c = ", clustering)
            print(" Z = ", reps)
```

[iv] `show_steps` is an optional, third parameter, which by default is `False`, which if toggled to `True` would show the steps of the algorithm.

```
            print("J^clust = ", J_obj[-1], "\n")
        if len(new_reps) == len(reps):
            if np.linalg.norm(np.array(new_reps) - np.array(reps)) < 1e-6:
                Stop = True
            else:
                reps = new_reps
        else:
            reps = new_reps
    return new_reps, clustering, J_obj, len(J_obj)-1
```

Example 1.6.2. Consider the thirteen 3-vectors

$$\boldsymbol{X} = [\,[1,2,3],[4,5,6],[1,2,0],[0,0,3],[3,2,1],[2,2,0],[0,5,0],[2,3,1],[6,6,4],[9,3,2],[2,3,4],[7,5,0],[2,7,8]\,].$$

Using the initial representatives

$$\boldsymbol{Z} = [\,[1,0,0],\ [0,1,0],\ [0,0,1]\,],$$

the k-means algorithm terminates after 5 steps with the clustering

$$\{\,\{\boldsymbol{x}_9,\boldsymbol{x}_{10},\boldsymbol{x}_{12}\},\quad \{\boldsymbol{x}_1,\boldsymbol{x}_3,\boldsymbol{x}_4,\boldsymbol{x}_5,\boldsymbol{x}_6,\boldsymbol{x}_7,\boldsymbol{x}_8\},\quad \{\boldsymbol{x}_2,\boldsymbol{x}_{11},\boldsymbol{x}_{13}\}\,\},$$

that is,

$$\boldsymbol{c} = [2,1,2,2,2,2,2,2,1,1,1,3],$$

with an objection of $J^{\text{clust}} = 5.208791208791208$ and representatives

$$\boldsymbol{Z} = [\,[7.33333333,4.66666667,2],\ [1.28571429,2.28571429,1.14285714],\ [2.66666667,5,6]\,].$$

Using instead the initial representatives

$$\boldsymbol{Z} = [\,[0.5,0.5,0.5],\ [7,1,0],\ [3,-1,4]\,],$$

the k-means algorithm terminates after 2 steps with the clustering

$$\{\,\{\boldsymbol{x}_1,\boldsymbol{x}_3,\boldsymbol{x}_4,\boldsymbol{x}_5,\boldsymbol{x}_6,\boldsymbol{x}_7,\boldsymbol{x}_8\},\quad \{\boldsymbol{x}_9,\boldsymbol{x}_{10},\boldsymbol{x}_{12}\},\quad \{\boldsymbol{x}_2,\boldsymbol{x}_{11},\boldsymbol{x}_{13}\}\,\},$$

that is,

$$\boldsymbol{c} = [1,3,1,1,1,1,1,1,2,2,3,2,3],$$

with an objection of $J^{\text{clust}} = 5.208791208791208$ and representatives

$$\boldsymbol{Z} = [\,[1.28571429,2.28571429,1.14285714],\ [7.33333333,4.66666667,2],\ [2.66666667,5,6]\,].$$

Notice that this partition just swapped the roles of C_1 and C_2, although it found it much faster.

Finally, using instead the initial representatives

$$\boldsymbol{Z} = [\,[1,0,0],\ [1,1,0],\ [1,1,1]\,],$$

the k-means algorithm terminates after 6 steps with the clustering

$$\{\,\{\boldsymbol{x}_1,\boldsymbol{x}_3,\boldsymbol{x}_4,\boldsymbol{x}_5,\boldsymbol{x}_6,\boldsymbol{x}_7,\boldsymbol{x}_8,\boldsymbol{x}_{11}\},\quad \{\boldsymbol{x}_2,\boldsymbol{x}_9\boldsymbol{x}_{10},\boldsymbol{x}_{12},\boldsymbol{x}_{13}\}\,\},$$

that is,

$$\boldsymbol{c} = [1,3,1,1,1,1,1,1,2,2,3,2,3],$$

with an objection of $J^{\text{clust}} = 9.057692307692308$ and representatives

$$\boldsymbol{Z} = [\,[1.375,2.375,1.5],\ [5.6,5.2,4]\,].$$

This time a worse objective is obtained and the number of clusters dropped down to two. ∎

We make a few comments about the above implementation of the k-means algorithm. First, if it so happens that a vector $\boldsymbol{x}_i$ is equidistant to more than one representative $\boldsymbol{z}_j$, then c_i will be assigned the smallest index of these representatives.

Second, the value k is initially determined by the number of representatives declared, but it is possible that through calculations that one or more clusters become empty. In such a situation, the empty clusters are removed, k is decreased, and cluster assignments are shifted accordingly.

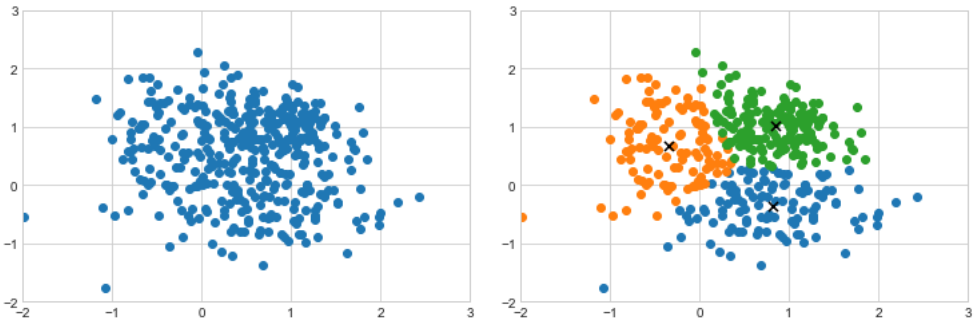
Third, if in two successive iterations the cluster representatives do not change, then J^{clust} will not improve. The implementation is set to terminate when J^{clust} does not improve from its previous calculation by more than 10^{-6} (of course, this sensitivity can be adjusted as appropriate).

Fourth, the number of clusters k affects the score J^{clust}. In particular, $\lim_{k \to n} J^{\text{clust}} = 0$. So the more clusters the smaller the J^{clust}. But having "too many" clusters can also minimize the usefulness of the partition. Hence, it is wise to keep k small relative to n and to experiment with the size of k and see which k works well. For example, if the value of J^{clust} with $k = 7$ is quite a bit smaller than the values of J^{clust} for $k = 2 \ldots, 6$, and not much larger than the values of J^{clust} for $k = 8, 9; \ldots$, we could reasonably choose $k = 7$, and conclude that our vectors partition nicely into 7 groups.

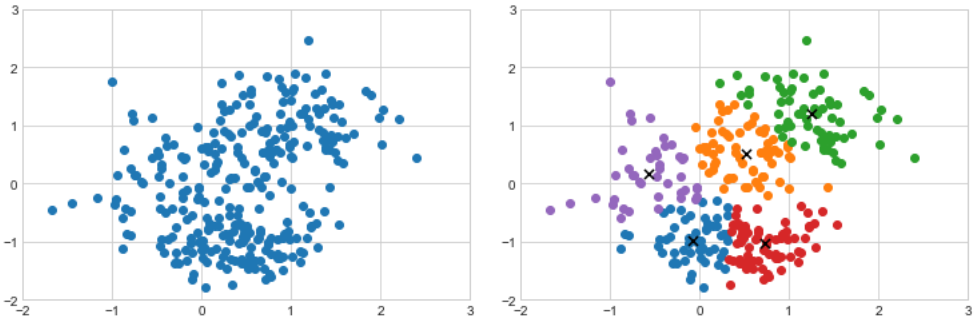
Fifth, the final J^{clust} is determined by the initial $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k$. Based upon this choice, the clustering objective can be improved (or worsened). Hence, it is efficient to provide a protocol for determining "good" initial representatives. Unfortunately, we will not discuss any such protocol and will instead rely on randomly generated representatives to begin the calculation and will repeat the calculation multiple times to attempt to improve J^{clust}.

Finally, to avoid an infinite loop in the k-means algorithm, the sequence J^{clust} must eventually converge toward some limit. Now, each step in the algorithm improves J^{clust}, meaning that J^{clust} is decreasing over time. Since it is always decreasing, we say this sequence is *monotonic*. As $J^{\text{clust}} \geq 0$, we say it is *bounded*. The Monotone Convergence Theorem from Calculus, written below, tells us this sequence is then convergent, which to us means that the algorithm will eventually terminate no matter how tight a bound we set.

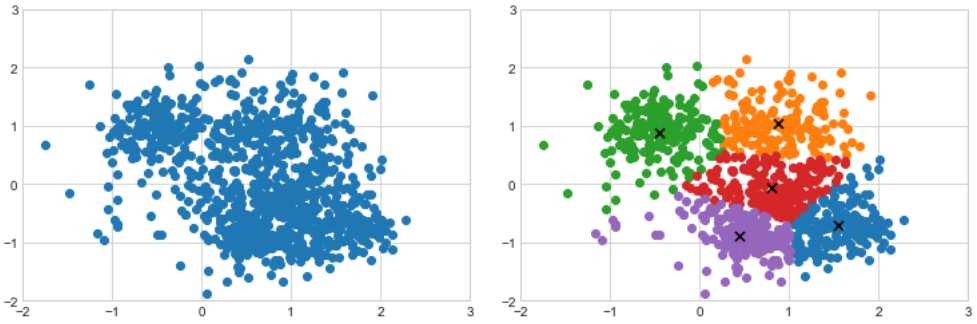
Example 1.6.3. Consider a set of $n = 300$ randomly generated vectors in the plane. Using k-means, the set is sorted into $k = 3$ subsets.



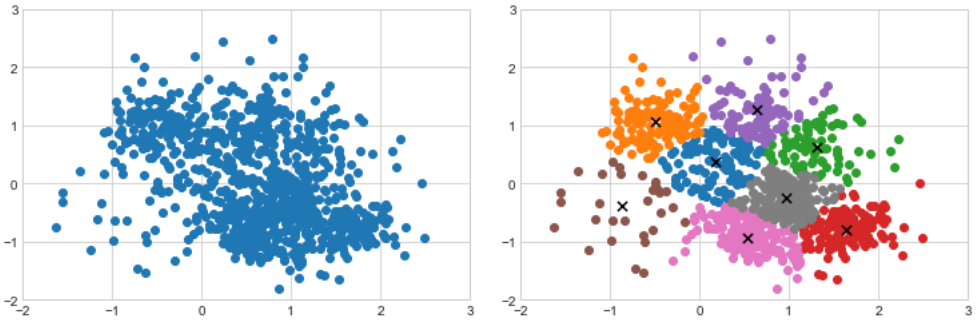
For another set of $n = 300$ random vectors, the vectors are placed into $k = 5$ subsets.



For another set of $n = 1000$ random vectors, the vectors are placed into 5 subsets ($k = 5$).



Finally, for another set of $n = 1000$ random vectors, the vectors are placed into $k = 8$ subsets.



■

Python Programming

In Python, the k-means algorithm is already pre-loaded and can be executed using the `KMeans` library from the `sklearn.cluster` package. Scikit Learn is a library in Python designed for many machine learning applications.

```
import numpy as np
from sklearn.cluster import KMeans
```

Example 1.6.4. The following code was used to create the random vectors and their clustering in Example 1.6.3 in Python:

```
#matplotlib, as the name suggests, is a library designed for plotting mathematical
    #functions. From time to time, we will need to use it to create graphics.
    #Please see the appendix on matplotlib for further details.
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-whitegrid')

import numpy as np
from sklearn.cluster import KMeans

k = 3     #the number of clusters to be formed in the partition

plt.ion() #interactive mode will be on
          #figures will automatically be shown
#X is a set of 1000 2-vectors randomly generated using methods within
#the random sublibrary of NumPy.
X = np.concatenate([[0.75*np.random.randn(2) for i in range(100)],
                    [[0,1] + 0.5*np.random.randn(2) for i in range(100)],
                    [[1,1] + 0.4* np.random.randn(2) for i in range(100)],
                    [[1,-1] + 0.3* np.random.randn(2) for i in range(100)],
                    [[-0.5,1] + 0.25* np.random.randn(2) for i in range(100)],
                    [[0.5,-1] + 0.1* np.random.randn(2) for i in range(100)],
                    [[0.5,-0.5] + 0.3* np.random.randn(2) for i in range(100)],
                    [[1,-0.25] + 0.25* np.random.randn(2) for i in range(100)],
                    [[1.75,-0.75] + 0.2* np.random.randn(2) for i in range(100)],
                    [[1,0] + 0.5* np.random.randn(2) for i in range(100)]])

#The following code formats the scatterplot of X w/o clustering.
x = np.linspace(-2, 3, 10)  #Used to create an evenly spaced sequence in the
                            #interval (-2 to 3). There will be (10) spaces.
plt.xlim(-2,3)              #Sets the x-axis boundaries for the graph
plt.ylim(-2,3)              #Sets the y-axis boundaries for the graph
plt.scatter( X[:,0],X[:,1]) #Plots all the vectors in X
plt.show()                  #Displays all open figures

#The k-means algorithm takes the number of clusters k and a random state,
    #which randomizes the initial representatives
kmeans = KMeans(n_clusters=k, random_state=0).fit(X)
labels = kmeans.labels_        #the cluster assignment vector
reps = kmeans.cluster_centers_ #the cluster representatives
J_clust = kmeans.inertia_      #the cluster objective
print(J_clust)

#formatting and creation of second scatterplot
#grps is the creation of the actual clustering of vectors for the purposes of coloring
    #the scatterplot.
grps = [[X[i,:] for i in range(len(X)) if labels[i]==j] for j in range(k)]
for i in range(k):
    plt.scatter([c[0] for c in grps[i]],[c[1] for c in grps[i]])
plt.scatter([z[0] for z in reps],[z[1] for z in reps],c='black',marker='x',s=50)
plt.xlim(-2,3)
plt.ylim(-2,3)
plt.show()
```

■

For educational purposes, students are expected to utilize the implementation of k-means algorithm as outlined in this section, which can be found here.

Exercises

(Go to Solutions)

For Exercises 1-4, perform j iterations of the k-means algorithm on the data set

$$X = \left\{ \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -2 \\ 4 \\ 3 \end{bmatrix}, \begin{bmatrix} 5 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix}, \begin{bmatrix} 5 \\ -5 \\ 0 \end{bmatrix}, \begin{bmatrix} 3 \\ -1 \\ 2 \end{bmatrix}, \begin{bmatrix} 3 \\ 4 \\ -5 \end{bmatrix} \right\}$$

for the given number of clusters k and initial representations $Z = \{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k\}$. Report the final representatives Z and cluster assignment vector $\boldsymbol{c}$.

1. $j = 1$, $k = 3$
   $Z = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$

♠ 2. $j = 2$, $k = 3$
   $Z = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$

♠ 3. $j = 3$, $k = 3$
   $Z = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$

4. $j = 4$, $k = 3$
   $Z = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$

For Exercises 5-5, consider the same data set from Example 1.6.2. Find a clustering for which $J^{\text{clust}} < \varepsilon$ using the given ε and k. Report your initial representatives, your final representatives, your final clustering assignment, and the final clustering objective. *A calculator or computer is recommended.*

5. $\varepsilon = 5.208$, $k = 4$

For Exercises 6-13, consider the following data set[v] of $n = 100$ vectors. Compute three different clusterings for the given k using three different randomized initializations. To report your clustering include a colored graphic illustrating the clustering (as demonstrated in the Python subsection of Section 1.6), list the cluster representatives, and J^{clust} for each of these three clusterings. Explain which clustering was the best of the three. *A calculator or computer is required.*

6. $k = 2$
7. $k = 3$
8. $k = 4$
9. $k = 5$
10. $k = 10$
11. $k = 20$
12. $k = 25$
13. $k = 100$

14. Complete all of the Python-based exercises found at:
    https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/kmeans.ipynb

[v]The data set can be found here: https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/kmeans.ipynb

Python Projects

1. For those unaware, *Major League Mathletics* (MLM) is the professional league of competitive mathematics here in the US and has major world followings (recent ratings showed that the MLM championship game surpassed the ratings of the NFL Super Bowl, FIFA World Cup, and the Summer Olympics combined). Each season, a most valuable player (MVP) trophy is awarded to the single mathlete that was ranked the very highest for that year and is considered the league's highest honor. Votes for MVP are cast by the 200 league coaches and each coach cast a vote for his/her top five choices, ranked 1st, 2nd, 3rd, 4th, and 5th. Based upon their ranked votes, mathletes receive points according to the following table.

1st Place	2nd Place	3rd Place	4th Place	5th Place
5 pts	4 pts	3pts	2 pts	1 pt

   For example, if Michael "Dean" Andrews from the Cedar City Calc's received 2 second place votes, 1 third place vote, and 4 fifth place votes, he would receive $0(5) + 2(4) + 1(3) + 0(2) + 4(1) = 15$ points in the MVP competition (sadly, Dean did not win). Only mathletes with at least one 1st place ranking are eligible to win the award, and they are ranked by their scores determined by the coaches' votes. The highest score wins the award. Python code is started for you at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/MLM.py

   (a) The following are the 15 mathletes that received a first place vote from last season (listed in no particular order):
   - Pete Weyl from the Duschesne Determinants
   - Karla Gauss from the Tremonton Trigs
   - Hermie Wielandt from the St. George Scalars
   - Artie Cayley from the Logan Logarithms
   - Emmy Noether from the SLC SVDs
   - Evie Galois from the Ephraim Exponentials
   - Rowan "Quaternion" Hamilton from the Parowan Permutations
   - Dave Hilbert from the Ogden Orthogonals
   - Sophie Lie from the Manti Matrices
   - Jorge Frobenius from the Vernal Vectors
   - Izzy Schur from the Alpine Algebros
   - Stevie Humphries from the Lindon Linears
   - Mikey Muzychuk from the Tooele Tangents
   - Manny Leung from the Deseret Derivatives
   - Ollie Tamaschke from the Enterprise Inner Products

   After all the votes were tabulated, each of these 15 players is assigned a 5-vector $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5)$ where x_i is that player's number of ith place votes. Compute a complete ranking of these 15 players based upon their MVP scores and determine who won the award this season.

   (b) As discussed above, the score is computed using the vector $(5, 4, 3, 2, 1)$. Many in the sport have advocated that the scoring vector should be adjusted to $(7, 4, 3, 1, 1)$ to more emphasize coach's first place votes and less emphasize their later votes. If this new vector were adopted, compute a complete ranking of these 15 players and compare this ranking to the official ranking you computed above. What significant differences do you observe? Why did they change?

   (c) Create two other point vectors to result in two other different MVP winners (not including the two winners from the two previous vectors). The only requirement for point vector $\boldsymbol{p} = (p_1, p_2, p_3, p_4, p_5)$ is that $p_1 \geq p_2 \geq p_3 \geq p_4 \geq p_5 \geq 0$. Is there a point vector for which Manny Leung does not get last place?

2. The linked data below is a set of 40 25-vectors. Determine which two vectors are the farthest apart and which two vectors are the closest together. Report these maximum and minimum distances and

the pairs of vectors which obtain them. The maximum distance between two vectors in a set is called the **diameter** of the set. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/hotcold.py

3. Recall the Iris flower data consisting of 150 flowers of three type: setosa, versicolor, virginica. In fact, the data set is organized so that the first 50 profiles are setosa, the next 50 are versicolor, and the last 50 are virginica. The Iris data set can be found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/code/iris.py

   (a) Compute a clustering of these 150 iris flowers using the k-means Algorithm where $k = 3$. Before doing so, make a prediction on how the flowers will cluster. Justify your prediction. The k-means algorithm can be found at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/code/kmeansalg.py

   (b) Analyze the clustering objective from the clustering you just obtained to quantify the effectiveness of the clustering. Is it a good or bad clustering? Justify your answer. Now, compare the cluster assignment vector you just obtained with the `iris_types` vectors. How many differences are there between this and your cluster assignment vector? How does this compare to your original prediction? What might account for the errors, if any?

4. The linked data set below contains 1000 Boolean 100-vectors:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/code/kmeansalg.py

   (a) Cluster these vectors into 15 groups using k-means. Note that all the entries in these vectors are 0's and 1's. The k-means algorithm can be found at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/code/kmeansalg.py

   (b) Next, we have received a 95-vector that is a corrupted data profile where the last 5 entries are missing. Using the clustering just determined, predict what the missing entries should be. Note that all the entries in these vectors are 0's and 1's. The corrupted data profile can be found at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/clustering_corrupt.py

5. Even with computers, it takes time to analyze data. The k-means clustering algorithm, while quite fast (and somewhat biased), will likewise slow down when the data set is sufficiently large. Because of its biased-nature, several iterations of the algorithm with different randomized initiations is also necessary, which slows it down more. This problem is exacerbated when the data comes with an expiration date, for example, trends in social media. What happens when the lifecycle of a data set is comparable to the duration of an algorithm? What happens if the algorithm terminates at the same time (or even after) the data is obsolete? This project will explore how we can modify the k-means Algorithm to accommodate such highly evolutionary data, a method which could be implement perhaps in other iterative methods. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/movingtarget.py

   (a) We will consider a data set X is a time series of ten data sets, where `X[0]` is the data at time $t = 0$, `X[1]` is the data at time $t = 1$, `X[2]` is the data at time $t = 2$, etc. Each `X[i]` is a set of 100 2-vectors (so we can graph the individuals vectors as points in a scatterplot). Execute the Python code above using the `Kmeans_alg` function, which is the standard k-means Algorithm, to solve the clustering problem for each `X[i]`. Note that this will cluster each data set into 7 groups ($k = 7$ and initial representatives will be `X[0,:7]`) and will output to the screen the sequence of J^{clust} for each `X[i]`, as well the number of steps in each case and a scatterplot of the clustered data. The code will do this 10 times. Which of the 10 `X[i]` had the best clustering and why?

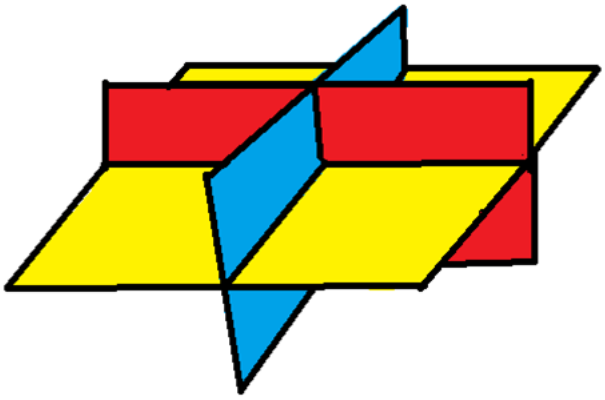
   (b) Create a new function `Kmeans_alg_temporal(data, reps)`. This will be a modification of `Kmeans_alg`, but instead of taking data as a set of vectors (a `2-d np.array`) data will now be a time series of sets of vectors (a `3-d np.array`). `reps` will still be `X[0,:7]`, hence $k = 7$. It will go through the k-means Algorithm as usual, but now the data set will also increment through the time series for each iteration of the algorithm (as in the data evolves at the same rate as the duration of each iteration of the algorithm). In other words, the first iteration will use `X[0]`, the second iteration will use `X[1]`, the third iteration will use `X[2]`, etc. The modified algorithm will stop after 10 iterations since that is when the time series ends. In particular, the cluster representatives found

on `X[i]` will be used to update the clustering on `X[i+1]`. Execute this modified code to solve the clustering problem for the time series `X`. Again output to screen 10 scatterplots for each step of the algorithm and the sequence of J^{clust} for `X` (there will only be 10 J^{clust} this time instead of 100+). How does the final J^{clust} of `X` clustered by `Kmean_alg_temporal` compare to the final J^{clust} of `X[9]` clustered by `Kmeans_alg` compare? Which one was found fastest? Conjecture why this is possible.

Chapter 2

Linear Systems

Systems of linear equations, also known as linear systems, are most students first exposure to linear algebra, with solutions often found by some combination of the Methods of Substitution or Elimination (see Appendix B). We will learn that linear systems are at the heart of all linear algebraic problems, as nearly all problems about vectors and matrices can be rebranded as a linear system (or vice versa). We will learn effecient algorithms to solve linear systems and apply them to ubiquitous applications.



"Education is that whole system of human training within and without the school house walls, which molds and develops men." – W. E. B. Du Bois

2.1 Linear Systems

In this section, we introduce the study of linear systems motivated by geometry. Thus, all numbers in this section will be real numbers. We introduce the notion of a system of linear equations and the nature of their solution sets. We discuss when a linear system is consistent/inconsistent, independent/dependent, underdetermined/overdetermined, and homogeneous/nonhomogeneous. We discuss the difference between free and dependent variables in a linear system.

2.1.1 Linear Systems

Definition 2.1.1. A **system of linear equations** (or a **linear system**) is a set of equations with common variables of the following form:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b,$$

where b and the **coefficients** $a_1, \ldots, a_n$ are fixed numbers.

A **solution** to a system is an assignment to each variable such that each equation is satisfied. The **solution set** is the set of all solutions to a system of equations. Two systems of equations are **equivalent** if they have the same solution set.

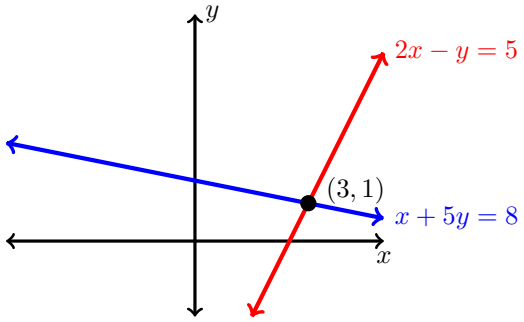
A linear equation of two real variables (typically x, y or x_1, x_2) geometrically forms a *line* in the plane. Likewise, a linear equation of three real variables (typically x, y, z or x_1, x_2, x_3) geometrically forms a *plane* in 3-space. Although much harder to visualize, a linear equation in four variables (typically x, y, z, w or x, y, z, t or x_1, x_2, x_3, x_4) geometrically forms a *hyperplane* in 4-space. Higher dimensional analogues likewise exist. As such, emphasis in this section will be placed on linear systems with two or three real variables.

Example 2.1.2. Test that $(3, 1)$ is a solution to the system

$$\begin{cases} x + 5y = 8 \\ 2x - y = 5. \end{cases}$$

Notice that $3 + 5(1) = 3 + 5 = 8$ and $2(3) - 1 = 6 - 1 = 5$. Thus, $(3, 1)$ is a solution to the above system.

Geometrically, the solution to the previous system is an intersection between two lines, as depicted to the right.



■

Example 2.1.3. Consider the linear system with three variables and three equations:

$$\begin{cases} x + 2y + 3z = 20 \\ -2x + y = -1 \\ -3x - 6y + 5z = -4. \end{cases}$$

We can check that $(2, 3, 4)$ is a solution of this system. Note that $(2) + 2(3) + 3(4) = 2 + 8 + 12 = 20$ ✓, $-2(2) + (3) = -4 + 3 = -1$ ✓, and $-3(2) - 6(3) + 5(4) = -6 - 18 + 20 = -4$ ✓, which shows that $(2, 3, 4)$ is in fact a solution of the system. Try as one might, another solution to this system CANNOT be found, that is, $(2, 3, 4)$ is the unique solution to this system of equations. ■

Example 2.1.4. Consider the linear system with three variables and three equations:

$$\begin{cases} 2x - y + z = 11 \\ x + 3y - 10z = 2 \\ -3x + 2y - 3z = -17. \end{cases}$$

It is simple to check that $(5, -1, 0)$ is a solution to this system. Likewise, we can also show that $(6, 2, 1)$ is a solution to this same system. Thus, it is possible to have multiple solutions to a linear solutions. ∎

2.1.2 Number of Solutions to a Linear System

Because of Example 2.1.4, it is worth investigating how many solutions a linear system may have. The beginning of our investigation is the following definitions.

Definition 2.1.5. A system of equations is **consistent** if it has a solution. Otherwise, we call the system **inconsistent**.

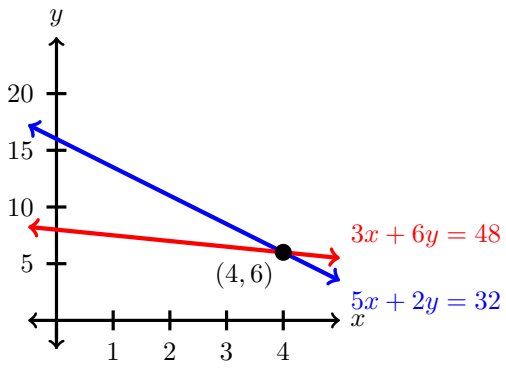
If a consistent linear system has a unique solution, we call this the **independent case**. If a consistent linear system has multiple solutions, we call this the **dependent case**.

Like we did in Example 2.1.2, we will investigate linear systems with two variables and their corresponding geometry.

Example 2.1.6. Solve each system of equations.

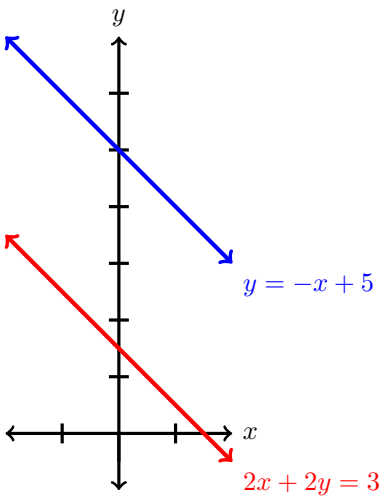
(a) $\begin{cases} 5x + 2y = 32 \\ 3x + 6y = 48 \end{cases}$

After graphing the two lines, as depicted to the right, we see that the two lines intersect at a unique point. This point of intersection, $\boxed{(4, 6)}$, means the system has a unique solution. Therefore, the system is consistent and independent.



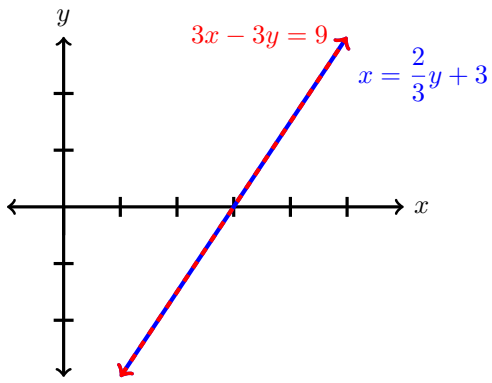
(b) $\begin{cases} y = -x + 5 \\ 2x + 2y = 3 \end{cases}$

After graphing the two lines, as depicted to the right, we see that the two lines are parallel. Thus, they do NOT ever intersect. Therefore, the system is $\boxed{\text{inconsisitent}}$, that is, there is no solution to the linear system.



(c) $\begin{cases} x &= \frac{2}{3}y + 3 \\ 3x - 3y &= 9 \end{cases}$

After graphing the two lines, as depicted to the right, we see that the two lines are the same line. Thus, every point on one line is also a point on the other line. Therefore, the system is dependent, that is, there are infinitely many solutions to the linear system. In particular, the system is consistent.

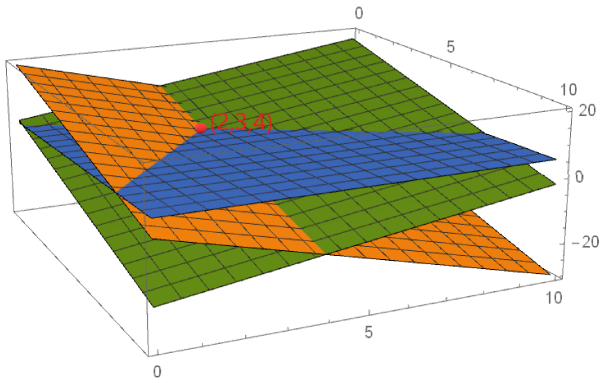


■

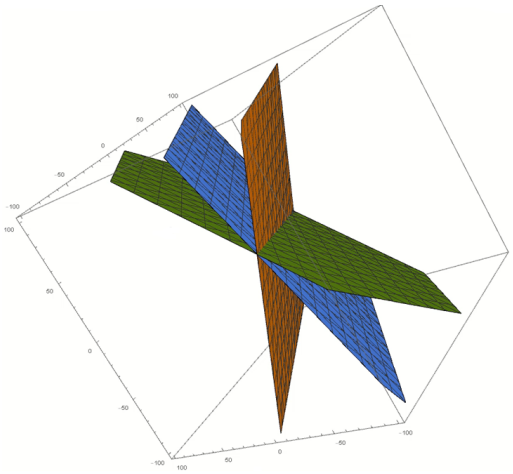
In general, any two lines in the real plane intersect at 0, 1, or infinitely many points (this last case occurs if the lines overlap). This holds also for higher dimensional systems of linear equations, that is, for any system of linear real equations the number of solutions is 0, 1, or ∞ and the solution set of a linear system will fall under one of these three types.

2.1.3 3-Dimensional and Homogeneous Linear Systems

These three possibilities are also the case for intersections of planes in 3-dimensional space. The *independent, consistent* case occurs when all planes in the system intersect at a unique point. This was the case for Example 2.1.3. Visually, this can be seen as three planes intersecting at a unique point, such as the three planes involved in Example 2.1.3, illustrated below:



The *inconsistent* case occurs when not all the planes simultaneously intersect at a common point, perhaps because some subset of planes are parallel with one another. Finally, the *dependent, consistent* case occurs when the collection of planes overlay at more than one point, e.g. three planes in 3-space intersect along a common line, as illustrated below:



This was the case for Example 2.1.4. In fact, any point of the form

$$(z + 5, 3z - 1, z), \tag{2.1.7}$$

where the variable z can be freely assigned to any real number, is a solution to this linear system. This is called the **general solution** to the system. In this example, z is a **free variable**, that is, a variable which can be assigned any value. The other two variables, x and y, are called **dependent variables**, because their assignment is dependent upon the assignments of the free variables corresponding to some linear relationship. Thus, the linear system in Example 2.1.4 has one free variable and two dependent variables. The two solutions listed in Example 2.1.4 arose exactly from setting $z = 0$ and $z = 1$, respectively. In general, the dependent case occurs exactly when the linear system has at least one free variable.

The following properties will be proven in the homework.

Proposition 2.1.8. *Suppose that a consistent linear system has m equations and n variables.*

(iii) If $n > m$, then the system has at least $n - m$ free variables which implies that it has multiple solutions.

(iv) If $m > n$, then the system has at least $m - n$ many equations which could be removed without modifying the solution set.

The situation where $n > m$ is called an **underdetermined system** because there are not enough equations to guarantee a unique solution. The situation where $m > n$ is called an **overdetermined system** because there might be too many equations to guarantee consistency.

Definition 2.1.9. A system of linear equations is called **homogeneous** if each equation is of the form:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = 0.$$

Otherwise, the system is called **nonhomogeneous**.

Example 2.1.10. The below 3×3 linear system is homogeneous since the right hand side of each equation is 0:

$$\begin{cases} x + 3y + 5z = 0 \\ 2x + 4y + 6z = 0 \\ 4x + 2y \quad\;\; = 0 \end{cases}$$

On the other hand, the below linear system appears to be homogeneous at first, but is really non-homogeneous:

$$\begin{cases} x + 3y - 2z + 4 = 0 \\ 4x + 2y + 3z - 5 = 0 \\ -x + 5y - 3z - 2 = 0 \end{cases}$$

The issue here is that the linear system is not in *standard form*, that is, all the variables are located on the left-hand sides of the equations, in descending order, and the constants are all on the right-hand sides of the equations. In standard form, the system would appear as:

$$\begin{cases} x + 3y - 2z = -4 \\ 4x + 2y + 3z = 5 \\ -x + 5y - 3z = 20 \end{cases}$$

In standard form, it is much more clear that the above linear system is non-homogeneous. ∎

Proposition 2.1.11. *A homogeneous system of equations is always consistent.*

Python Programming

Given the linear equation

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b,$$

we can naturally view the sequence of coefficients on the LHS of the equation as a vector $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$. If $\boldsymbol{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$, then the inner product $\boldsymbol{a} \cdot \boldsymbol{x}$ is the evaluation of the LHS. Recall from Section 1.3 that this is computed in `NumPy` as `a@x`. Hence, we can check if a vector $\boldsymbol{x}$ is a solution to the above linear equation by the Boolean `a@x == b`.[i]

Iterating this method, we can similar check if $\boldsymbol{x}$ is a solution to a linear system. Let A be an `2d-np.array` where `A[i]` is the vector of coefficient for the ith equation in the linear system and let $\boldsymbol{b}$ be the vector where `b[i]` is the RHS of the ith linear equation in the linear system. Hence, `A[i] @ x == b[i]` checks if $\boldsymbol{x}$ is a solution to the ith linear equation, like above. Note that `A @ x` will return a vector whose ith entry is `A[i] @ x`. Hence, `A @ x == b` returns a Boolean vector whose ith entry is `A[i] @ x == b[i]`.[ii] Since $\boldsymbol{x}$ is a solution to the linear system if and only if each entry in `A[i]@x == b[i]` is `True`, it suffices to check if all of these Booleans are `True`. To determine if all the entries agree, and hence the vectors are equal, we may use the command `np.all` which will return the logical AND of all the Booleans in the `np.array`. Hence, $\boldsymbol{x}$ is a solution of the linear system if and only if `np.all(A @ x == b)` is `True`.

[i] Note that the double equal sign `==` checks if two values are equal (a Boolean) instead of changing the value on the left-hand-side which a single equal sign `=` does in Python.

[ii] When comparing `np.array`'s, instead of checking if the vectors are equal, `==` will return instead a vector of Booleans, where each entry is `True` or `False` if those specific entries match.

Exercises

(Go to Solutions)

For Exercises 1–4, determine if the point is a solution to the following linear system

$$\begin{cases} 2x - 3y + z = 0 \\ y + z = 0 \\ -x - 2z = 0. \end{cases}$$

1. $(0, 0, 0)$ 2. $(4, 2, -2)$ 3. $(3, 2, 0)$ 4. $(1, 1, 1)$

For Exercises 5–8, determine if the point is a solution to the following linear system

$$\begin{cases} x + y = 7 \\ 3x + 3y - 10z = -79 \\ x + 5y + z = 33. \end{cases}$$

5. $(4, 3, 10)$ 6. $(0, 0, 0)$ 7. $(3, 4, 10)$ 8. $(-10, 3, -38)$

For Exercises 9–12, determine if the point is a solution to the following linear system

$$\begin{cases} x + 3y - 2z = 13 \\ 2x - y = 2 \\ -x + 6y - 4z = 17. \end{cases}$$

♠ 9. $(6, 10, 4)$ ♠ 10. $(5, 8, 8)$ ♠ 11. $(3, 4, 1)$ ♠ 12. $(9, 3, 2)$

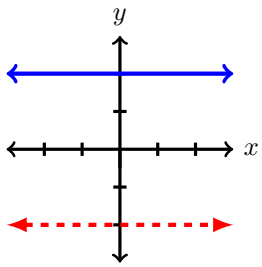
For Exercises 13–17, determine if the point is a solution to the following linear system

$$\begin{cases} x - y + z = 5 \\ 3x + 4y - 5z = -2 \\ -2x - 4y + 7z = 11. \end{cases}$$

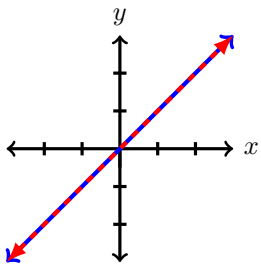
13. $(1, 2, 3)$ 14. $(3, 2, 1)$ 15. $(5, 0, 6)$ 16. $(3, 1, 3)$ 17. $(9, 3, 2)$

For Exercises 18-21, the graph of two lines in $\mathbb{R}^2$ are provided. Describe the number of solutions of the associated linear system. Is the system consistent or inconsistent?

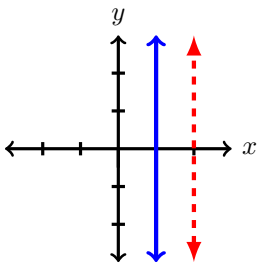
18.



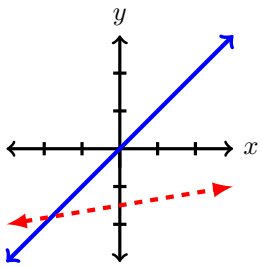
19.



20.



21.



For Exercises 22–31, graph the linear system and determine if it is consistent or inconsistent. If consistent, determine which whether it has a unique solution or multiple solutions.

22. $\begin{cases} 14x + 3y = 4 \\ 5x + 9y = 1 \end{cases}$

23. $\begin{cases} 2x - 5y = 10 \\ \frac{4}{5}x - 2y = 4 \end{cases}$

24. $\begin{cases} x = 5y - 4 \\ \frac{1}{5}x = y - 3 \end{cases}$

25. $\begin{cases} 2x + 3y = 4 \\ x - y = 2 \end{cases}$

26. $\begin{cases} -4x + y = 7 \\ -8x + 2y = 2 \end{cases}$

27. $\begin{cases} x - y = 1 \\ 2y = 2x - 2 \end{cases}$

28. $\begin{cases} y = 2x + 7 \\ y = -4x - 2 \end{cases}$

29. $\begin{cases} y = 5x + 8 \\ -5x + y - 2 = 6 \end{cases}$

30. $\begin{cases} y = 3x - 7 \\ y - 3x = 2 \end{cases}$

31. $\begin{cases} 9x + 5 = 2y \\ x + 2y = 5 \\ y = 2.5 \end{cases}$

For Exercises 32–42, determine if the linear system is homogeneous. For those which are, find a solution.

32. $\begin{cases} 4x + 2y = 0 \\ 5x + 3y = 0 \end{cases}$

33. $\begin{cases} 6x - 7y = 3 \\ 2x + 8y = 0 \end{cases}$

♠ 34. $\begin{cases} 3x - y = 0 \\ 2x + 5y = 0 \end{cases}$

♠ 35. $\begin{cases} 2x - 4y = 0 \\ -6x + 8y = 3 \end{cases}$

36. $\begin{cases} -5 - y - 2 = 0 \\ x + y + 5 = 0 \end{cases}$

37. $\begin{cases} 2x + y = 10 \\ -7x - 3y = 3 \end{cases}$

38. $\begin{cases} 5x + 2y + 3z = 2 \\ 3x - 4z = -1 \\ 5y + 3z = 5 \end{cases}$

39. $\begin{cases} 5x + 2y + 3z = 2 \\ -3x - 2y + z = 5 \\ 2x + 2y - 4z = 1 \end{cases}$

♠ 40. $\begin{cases} x - 2z = 0 \\ 3x + 4y + 4z + w = 0 \\ -2x + 6y + 6z + 2w = 0 \end{cases}$

41. $\begin{cases} 2x - 2y - 2z = 6 \\ -x - 3y + 9z = 1 \\ 4x + 3y - 18z = 3 \end{cases}$

42. $\begin{cases} x + y - w + v = 0 \\ 2x + 3z + 15w + 3v = 0 \\ -x + 37y - 5z - 8v = 0 \end{cases}$

For Exercises 43-43, create a $m \times n$ linear system that has the given point as a solution. (Answers may vary).

43. 3×3, $(4, 0, -1)$

44. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/linearsystems.ipynb

"When you put your hand to the plow, you can't put it down until you get to the end of the row."
– Alice Paul

2.2 Augmented Matrices

In this section, we introduce matrices to encode linear systems. We begin to find an effective algorithm for solving linear systems. The three Elementary Row Operations will be the basic techniques used to solve linear systems. In the elementary row operations, we refer to equations as "rows" for reasons that will be more clear by the end of this section.

2.2.1 The Elementary Row Operations

Elementary Row Operations[i]

1. (Replacement) Replace one row by the sum of itself and a multiple of another row.

2. (Interchange) Interchange the order of any two rows in the system.

3. (Scaling) Multiply all scalars in a row by a nonzero scalar.

We say that two systems of linear equations are **row equivalent** if there is a sequence of row operations that transforms one system into the other.

Theorem 2.2.1. *Two linear systems are equivalent if and only if they are row equivalent.*

Example 2.2.2. Solve

$$\begin{cases} x_1 - 2x_2 + 2x_3 = 0 \\ 2x_2 - 8x_3 = -8 \\ -4x_1 + 6x_2 + 2x_3 = 10 \end{cases}$$

using row operations.

We begin by replacing Row 3 with (Row 3 + 4 ∗ Row 1). This gives:

$$\begin{cases} x_1 - 2x_2 + 2x_3 = 0 \\ 2x_2 - 8x_3 = -8 \\ -2x_2 + 10x_3 = 10 \end{cases}$$

Next, we scale Row 2 by a factor of $\frac{1}{2}$, which gives:

$$\begin{cases} x_1 - 2x_2 + 2x_3 = 0 \\ x_2 - 4x_3 = -4 \\ -2x_2 + 10x_3 = 10 \end{cases}$$

[i]The elementary row operations come by many names, and many texts never give them names are all. The replacement operations is sometimes called row addition or row combination because it consists of adding a multiple of a row to another row. The interchange operation is sometimes called row swap or row switch for obvious reasons. Finally, the scaling operation is often called row multiplication because we multiply both sides of an equation by a nonzero constant. The naming scheme used here for the elementary row operations follows the scheme used by Lay [**?**]. We prefer this naming scheme for two reasons. First, students will remember them better and be able to talk with their classmates about them more easily when they have short, simple names. Second, we will eventually see that these three elementary operations correspond to many concepts in linear algebra, including elementary column operations. For this reason, it is preferable to have names which do not use the word "row."

Lastly, we replace Row 3 with
(Row 3 + 2 ∗ Row 2), which gives:

$$\begin{cases} x_1 - 2x_2 + 2x_3 = 0 \\ x_2 - 4x_3 = -4 \\ 2x_3 = 2 \end{cases}$$

At this point, we recognize that we have solved for x_3, that is, $x_3 = 1$. Substituting this into Row 2 gives

$$x_2 - 4x_3 = -4 \quad \Rightarrow \quad x_2 = -4 + 4x_3 = -4 + 4(1) = -4 + 4 = 0.$$

Now we have solved for x_2. Lastly, we use these values to solve for x_1:

$$x_1 - 2x_2 + 2x_3 = 0 \quad \Rightarrow \quad x_1 = 2x_2 - 2x_3 = 2(0) - 2(1) = 0 - 2 = -2.$$

Therefore, $(x_1, x_2, x_3) = \boxed{(-2, 0, 1)}$ is the unique solution to the above system. ∎

2.2.2 Augmented Matrices

Definition 2.2.3. If m and n are positive integers, an $m \times n$ **matrix** is a rectangular array of scalars with m rows and n columns. Note that the numbers of rows always comes first.

For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 0 & -3 \end{bmatrix}$$

is a 2×3 matrix over $\mathbb{R}$.

The essential information of a linear system can be recorded compactly in a matrix. Consider the system from the previous example

$$\begin{cases} x_1 - 2x_2 + 2x_3 = 0 \\ 2x_2 - 8x_3 = -8 \\ -4x_1 + 6x_2 + 2x_3 = 10 \end{cases}$$

Aligning similar variables into columns, we create the **coefficient matrix**

$$\begin{bmatrix} 1 & -2 & 2 \\ 0 & 2 & -8 \\ -4 & 6 & 2 \end{bmatrix}$$

and the **augmented matrix**

$$\left[\begin{array}{ccc|c} 1 & -2 & 2 & 0 \\ 0 & 2 & -8 & -8 \\ -4 & 6 & 2 & 10 \end{array}\right]$$

Example 2.2.4. Solve the following system of equations:

$$\begin{cases} 3x_2 + 3x_3 = 11 \\ 2x_1 - 3x_2 + 3x_3 = -4 \\ x_1 + x_2 + 4x_3 = 3 \end{cases}$$

using the augmented matrix.

The augmented matrix is:

$$\left[\begin{array}{ccc|c} 0 & 3 & 3 & 11 \\ 2 & -3 & 3 & -4 \\ 1 & 1 & 4 & 3 \end{array}\right]$$

Now, the row operations for linear systems work on augmented matrices as well. Interchanging Row's 1 and 3 gives:

$$\left[\begin{array}{ccc|c} 1 & 1 & 4 & 3 \\ 2 & -3 & 3 & -4 \\ 0 & 3 & 3 & 11 \end{array}\right]$$

Next, replace Row 2 with (Row 2 − 2Row 1), which gives:

$$\left[\begin{array}{ccc|c} 1 & 1 & 4 & 3 \\ 0 & -5 & -5 & -10 \\ 0 & 3 & 3 & 11 \end{array}\right]$$

Next, scale Row 2 by $-\frac{1}{5}$ to get:

$$\left[\begin{array}{ccc|c} 1 & 1 & 4 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 3 & 3 & 11 \end{array}\right]$$

Finally, replace Row 3 with (Row 3 − 3Row 2), which gives:

$$\left[\begin{array}{ccc|c} 1 & 1 & 4 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 5 \end{array}\right]$$

This implies that

$$\begin{cases} x_1 + x_2 + 4x_3 = 3 \\ x_2 + x_3 = 2 \\ 0 = 5 \end{cases}$$

which is impossible. Therefore, the system has no solution, that is, the system is $\boxed{\text{inconsistent}}$. ∎

2.2.3 Echelon Forms

Definition 2.2.5. In a matrix, we say a row is a **zero row** if all scalars in this row are zero. Otherwise, we call it a **nonzero row**. In a nonzero row, we say the **leading entry** of the row is the leftmost, nonzero entry in the row.

A matrix is in (**row**) **echelon form** if it has the following three properties:

1. There is no zero row above a nonzero row.
2. Each leading entry is in a column to the right of the leading entry of the row above it.
3. All entries in a column below a leading entry are zero.

Essentially, 1. requires all zero rows to be at the bottom of a matrix in echelon form. Additionally, 2. and 3. require there exists a downward staircase of zeros in the lower left of the matrix, hence the name echelon.

A matrix in echelon form is in (**row**) **reduced echelon form** (or **RREF**) if additionally:

4. The leading entry in each nonzero row is 1.
5. All entries in a column above a leading entry are zero.

When considering whether an augmented matrix is in (row reduced) echelon form, consider only those columns to the left of the vertical line. That is, an augmented matrix is in (row reduced) echelon form if and only if its coefficient matrix is.

A **pivot position**, or simply just a **pivot**, in a matrix is a location that corresponds to a leading entry in one of its echelon forms. A **pivot column** (or **row**) is a column (or row) that contains a pivot position. The number of pivots of a matrix A is called its **rank**, denoted $\text{rank}(A)$.

A priori, we do not know the locations of the pivots in a matrix, but we can easily see them when the matrix is in echelon form. Now if two matrices are row equivalent, then their pivot positions will be the same. Thus, it will be highly useful to be able to compute echelon forms row equivalent to given matrices.

Example 2.2.6. The following two matrices are in echelon form:

$$\left[\begin{array}{ccc|c} \boxed{1} & 2 & -5 & 3 \\ 0 & 0 & \boxed{1} & -1 \\ 0 & 0 & 0 & 5 \end{array}\right] \quad \text{and} \quad \left[\begin{array}{ccc|c} \boxed{1} & 0 & 0 & -3 \\ 0 & \boxed{1} & 0 & 0 \\ 0 & 0 & \boxed{1} & 4 \end{array}\right].$$

The first matrix is NOT in row reduced echelon form, since there are nonzero entries above the pivot in the third column. The second one is and the first three columns are pivot columns. The first matrix has rank 2, and the second has rank 3. ∎

Solving systems of linear equations when in echelon form is very simple. Solve first for the equation involving one variable. Substitute this assignment of the variable into the linear equation involving two variables. Solve for the remaining unknown. Plug these two assignments into the equation with three unknowns. Repeat this process as necessary. This technique is often called **back-substitution**.

Example 2.2.7. Solve the following linear systems from their augmented matrices which are in echelon form.

(a) Examining below the echelon matrix and its corresponding system of linear equations, we see that we can easily solve this system. Starting with the third equation, we get that $z = 5$. Substituting this into the second equation, which depends only on y and z, we get $y = -46+8z = -46+40 = -6$. Finally, we substitute these values into the first equation and get $x = \frac{1}{2}(y - 3z + 25) = \frac{1}{2}(-6 - 15 + 25) = \frac{4}{2} = 2$. Therefore, the unique solution to the system is $\boxed{(2, -6, 5)}$.

$$\left[\begin{array}{ccc|c} 2 & -1 & 3 & 25 \\ 0 & -1 & 8 & 46 \\ 0 & 0 & 15 & 75 \end{array}\right] \sim \begin{cases} 2x - y + 3z = 25 \\ -y + 8z = 46 \\ 15z = 75 \end{cases}$$

(b) This augmented matrix is in row reduced echelon form. In terms of the linear system, the system is already solved, and the solution corresponds to the augmented column of the matrix, that is, $\boxed{(3,-5,3)}$ is the unique solution to this linear system.

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & -5 \\ 0 & 0 & 1 & 3 \end{array}\right] \sim \begin{cases} x = 3 \\ y = -5 \\ z = 3 \end{cases}$$

(c) The final equation in this system is an identity which is true for all assignments of the variables. It neither adds or takes away any restrictions on the solution set. In fact, this equation could be removed without changing the solution set. Because the system essentially has two equations but three variables, it must be true that at least one of the variables is free.

$$\left[\begin{array}{ccc|c} 1 & 0 & -3 & 2 \\ 0 & 1 & 3 & 7 \\ 0 & 0 & 0 & 0 \end{array}\right] \sim \begin{cases} x_1 - 3x_3 = 2 \\ x_2 + 3x_3 = 7 \\ 0 = 0 \end{cases} \sim \begin{cases} x_1 = 2 + 3x_3 \\ x_2 = 7 - 3x_3 \\ x_3 = \text{any number} \end{cases}$$

Notice that the equation $0 = 0$ places no restriction on the variables. It could be removed without altering the solution set. Thus, there is no restriction placed on x_3, that is, x_3 could be freely assigned any real number t. This is what we mean by calling x_3 a *free variable*. Upon choosing any such number, x_1 and x_2 are determined by their relation with x_3, that is, their assignment is restricted by equations 1 and 2. Thus, this system has multiple solutions, whose general form is $\boxed{(2+3t, 7-3t, t)}$.

Whether there is a free variable or not in the a linear system is relatively easy to determine once you know how to look for it. In particular, dependent variables correspond to pivot columns of the augmented matrix, and free variables correspond to the remaining columns, the so-called **non-pivot columns**.

Theorem 2.2.8. *A linear system has multiple solutions if and only if it is consistent and have at least one non-pivot column. These non-pivot columns will correspond in a one-to-one manner with the free variables of the linear system.*

(d) Examples like the previous one sometimes give the false narrative that multiple solutions occur only when the linear system contains the identity $0 = 0$ in some echelon form. This is patently false. Consider the following *overdetermined* system (more rows than columns) below. The linear system contains two rows of zeros but has a unique solution, namely $(5, -2)$.

$$\left[\begin{array}{cc|c} 1 & 0 & 5 \\ 0 & 1 & -2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right].$$

On the other hand, an *underdetermined* system (more columns than rows) will necessarily have some columns which cannot have any pivots, as the number of pivots is bounded above by the number of rows and the number of columns. Thus, it MUST have at least one non-pivot column (see Proposition 2.1.8). This translates to mean that an undetermined system MUST have free variables, which will provide multiple solutions if the linear system is consistent. For example, the below linear system is underdetermined but has no row of zeros. Nonetheless, the third column is non-pivot, meaning

that x_3 is a free variable of the system. The general solution would then be $(2, 3-2t, t)$ for any scalar t.

$$\left[\begin{array}{cccc} \boxed{1} & 1 & 2 & 5 \\ 0 & \boxed{1} & 2 & 3 \end{array}\right] \sim \left[\begin{array}{cccc} \boxed{1} & 0 & 0 & 2 \\ 0 & \boxed{1} & 2 & 3 \end{array}\right] \sim \begin{cases} x_1 & & = 2 \\ & x_2 + 2x_3 & = 3 \end{cases} \sim \begin{cases} x_1 & = 2 \\ x_2 & = 3 - 2x_3. \end{cases}$$

The false narrative that rows of zeros produce free variables in a linear system derives from the over-exposure linear algebra students often get from $n \times n$ (square) linear systems. In this case, if there is a row of zeros then the square system devolves into an undetermined system which has a free variable. The multiple solutions follow, not from the row of zeros, but from the non-pivot column(s).

(e) The third equation gives a contradiction, since there is no choice of variables such that $0 = 8$. This tells us that the linear system is $\boxed{\text{inconsistent}}$.

$$\left[\begin{array}{ccc|c} 1 & 0 & -1 & 1 \\ 0 & 2 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{array}\right] \sim \begin{cases} x_1 & - & x_3 & = 1 \\ & 2x_2 & + 4x_3 & = 4 \\ & & 0 & = 8 \end{cases}$$

Theorem 2.2.9 (Rouché-Capelli Theorem)**.** *A linear system is inconsistent if and only if it contains a row of the form*

$$\left[\begin{array}{cccc|c} 0 & 0 & \dots & 0 & b \end{array}\right] \qquad (b \neq 0)$$

in some echelon form of the matrix. If the linear system is consistent, then the solution is unique if and only if the system has no free variables. ∎

Python Programming

When implemented in Python, an $m \times n$ matrix is an array of array (or a vector of vectors). Each vector in the array is considered a row vector of the matrix and each will have n entries. Then the array that holds the vectors will have length m. Essentially a matrix is entered as a block vector, where each vector in the block is itself a row vector with the same length n (do make sure all the internal arrays have the same length.). Unlike the block vectors from before which were concatenated together, a matrix is a stack of these row vectors. When using an augmented matrix, note that NumPy does not have a feature to indicate the partition between the coefficient matrix and the augmented column.

The `shape` of a matrix outputs the number of rows first m, followed by the number of columns n.

To grab an entire row vector, say the ith row of matrix A, use a single index, i.e. `A[i]`. To grab a specific entry in the matrix A, say $a_{i,j}$, use a double index, i.e., `A[i,j]`. Grabbing a column vector, say the jth column, is less intuitive but can be done with a double index involving a colon, e.g. `A[:,j]`.

Conversely, if one wants to generate the matrix using the column vectors, one can use the command `np.column_stack( ...)` instead, where an array of vectors is included between the parenthesis.

Example 2.2.10. The code:

```
import numpy as np

A = np.array([[0, 1, -3, -1, -2],# 4x5 matrix called A, which is implemented as an array of
    arrays;
              [1, 1, 2, 4, 3],   # each array represents a row of A;
              [3, 7, -6, 8, 1],  # each entry corresponds to a scalar in that row.
              [0, -1, 3, 4, -4]])
print(A)

m,n = A.shape                    # shape returns (m, n), where m = rows, n = columns
print(A.shape)
print('m =', m)
print('n =', n)

print("The 2nd row: ", A[1])     # A[i] return i-th row, so A[1] grabs the second row
print("The 3rd column: ", A[:,2])# the colon grabs all rows, hence grabs column 3 (index 2)
print("a_(1,3) = ", A[0,2])      # A[i, j] grabs single entry at row i, column j.

# Each list inside np.column_stack is treates as a column vector.
# This is the counter-wise counterpart to np.array, which stacks rows.
B = np.column_stack([[0,1,3,0],
                     [1,1,7,-1],
                     [-3,2,-6,3],
                     [-1,4,8,4],
                     [-2, 3, 1, -4]])
print(B)
```

records and prints the matrix $\left[\begin{array}{cccc|c} 0 & 1 & -3 & -1 & -2 \\ 1 & 1 & 2 & 4 & 3 \\ 3 & 7 & -6 & 8 & 1 \\ 0 & -1 & 3 & 4 & -4 \end{array}\right]$ and produces the output:

```
[[ 0  1 -3 -1 -2]
 [ 1  1  2  4  3]
 [ 3  7 -6  8  1]
 [ 0 -1  3  4 -4]]
(4, 5)
m:  4
n:  5
The 2nd row:  [1 1 2 4 3]
The 3rd column:  [-3 2 -6 3]
```

```
a_(1,3) = -3
[[ 0  1 -3 -1 -2]
 [ 1  1  2  4  3]
 [ 3  7 -6  8  1]
 [ 0 -1  3  4 -4]]
```

■

Exercises

(Go to Solutions)

For Exercises 1-5, each of the augmented matrices are in echelon form. Identify the pivot positions and which variables, e.g. x_1, x_2, $x_3, \ldots$, are free variables in the associated linear system.

1. $\left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]$

2. $\left[\begin{array}{cccc|c} 1 & 2 & 1 & 4 & -1 \\ 0 & 1 & 3 & 5 & 0 \end{array}\right]$

3. $\left[\begin{array}{cccc|c} 0 & 2 & 1 & 3 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 4 & 0 \end{array}\right]$

4. $\left[\begin{array}{cccc|c} 1 & 4 & 2 & 6 & 1 \\ 0 & 2 & 2 & 1 & 2 \\ 0 & 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$

5. $\left[\begin{array}{ccccc|c} 1 & 3 & 5 & 0 & 1 & 1 \\ 0 & 3 & 4 & 1 & 2 & 2 \\ 0 & 0 & 0 & 6 & 1 & 3 \\ 0 & 0 & 0 & 0 & 2 & 4 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{array}\right]$

For Exercises 6-30, identify if the matrix is in echelon form and if it is in row reduced echelon form. If in echelon form, identify the pivot positions and the rank of the matrix.

6. $\left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]$

7. $\left[\begin{array}{cccc} 1 & 2 & 0 & -3 \\ 0 & 0 & 1 & 5 \end{array}\right]$

8. $\left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]$

9. $\left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]$

10. $\left[\begin{array}{cccc} 3 & -2 & 4 & 5 \end{array}\right]$

11. $\left[\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array}\right]$

12. $\left[\begin{array}{ccc} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]$

13. $\left[\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]$

14. $\left[\begin{array}{cccc} 1 & 2 & 4 & -3 \\ 0 & 0 & 1 & 5 \end{array}\right]$

15. $\left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]$

16. $\left[\begin{array}{cccc} 1 & 6 & -5 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{array}\right]$

17. $\left[\begin{array}{cc|c} 5 & 6 & 3 \\ 0 & 4 & 12 \end{array}\right]$

18. $\left[\begin{array}{ccc|c} 2 & 4 & 3 & 6 \\ 0 & 2 & 6 & 2 \\ 0 & 0 & 1 & 4 \end{array}\right]$

19. $\left[\begin{array}{ccc|c} 1 & 0 & 6 & 2 \\ 0 & 0 & 3 & 4 \\ 4 & 0 & 1 & 6 \end{array}\right]$

♠ 20. $\left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 5 \\ 0 & 6 & 7 & 8 & 9 \\ 0 & 0 & 0 & 3 & 4 \end{array}\right]$

♠ 21. $\left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 & 2 \end{array}\right]$

♠ 22. $\left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 6 & 7 & 8 & 9 \end{array}\right]$

♠ 23. $\left[\begin{array}{cccc|c} 1 & 0 & 3 & 0 & 5 \\ 0 & 1 & 7 & 0 & 9 \\ 0 & 0 & 0 & 1 & 2 \end{array}\right]$

♠ 24. $\left[\begin{array}{cccc|c} 1 & 0 & 3 & 4 & 5 \\ 0 & 1 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 & 2 \end{array}\right]$

♠ 25. $\left[\begin{array}{cccc|c} 1 & 0 & 0 & 4 & 5 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 8 & 9 \end{array}\right]$

26. $\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 5 & -1 & 2 & -2 \\ 3 & -1 & 1 & 3 \end{array}\right]$

27. $\left[\begin{array}{ccc|c} 1 & -\frac{1}{5} & \frac{2}{5} & -\frac{2}{5} \\ 0 & 2 & 1 & \frac{7}{3} \\ 0 & 0 & 0 & 1 \end{array}\right]$

28. $\left[\begin{array}{ccc|c} 1 & 0 & \frac{1}{2} & 0 \\ 0 & 1 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]$

29. $\left[\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 9 & 1 & 3 \\ 0 & 0 & 6 & 7 \\ 2 & 1 & 3 & 4 \end{array}\right]$

30. $\left[\begin{array}{cccc|c} 1 & 0 & 0 & 3 & 5 \\ 0 & 7 & 0 & 0 & 4 \\ 0 & 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$

For Exercises 31-31, write an augmented matrix in echelon form satisfying the listed conditions. Identify the pivot positions. (Answers may vary).

31. rank 3, no zero rows, underdetermined

For Exercises 32-40, write the linear system as augmented matrix or vice versa.

32. $\begin{cases} 4x_2 + 2x_3 = 6 \\ 3x_1 + 5x_2 + 2x_3 = 7 \\ 3x_1 + 17x_2 + 8x_3 = 24 \end{cases}$

♠ 33. $\begin{cases} 3x - y = 0 \\ -2y = 5 \\ x + 7y = 13 \end{cases}$

♠ 34. $\begin{cases} x + y + z = 1 \\ -2x - 6y + z = 12 \end{cases}$

35. $\begin{cases} 10x - 7y + 2z - 4w = 10 \\ 3x + 4y - 3z + w = 20 \\ x + 4z = 73 \\ 6y - 3z + 2w = 10 \end{cases}$

36. $\begin{cases} 12x - 2y + 3z + \frac{3}{2}w = 13 \\ x + 3y - 2z = 9 \\ y + 17z - 20w = 12 \end{cases}$

37. $\left[\begin{array}{ccc|c} 1 & -2 & -3 & -4 \\ 0 & 1 & 2 & 1 \\ -2 & 0 & -2 & 8 \end{array}\right]$

♠ 38. $\left[\begin{array}{ccc|c} 3 & -2 & -1 & 4 \\ 1 & 0 & 3 & -3 \\ 0 & 0 & 0 & 2 \end{array}\right]$

♠ 39. $\left[\begin{array}{cccc|c} 4 & -1 & -2 & 5 & 12 \\ -3 & 0 & 0 & 1 & 5 \\ 1 & 0 & 0 & 0 & 0 \end{array}\right]$

40. $\left[\begin{array}{cccc|c} 2 & 0 & 1 & -9 & 3 \\ 0 & 9 & 21 & 25 & -9 \\ 14 & -7 & 0 & 36 & 67 \\ 3 & 4 & -7 & 19 & 2 \\ -18 & 0 & 2 & 8 & 1 \end{array}\right]$

For Exercises 41-50, perform the indicated elementary row operation(s) to the linear system or augmented matrix.

41. $\begin{cases} 2x + 2y = 5 \\ x + y - z = 1 \\ -5y + 4z = -1 \end{cases}$
scale Row 2 by 3

♠ 42. $\begin{cases} 2y + 5z = 10 \\ x - y + 3z = 6 \\ 2x + 7y - z = 0 \end{cases}$
interchange Rows 1 and 2

♠ 43. $\begin{cases} 3x - y = 0 \\ -2y = 5 \\ x + 7y = 13 \end{cases}$
scale Row 2 by $-1/2$

♠ 44. $\begin{cases} x + y + z = 1 \\ -2x - 6y + z = 12 \end{cases}$ replace Row 2 with Row 2 + 2Row 1

45. $\left[\begin{array}{ccc|c} 1 & 3 & 1 & 3 \\ 0 & 1 & 7 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right]$
replace Row 1 with Row 1 + Row 3

46. $\left[\begin{array}{cccc|c} 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 0 & 0 & 0 & 7 \end{array}\right]$
replace Row 3 with Row 3 + 2Row 2

47. $\left[\begin{array}{cccc|c} 2 & 4 & 6 & -2 & 4 \\ 0 & 1 & 3 & 5 & 5 \\ 0 & 0 & 0 & 1 & 1 \end{array}\right]$
scale Row 1 by 1/2

48. $\left[\begin{array}{ccc|c} 2 & 3 & 4 & 6 \\ 1 & 2 & 2 & 3 \\ 9 & 4 & 7 & 2 \end{array}\right]$
replace Row 1 with Row 1 - Row 2

49. $\left[\begin{array}{ccc|c} 0 & 2 & 7 & 6 \\ 4 & 8 & 0 & 2 \\ 1 & 2 & 4 & 3 \end{array}\right]$
scale Row 2 by $\frac{1}{4}$,
replace Row 3 with Row 3 + 2Row 2

50. $\left[\begin{array}{ccc|c} 1 & 2 & 3 & 6 \\ 2 & 3 & 4 & 1 \\ 5 & 6 & 7 & 2 \end{array}\right]$
replace Row 2 with Row 2 - 2Row 1,
replace Row 3 with Row 3 - 5Row 1,
replace Row 3 with Row 3 - 4Row 2

For Exercises 51-55, identify the elementary row operation(s) which transform the first matrix into the second matrix. Answers may vary.

51. $\begin{bmatrix} 0 & 2 & 6 \\ 7 & 1 & 2 \\ -3 & 5 & 0 \end{bmatrix} \sim \begin{bmatrix} 7 & 1 & 2 \\ 0 & 2 & 6 \\ -3 & 5 & 0 \end{bmatrix}$

52. $\begin{bmatrix} 3 & 2 & 0 \\ 5 & 4 & 1 \\ 7 & 0 & 1 \end{bmatrix} \sim \begin{bmatrix} 1 & -2 & 0 \\ 5 & 4 & 1 \\ 3 & 2 & 0 \end{bmatrix}$

53. $\begin{bmatrix} 2 & 0 & 3 \\ 0 & 2 & -2 \\ 3 & 4 & -1 \end{bmatrix} \sim \begin{bmatrix} 1 & 4 & -4 \\ 0 & 1 & -1 \\ 0 & -8 & 11 \end{bmatrix}$

54. $\begin{bmatrix} 1 & -3 & 0 & -9 \\ 3 & -8 & 2 & -21 \\ 2 & -2 & 9 & 7 \\ -2 & 4 & -1 & 10 \end{bmatrix} \sim \begin{bmatrix} 1 & -3 & 0 & -9 \\ 0 & 1 & 2 & 6 \\ 0 & 4 & 9 & 25 \\ 0 & -2 & -1 & -8 \end{bmatrix}$

55. $\begin{bmatrix} 1 & 0 & 6 & 9 \\ 0 & 1 & 2 & 6 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 3 & 4 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

For Exercises 56–56, solve the linear systems, already in echelon form, using back-substitution.

56. $\begin{cases} x + y + 3z = 9 \\ y + 4z = 8 \\ z = 1 \end{cases}$

57. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/aug.ipynb

"A leader is someone who helps improve the lives of other people or improve the system they live under."
– Sam Houston

2.3 Reduction of Linear Systems

In this section, we discuss algorithms that can be used to row reduce a matrix into echelon form, from which we can use the echelon form to solve the linear system associated with this matrix.

2.3.1 Gaussian Elimination

The key to solving linear systems is to row reduce the augmented matrix to an echelon form. To accomplish this, conduct the following recursive process: identify the leftmost nonzero column. This becomes a pivot column, with the pivot in the top position. If this pivot entry is zero, interchange the pivot row with any nonzero row below it. Next, zero out each entry below the pivot by replacing this row with itself plus a multiple of the pivot row. Consider next the minor matrix where this pivot row and pivot column are removed. Repeat this process on the minor matrix until an echelon form is obtained.

Algorithm 2.3.1 (The Forward Phase Algorithm)**.**

Given *an $m \times n$ matrix A.*

1. Identify the next pivot column. *Identify the leftmost nonzero column of A[i], if it exists. If so, label this a pivot column, say C_j, and place a pivot position in $(1, j)$ position inside the first row R_1. If not,* ***return*** *the $m \times n$ zero matrix.*

2. Place a nonzero entry in the pivot position, if necessary. *If the pivot position $(1, j)$ is nonzero, skip this step. Otherwise, interchange R_1 with the first row with a nonzero entry in the pivot column.*

3. Zero out every nonzero entry below the pivot position. *For each row R_i below the pivot row, replace R_i with $R_i - \dfrac{a_{ij}}{a_{1j}} R_1$.*

4. Recuse on the submatrix where already determined pivot columns and rows are removed. *If $m > 1$ and $j < n$, then replace the submatrix of A consisting of rows R_2 through R_m and columns C_j through C_n with an echelon by recursively calling the Forward Phase again. Then* ***return*** *the current matrix which is in echelon form.*

Returns *an echelon form of A.*

The Forward Phase Algorithm row reduction to echelon form of an augmented matrix coupled with the technique of back-substitution to solve the reduced system is call **Gaussian Elimination**. This provides an efficient procedure to solve linear systems.

Example 2.3.2. Reduce the matrix A to echelon form and solve the associated linear system using Gaussian Elimination, where $A = \left[\begin{array}{rrrrr|r} 0 & 0 & 0 & -2 & 3 & -7 \\ -2 & -4 & -6 & 1 & -1 & -3 \\ 4 & 8 & 12 & 1 & -2 & 16 \end{array}\right]$.

We begin by identifying the leftmost nonzero column of the matrix, which is the first column of A. As such, we place a pivot position in the first row of the first column. As this pivot position contains a zero value,

[i] If A is an augmented matrix, then we only search for pivot columns before the partitioning line.

we will interchange Rows 1 and 2, since Row 2 contains a nonzero value. (Note that the red double-arrow below will be shorthand we commonly use to denote the interchange of two rows in a matrix.)

$$\left[\begin{array}{ccccc|c} \boxed{0} & 0 & 0 & -2 & 3 & -7 \\ -2 & -4 & -6 & 1 & -1 & -3 \\ 4 & 8 & 12 & 1 & -2 & 16 \end{array}\right] \sim \left[\begin{array}{ccccc|c} \boxed{-2} & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & -2 & 3 & -7 \\ 4 & 8 & 12 & 1 & -2 & 16 \end{array}\right]$$

Now we proceed to place zero values in the all the entries below the pivot position in Column 1 using Row Replacement. As Row 2 already has a zero entry in Column 2 we may disregard it for now. To create a zero entry in Row 3 we will replace Row 3 with $\text{Row}3 + 2 \cdot \text{Row}1$, as shown. (Note that red statement "(Row i + cRow j)" written adjacent to Row i will indicate we are replacing Row i with Row i + cRow j. For convenience and to avoid arithmetic errors, we write cRow j write above Row i in red to indicate what values we will add together to produce the next matrix in the row reduction.)

$$\left[\begin{array}{ccccc|c} \boxed{-2} & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & -2 & 3 & -7 \\ 4 & 8 & 12 & 1 & -2 & 16 \end{array}\right] \begin{array}{l} \\ \\ {\scriptstyle -4 \quad -8 \quad -12 \quad 2 \quad -2 \quad -6} \\ \text{(Row 3 + 2 Row 1)} \end{array} \sim \left[\begin{array}{ccccc|c} \boxed{-2} & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & -2 & 3 & -7 \\ 0 & 0 & 0 & 3 & -4 & 10 \end{array}\right]$$

This completes the row reduction in Column 1. We now search for the next pivot position in the minor matrix where Row 1 and Column 1 (the location of the first pivot) are ignored. We notice there are only zero entries remaining in Column 2, so we skip this column. We do the same for Column 3. The leftmost nonzero column is then Column 4. So we place the pivot in the $(2, 4)$ position. As this entry is already nonzero, no interchange is necessary here.

$$\left[\begin{array}{ccccc|c} -2 & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & \boxed{-2} & 3 & -7 \\ 0 & 0 & 0 & 3 & -4 & 10 \end{array}\right]$$

Next, to remove the 3 below this second pivot we replace Row 3 with Row 3 + $\frac{3}{2}$Row 2. This is demonstrated below.

$$\left[\begin{array}{ccccc|c} -2 & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & \boxed{-2} & 3 & -7 \\ 0 & 0 & 0 & 3 & -4 & 10 \end{array}\right] \sim \left[\begin{array}{ccccc|c} -2 & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & \boxed{-2} & 3 & -7 \\ 0 & 0 & 0 & 0 & 1/2 & -1/2 \end{array}\right]$$

This completes the row reduction in the second pivot column. The third and final pivot would then be placed in the $(3, 5)$ position, as this is the leftmost nonzero column when Rows 1 and 2 and Columns 1-4 are ignored. As there are no entries below Row 3, the row reduction is completed automatically.

$$\left[\begin{array}{ccccc|c} -2 & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & -2 & 3 & -7 \\ 0 & 0 & 0 & 0 & \boxed{1/2} & -1/2 \end{array}\right]$$

Finally, row reduction ends when we reach the end of the coefficient matrix. Thus, our matrix is in echelon form. In order to solve the associated linear system, we will rewrite the augmented matrix as a system of equations and solve by back-substitution.

$$\begin{cases} -2x_1 - 4x_2 - 6x_3 + x_4 - x_5 = -3 \\ \qquad\qquad\quad - 2x_4 + 3x_5 = -7 \\ \qquad\qquad\qquad\quad \frac{1}{2}x_5 = -\frac{1}{2} \end{cases}$$

■

Remark 2.3.3. Many like to deviate from the Forward Phase algorithm in order to avoid the somewhat more cumbersome computations that follows by having entries which are fractions and having to do arithmetic by hand. One strategy to follow to avoid sometimes unnecessary fractions includes always choosing a row so that the pivot position is a one. Perhaps there is a row with already a one in it. Perhaps a row could be rescaled so there is a row in the pivot position, but if chosen incorrectly we might introduce fractions we are trying to avoid. Finally, a combination of row replacements can produce a pivot one if the greatest common divisor between two entries in the same column is itself one.

For example, in Example 2.3.2 when working on the second pivot, instead of replacing Row 3 with Row 3 + $\frac{3}{2}$Row 2, we could have replaced Row 2 with Row 2 + Row 3 and then replaced Row 3 with Row 3 – 3Row 2, as illustrated below. It takes more row operations but avoiding all fractions.

$$\left[\begin{array}{ccccc|c} -2 & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & \boxed{-2} & 3 & -7 \\ 0 & 0 & 0 & 3 & -4 & 10 \end{array}\right] \sim \left[\begin{array}{ccccc|c} -2 & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & \boxed{1} & -1 & 3 \\ 0 & 0 & 0 & 3 & -4 & 10 \end{array}\right]$$

$$\sim \left[\begin{array}{ccccc|c} -2 & -4 & -6 & 1 & -1 & -3 \\ 0 & 0 & 0 & \boxed{1} & -1 & 3 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{array}\right] \quad \blacktriangledown$$

2.3.2 Gauss-Jordan Elimination

Because the row reduced echelon form is more desirable, we can continue to row reduce a matrix in echelon form that was produced by the Forward Phase by transforming it into row reduced echelon form.

Algorithm 2.3.4 (The Backward Phase Algorithm)**.**

Given *an $m \times n$ matrix A in echelon form and pivot positions identified.*

1. Ignore any rows of zeros. *If there is no pivot in row R_m, skip to step 5.*
2. Identify the next pivot column. *Identify the rightmost pivot column C_j of A. Note this pivot position is in the (m, j) position.*
3. Create a leading one in the pivot column. *If the pivot position (m, j) is not 1, scale R_m by $\dfrac{1}{a_{mj}}$.*
4. Zero out every nonzero entry above the pivot position. *For each row R_i above R_m, replace R_m with $R_i - a_{ij}R_m$.*
5. Recuse on the submatrix where already reduced pivot columns are removed. *If $m > 1$, then replace the submatrix of A consisting of rows R_1 through R_{m-1} with the row reduced echelon form by recursively calling the Backward Phase again. Then* ***return*** *the current matrix which is in row reduced echelon form.*

Returns *the row reduced echelon form of A.*

The **Gauss-Jordan Elimination** method solves a linear system by first computing the row reduced echelon form of the augmented matrix (using the Forward and Backward Phases) and extracting the solution from this.

Example 2.3.5. Row reduce the matrix A to reduced echelon form, where $A = \left[\begin{array}{cccc|c} 0 & 1 & -3 & -1 & -2 \\ 1 & 1 & 2 & 4 & 3 \\ 3 & 7 & -6 & 8 & 1 \\ 0 & -1 & 3 & 4 & -4 \end{array}\right]$.

We begin with the forward phase of row reduction. We select position $(1,1)$ for the first pivot. As there is a zero in this pivot, we interchange Row 1 and Row 2:

$$\left[\begin{array}{cccc|c} \boxed{0} & 1 & -3 & -1 & -2 \\ 1 & 1 & 2 & 4 & 3 \\ 3 & 7 & -6 & 8 & 1 \\ 0 & -1 & 3 & 4 & -4 \end{array}\right] \sim \left[\begin{array}{cccc|c} \boxed{1} & 1 & 2 & 4 & 3 \\ 0 & 1 & -3 & -1 & -2 \\ 3 & 7 & -6 & 8 & 1 \\ 0 & -1 & 3 & 4 & -4 \end{array}\right]$$

Next, we replace Row 3 with Row 3 – 3 Row 1:

$$\left[\begin{array}{cccc|c} \boxed{1} & 1 & 2 & 4 & 3 \\ 0 & 1 & -3 & -1 & -2 \\ 3 & 7 & -6 & 8 & 1 \\ 0 & -1 & 3 & 4 & -4 \end{array}\right] \begin{array}{l} \\ \\ {\scriptstyle -3} \\ \\ \end{array} \begin{array}{l} \\ \\ {\scriptstyle -3} \\ \\ \end{array} \begin{array}{l} \\ \\ {\scriptstyle -6} \\ \\ \end{array} \begin{array}{l} \\ \\ {\scriptstyle -12} \\ \\ \end{array} \begin{array}{l} \\ \\ {\scriptstyle -9} \\ \\ \end{array} \begin{array}{l} \\ \\ \text{(Row 3 – 3 Row 1)} \\ \\ \end{array} \sim \left[\begin{array}{cccc|c} \boxed{1} & 1 & 2 & 4 & 3 \\ 0 & 1 & -3 & -1 & -2 \\ 0 & 4 & -12 & -4 & -8 \\ 0 & -1 & 3 & 4 & -4 \end{array}\right]$$

Next, we move the pivot to position $(2,2)$. Using this pivot, we replace Row 3 with Row 3 – 4 Row 2 and replace Row 4 with Row 4 + Row 1:

$$\left[\begin{array}{cccc|c} 1 & 1 & 2 & 4 & 3 \\ 0 & \boxed{1} & -3 & -1 & -2 \\ 0 & 4 & -12 & -4 & -8 \\ 0 & -1 & 3 & 4 & -4 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & 1 & 2 & 4 & 3 \\ 0 & \boxed{1} & -3 & -1 & -2 \\ 0 & {\scriptstyle -4}\,4 & -12 & {\scriptstyle 12}\,-4 & {\scriptstyle 4}\,-8 & {\scriptstyle 8} \\ 0 & {\scriptstyle 1}\,-1 & 3 & {\scriptstyle -3}\,4 & {\scriptstyle -1}\,-4 & {\scriptstyle -2} \end{array}\right] \begin{array}{l} \\ \\ \text{(Row 3 – 4 Row 1)} \\ \text{(Row 4 + Row 2)} \end{array} \sim \left[\begin{array}{cccc|c} 1 & 1 & 2 & 4 & 3 \\ 0 & \boxed{1} & -3 & -1 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & -6 \end{array}\right]$$

We next move the pivot to position $(3,4)$ as there are only zeros in $(3,3)$ and $(4,3)$. As the third row is a zero row, we next interchange Rows 3 and 4. We note that this matrix is now in echelon form. To continue to RREF, we scale Row 3 by $\frac{1}{3}$.

$$\left[\begin{array}{cccc|c} 1 & 1 & 2 & 4 & 3 \\ 0 & 1 & -3 & -1 & -2 \\ 0 & 0 & 0 & \boxed{0} & 0 \\ 0 & 0 & 0 & 3 & -6 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & 1 & 2 & 4 & 3 \\ 0 & 1 & -3 & -1 & -2 \\ 0 & 0 & 0 & \boxed{3} & -6 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \begin{array}{l} \\ \\ \frac{1}{3}\text{(Row 3)} \\ \\ \end{array} \sim \left[\begin{array}{cccc|c} 1 & 1 & 2 & 4 & 3 \\ 0 & 1 & -3 & -1 & -2 \\ 0 & 0 & 0 & \boxed{1} & -2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

We begin now the backward phase of row reduction. In order to zero-out the entries above the pivot, we replace Row 1 with Row 1 – 4 Row 3 and Row 2 with Row 2 + Row 3:

$$\left[\begin{array}{cccc|c} 1 & 1 & 2 & 4 & 3 \\ 0 & 1 & -3 & -1 & -2 \\ 0 & 0 & 0 & \boxed{1} & -2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \begin{array}{l} \\ \\ {\scriptstyle -4} \\ \\ \end{array} \begin{array}{l} \\ \\ {\scriptstyle 8} \\ \\ \end{array} \begin{array}{l} \\ \text{(Row 1 – 4 Row 3)} \\ \text{(Row 2 + Row 3)} \\ \\ \end{array} \sim \left[\begin{array}{cccc|c} 1 & 1 & 2 & 0 & 11 \\ 0 & 1 & -3 & 0 & -4 \\ 0 & 0 & 0 & \boxed{1} & -2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

We next move back to the pivot in position $(2,2)$. As this entry is already one, we proceed to eliminate the one above it. This is accomplished by replacing Row 1 with Row 1 – Row 2. This final matrix is the row reduced echelon form of the original matrix. All the pivot columns are indicated.

$$\left[\begin{array}{cccc|c} 1 & \overset{-1}{1} & 2 & \overset{3}{0} & \overset{4}{11} \\ 0 & \boxed{1} & -3 & 0 & -4 \\ 0 & 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \text{(Row 1 – Row 2)} \sim \left[\begin{array}{cccc|c} \boxed{1} & 0 & 5 & 0 & 15 \\ 0 & \boxed{1} & -3 & 0 & -4 \\ 0 & 0 & 0 & \boxed{1} & -2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \blacksquare$$

Theorem 2.3.6. *Each matrix is row equivalent to one and only one reduced echelon matrix.*

The proof of this important theorem essentially follows from Gauss-Jordan elimination.

Python Programming

Solving linear systems can be a little more problematic in NumPy (short for *Numerical Python*). The reason for this is because numerical linear algebra in Python is designed for speed over accuracy, that is, floating point numbers are used for their efficiency but have the weakness of rounding errors when too many digits are involved. Often these rounding errors are insignificant enough to ignore. To solve a linear system, use the command `np.linalg.solve(A,b)` to solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$. Be warned that this command will return an error if the coefficient matrix is non-square or singular. An $m \times n$ matrix is **square** if $m = n$, that is, the number of rows equals the number of columns. A square matrix is **singular** if there is not a pivot in each column. It is **nonsingular** otherwise. As we will see later, a linear system with a nonsingular coefficient matrix will always have a unique solution. Hence, if a matrix is non-square or singular, the system may be inconsistent or there might be multiple solutions, in which case NumPy is not capable of addressing such linear systems.

Alternatively, when a symbolic or exact calculation is necessary, the package SymPy (short for *Symbolic Python*) is preferred. Similar to NumPy, SymPy as a function for creating vectors and matrices, namely `Matrix(A)`. Each matrix object A has then the function `.rref()`, which takes no other input and returns an array of two quantities: first, the matrix representing the row-reduced echelon form of A, and, second, an array listing the pivot columns.

Note that symbolic linear algebra will be much slower and will use much more memory than numerical linear algebra. As such, we advise the usage of SymPy only on small matrices when NumPy is unacceptable.

Example 2.3.7. The code:

```
from sympy import pprint
import sympy as sp

A = sp.Matrix([[0, 1, -3, -1, -2], [1, 1, 2, 4, 3], [3, 7, -6, 8, 1], [0, -1, 3, 4, -4]])

pprint(A.rref()[0])
```

computes the row-reduced echelon form of the matrix from Example 2.3.5 and produces the output:

```
⎡1  0  5  0  15⎤
⎢              ⎥
⎢0  1  -3 0  -4⎥
⎢              ⎥
⎢0  0  0  1  -2⎥
⎢              ⎥
⎣0  0  0  0  0 ⎦
```

■

Exercises

(Go to Solutions)

For Exercises 1-8, compute the row reduced echelon form for the matrix. List the row operations used to transform the matrix to this form. Indicate the first time the matrix is in echelon form. Answers may vary.

1. $\begin{bmatrix} 1 & 2 \\ -4 & 5 \end{bmatrix}$

2. $\begin{bmatrix} 6 & -2 \\ 3 & 1 \end{bmatrix}$

♠ 3. $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 3 \\ 2 & 4 & 5 \end{bmatrix}$

♠ 4. $\begin{bmatrix} 1 & 2 & 2 & 1 \\ 2 & 1 & -2 & -2 \\ 1 & -1 & -4 & -3 \end{bmatrix}$

♠ 5. $\begin{bmatrix} 1 & 5 & 6 & -5 \\ 3 & 0 & -2 & 1 \\ 0 & 1 & 3 & 3 \\ -3 & 5 & 0 & 1 \end{bmatrix}$

6. $\left[\begin{array}{ccc|c} 1 & 2 & 0 & 9 \\ 0 & 2 & 4 & 6 \\ 0 & 0 & 5 & 10 \end{array}\right]$

7. $\left[\begin{array}{ccc|c} 8 & 16 & 0 & 14 \\ 0 & 0 & 4 & 7 \\ 0 & 9 & 3 & 9 \end{array}\right]$

8. $\left[\begin{array}{cccc|c} 3 & 0 & 6 & 9 & 18 \\ 0 & 2 & 4 & 0 & 8 \\ 0 & 0 & 0 & 3 & 6 \end{array}\right]$

For Exercises 9-13, solve the linear system using Gaussian elimination, that is, convert the linear system into an augmented matrix, reduce it to any echelon form, switch it back into a system of equations, and solve the reduced system using back-substitution.

9. $\begin{cases} x - y + 3z = 1 \\ 2x - 2y + 5z = 6 \\ 2x + 3y - 4z = 2 \end{cases}$

10. $\begin{cases} 2y + 6z = 4 \\ 3y + 2z + x = 5 \\ 3z + 3x + 3y = 3 \end{cases}$

11. $\begin{cases} 3x_1 + 2x_2 + x_3 = 1 \\ 3x_1 + 8x_2 + 3x_3 = 5 \\ 3x_2 + x_3 = 6 \end{cases}$

12. $\begin{cases} 2x_1 + 3x_2 - x_3 = 7 \\ 4x_1 - 2x_2 + 3x_3 = 5 \\ x_1 - x_2 + 2x_3 = 1 \end{cases}$

13. $\begin{cases} 2x_1 + x_2 + x_3 = 3 \\ x_1 + 2x_2 - x_3 = 1 \\ 3x_1 + x_2 + 2x_3 = 9 \end{cases}$

For Exercises 14-27, solve the linear system using Gauss-Jordan elimination, that is, convert the linear system into an augmented matrix, reduce it to row reduced echelon form, and use this to solve the linear system.

14. $\begin{cases} x + 2y = 2 \\ -4y = 8 \\ 2x + 2y = 6 \end{cases}$

♠ 15. $\begin{cases} x + y = 8 \\ x - y = 4 \end{cases}$

♠ 16. $\begin{cases} x - y = 6 \\ 2x - 3z = 16 \\ 2y + z = 4 \end{cases}$

17. $\begin{cases} 3y + 6z = 3 \\ x + 2y - 3z = 2 \\ 4x - 3z = 0 \end{cases}$

18. $\begin{cases} 3x_2 - 2x_3 = 0 \\ x_1 + x_2 + 3x_3 = 12 \\ 6x_1 + 2x_2 + x_3 = 13 \end{cases}$

19. $\begin{cases} x_1 + 4x_2 + 7x_3 = 3 \\ 2x_1 + 5x_2 + 8x_3 = 3 \\ 3x_1 + 6x_2 + 9x_3 = 3 \end{cases}$

♠ 20. $\begin{cases} 2x - 2y - 2z = 2 \\ 2x + 3y + z = 2 \\ 3x + 2y = 0 \end{cases}$

21. $\begin{cases} 7x + 8y + 9z = 10 \\ 4x + 5y + 6z = 10 \\ x + 2y + 3z = 10 \end{cases}$

22. $\begin{cases} 2x_1 + x_2 - x_3 = -10 \\ x_1 - 3x_2 - 2x_3 = -4 \\ 3x_1 + x_2 + x_3 = 15 \end{cases}$

23. $\begin{cases} 2x_1 + x_2 + 4x_3 = 199 \\ 2x_1 + 10x_2 + 15x_3 = 984 \\ -x_1 + 10x_3 = 58 \end{cases}$

24. $\begin{cases} 5x_1 + 2x_2 + 6x_3 = 11 \\ x_2 + 2x_3 = 5 \\ 3x_1 + 2x_2 + x_3 = 7 \\ 2x_1 + x_2 + 4x_3 = 6 \end{cases}$

25. $\begin{cases} x + y + z + w = 1 \\ 3y + z = 1 \\ x + y + 2z = 0 \\ 2x + z + 4w = 0 \end{cases}$

26. $\begin{cases} 2x_1 - 3x_2 + 6x_3 + 2x_4 = 5 \\ -2x_1 + 3x_2 - 3x_3 - 3x_4 = -4 \\ 4x_1 + 6x_2 + 9x_3 + 5x_4 = 9 \\ -2x_1 + 3x_2 + 3x_3 - 4x_4 = 1 \end{cases}$

27. $\begin{cases} 2x = 10 - 4y + 2z \\ -10y = 20 - 16x - 3z - 3 + 13y + 16 - 2z \\ -3z = -4x + 3z - 2y + z \end{cases}$

28. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/echelon.ipynb

"We cannot solve our problems with the same thinking we used when we created them." – Albert Einstein

2.4 Applications of Linear Systems

In this section, we discuss how to set up and solve applications of linear systems arising in mathematics and science.

2.4.1 Data Fitting

Linear systems can be used for **data fitting** (or **interpolation**), that is, given a mathematical model $g(x) = \sum_{i=1}^{n} c_i f_i(x)$ where $c_i \in \mathbb{R}$ are scalars and f_i are real-valued functions and given a set of n points $(x_1, y_1), \ldots, (x_n, y_n)$, we can find the missing coefficients $c_1, \ldots, c_n$ so that the data fits into the model g. The method involves substituting the data points (x_i, y_i) into the model g. Each evaluation of a point gives a linear equation in terms of the coefficients, that is, to find the missing coefficients we must solve the linear system

$$\begin{cases} c_1 f_1(x_1) + c_2 f_2(x_1) + \ldots + c_n f_n(x_1) = y_1 \\ c_1 f_1(x_2) + c_2 f_2(x_2) + \ldots + c_n f_n(x_2) = y_2 \\ \quad \vdots \qquad\quad \vdots \qquad \ddots \qquad \vdots \qquad\quad \vdots \\ c_1 f_1(x_n) + c_2 f_2(x_n) + \ldots + c_n f_n(x_n) = y_n \end{cases}$$

In the case of a polynomial function, the general template would be $f(x) = c_{n-1}x^{n-1} + \ldots + c_2 x^2 + c_1 x + c_0$. In this case, we would need n points $(x_1, y_1), \ldots, (x_n, y_n)$ to solve the $n \times n$ linear system and interpolate the polynomial. The coefficient matrix for a polynomial interpolation takes on a special form, called a **Vandermonde matrix**:

$$V(\boldsymbol{x}) = \begin{bmatrix} x_1^{n-1} & \cdots & x_1^3 & x_1^2 & x_1 & 1 \\ x_2^{n-1} & \cdots & x_2^3 & x_2^2 & x_2 & 1 \\ x_3^{n-1} & \cdots & x_3^3 & x_3^2 & x_3 & 1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n^{n-1} & \cdots & x_n^3 & x_n^2 & x_n & 1 \end{bmatrix}$$

Note that a Vandermonde matrix depends on the x-coordinates of the interpolation. If $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ is the n-vector of x-coordinates involved in this interpolation, $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ is the n-vector of y-coordinates, and $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ is the yet-unknown n-vector of coefficients, then to solve the interpolation associated to the polynomial f, we must solve the linear system $[V(\boldsymbol{x})]\boldsymbol{c} = \boldsymbol{y}$ for $\boldsymbol{c}$.

Example 2.4.1. Consider the degree-4 polynomial

$$f(x) = c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0$$

with unknown coefficients. If we know

$$f(-1) = 3, \quad f(-0.5) = 0.5625, \quad f(0) = 1, \quad f(0.5) = 3.5625, \quad f(1) = 15,$$

then find the coefficients of f.

Let $\boldsymbol{c} = (c_4, c_3, c_2, c_1, c_0)$ be the coefficient vector for f that we need to find. Each evaluation of the polynomial gives a linear equation with regard to its coefficients, for example, since $f(1) = 15$,

$$(1)^4 c_4 + (1)^3 c_3 + (1)^2 c_2 + (1) c_1 + c_0 = 15.$$

Likewise, $f(0.5) = 3.5625$ gives the linear equation

$$(0.5)^4 c_4 + (0.5)^3 c_3 + (0.5)^2 c_2 + (0.5) c_1 + c_0 = 3.5625.$$

Putting these all together gives the linear system

$$\begin{cases} (-1)^4c_4 + (-1)^3c_3 + (-1)^2c_2 + (-1)c_1 + c_0 = 3 \\ (-0.5)^4c_4 + (-0.5)^3c_3 + (-0.5)^2c_2 + (-0.5)c_1 + c_0 = 0.5625 \\ (0)^4c_4 + (0)^3c_3 + (0)^2c_2 + (0)c_1 + c_0 = 1 \\ (0.5)^4c_4 + (0.5)^3c_3 + (0.5)^2c_2 + (0.5)c_1 + c_0 = 3.5625 \\ (1)^4c_4 + (1)^3c_3 + (1)^2c_2 + (1)c_1 + c_0 = 15 \end{cases} \sim \left[\begin{array}{ccccc|c} (-1)^4 & (-1)^3 & (-1)^2 & -1 & 1 & 3 \\ (-0.5)^4 & (-0.5)^3 & (-0.5)^2 & -0.5 & 1 & 0.5625 \\ (0)^4 & (0)^3 & (0)^2 & 0 & 1 & 1 \\ (0.5)^4 & (0.5)^3 & (0.5)^2 & 0.5 & 1 & 3.5625 \\ (1)^4 & (1)^3 & (1)^2 & 1 & 1 & 15 \end{array}\right]$$

This augmented matrix represents the linear system $[V(\boldsymbol{x})]\boldsymbol{c} = \boldsymbol{y}$ where $\boldsymbol{x} = (-1, -0.5, 0, 0.5, 1)$ and $\boldsymbol{y} = (3, 0.5625, 1, 3.5625, 15)$. Solving this for $\boldsymbol{c}$ gives $\boldsymbol{c} = (5, 4, 3, 2, 1)$. Therefore, $f(x) = 5x^4 + 4x^3 + 3x^2 + 2x + 1$. Details of this calculation can be found in the subsection on Python. ∎

Suppose that we want to fit a rational function to specific data. It is very tempting to use the generic form $f(x) = \dfrac{p(x)}{q(x)}$ where $p(x)$ and $q(x)$ are generic polynomials of degrees n, m, respectively. In order to solve this as a linear system, we plug in points (x_i, y_i) like usual, but we must also clear the denominators. Thus, $y_i = \dfrac{p(x_i)}{q(x_i)}$ becomes $p(x_i) - y_i q(x_i)$, which is linear. Unfortunately, the linear system we construct here is homogeneous and has a unique solution, namely $\mathbf{0}$. The issue here is that in rational functions, we actually have a *false free variable*, that is, while we may believe that the coefficients $p(x)$ and $q(x)$ could be any number, that is actually not true. If all the coefficients of $q(x)$ were 0, then $q(x) = 0$ and the rational function would be undefined. As such, one of the coefficients of $q(x)$ is actually a dependent variable. In particular, if $p(x) = a_m x^m + \ldots + a_1 x + a_0$ and $q(x) = b_n x^n + \ldots + b_1 x + b_0$, then $b_i \neq 0$ for at least one i. Suppose that $b_0 \neq 0$, then

$$f(x) = \frac{a_m x^m + \ldots + a_1 x + a_0}{b_n x^n + \ldots + b_1 x + b_0} = \left(\frac{b_0}{b_0}\right) \frac{\frac{a_m}{b_0} x^m + \ldots + \frac{a_1}{b_0} x + \frac{a_0}{b_0}}{\frac{b_n}{b_0} x^n + \ldots + \frac{b_1}{b_0} x + 1} = \frac{a'_m x^m + \ldots + a'_1 x + a'_0}{b'_n x^n + \ldots + b'_1 x + 1},$$

where $a'_j = \dfrac{a_j}{b_0}$ and $b'_j = \dfrac{b_j}{b_0}$ are free variables. Therefore, when modeling with rational functions, it is common place to assume the constant of the denominator is 1. This will produce a non-homogeneous linear system whose solution provides the coefficients.

Example 2.4.2. Consider the linear fractional

$$f(x) = \frac{a_1 x + a_0}{b_1 x + 1},$$

with unknown coefficients a_1, a_0, b_1. If we know,

$$f(1) = \frac{1}{6}, \qquad f(2) = \frac{3}{11}, \qquad f(3) = \frac{5}{16},$$

then find the coefficients of f.

Each evaluation of the rational function gives an equation with regard to its coefficients, for example, since $f(2) = \dfrac{a_1(2) + a_0}{b_1(2) + 1} = \dfrac{3}{11}$. Clearing the denominators, we have $22a_1 + 11a_0 = 6b_1 + 3$ or $22a_1 + 11a_0 - 6b_1 = 3$. Repeating this process for all the points gives the linear system:

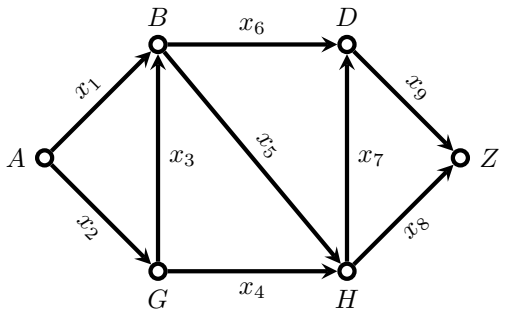
$$\begin{cases} 6a_1 + 6a_0 - b_1 = 1 \\ 22a_1 + 11a_0 - 6b_1 = 3 \\ 48a_1 + 16a_0 - 15b_1 = 5 \end{cases} \sim \left[\begin{array}{ccc|c} 6 & 6 & -1 & 1 \\ 22 & 11 & -6 & 3 \\ 48 & 16 & -15 & 5 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 5 \end{array}\right].$$

Therefore, $f(x) = \dfrac{2x - 1}{5x + 1}$. ∎

2.4.2 Conservation Laws

Systems of linear equations arise naturally when scientists, engineers, or economists study the flow of some quantity through a network. For instance, urban planners and traffic engineers monitor the pattern of traffic flow in a grid of city streets. Electrical engineers calculate current flow through electrical circuits. And economists analyze the distribution of products from manufacturers to consumers through a network of wholesalers and retailers. For many networks, the systems of equations involve hundreds or even thousands of variables and equations.

Example 2.4.3. The following graph illustrated below shows the flow of a certain commodity through a network. Seven thousand commodities are transported from node A to node Z. The exact amount transported through an edge is unknown. Each node in the system produces an equation, using the conversation law that the amount entering a node is equal to the amount exiting the node. This leads to the following linear system, going in alphabetic order, with in-flow on the left and out-flow on the right:



$$\begin{cases} 7000 = x_1 + x_2 \\ x_1 + x_3 = x_5 + x_6 \\ x_6 + x_7 = x_9 \\ x_2 = x_3 + x_4 \\ x_4 + x_5 = x_7 + x_8 \\ x_8 + x_9 = 7000 \end{cases} \sim \begin{cases} x_1 + x_2 = 7000 \\ x_1 + x_3 - x_5 - x_6 = 0 \\ x_6 + x_7 - x_9 = 0 \\ x_2 - x_3 - x_4 = 0 \\ x_4 + x_5 - x_7 - x_8 = 0 \\ x_8 + x_9 = 7000 \end{cases}$$

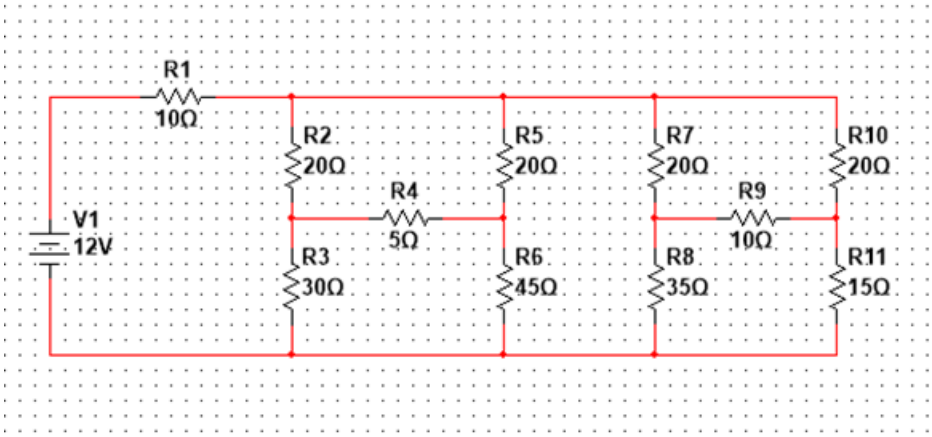
$$\sim \left[\begin{array}{ccccccccc|c} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7000 \\ 1 & 0 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 7000 \end{array}\right] \sim \left[\begin{array}{ccccccccc|c} \boxed{1} & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\ 0 & \boxed{1} & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 7000 \\ 0 & 0 & 0 & \boxed{1} & 1 & 0 & -1 & 0 & 1 & 7000 \\ 0 & 0 & 0 & 0 & 0 & \boxed{1} & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & 1 & 7000 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

This system is underdetermined and must have some free variables, namely x_3, x_5, x_7, and x_9. This accounts for the reality that there are lots of options on how to transport the commodities through the network. In practice, there are additional constraints that would be mentioned here. For example, $x_i \geq 0$ for all i. This can lead to other inequalities, As $x_8 + x_9 = 7000$, if $x_8, x_9 \geq 0$, then this also implies that $x_8, x_9 \leq 7000$. Of course, for all the edges, $x_i \leq 7000$. But certain edges might also have capacities $x_i \leq c_i$ less than 7000 that limit the amount that passes through. These considerations lead to the more advanced problem of *linear programming*. ∎

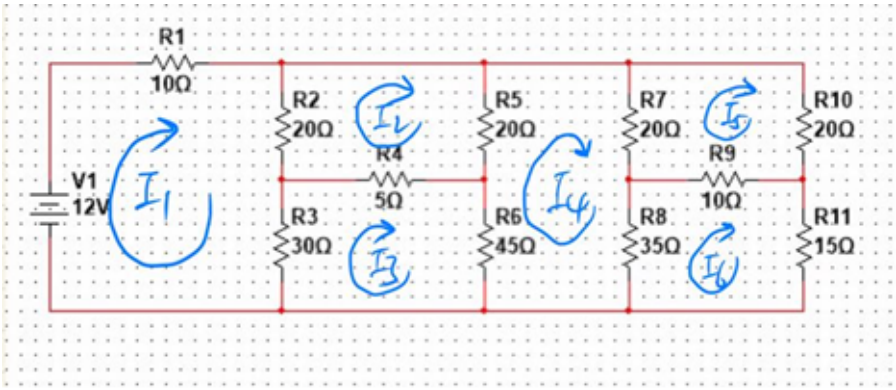
Example 2.4.4. Current flow in a simple electrical network can be described by a system of linear equations. A voltage source such as a battery forces a current of electrons to flow through the network. When the current passes through a resistor (such as a light bulb or motor), some of the voltage is "used up"; by Ohm's law, this "voltage drop" across a resistor is given by

$$RI = V$$

where the voltage V is measured in *volts*, the resistance R is *ohms* (denoted by Ω), and the current flow I in *amperes* (*amps*, for short).



The circuit illustrated below contains several closed loops. The current flowing in these loops are denoted by I_1, I_2, I_3, etc. The designated directions of such **loop currents** are arbitrary. If a current turns out to be negative, then the actual direction of current flow is opposite to that chosen in the figure. If the current direction shown is away from the positive (longer) side of a battery around to the negative (shorter) side, the voltage is positive; otherwise, the voltage is negative. Current flow in a loop is governed by the following rule.



Theorem 2.4.5 (Kirchhoff's Voltage Law)**.** *The algebraic sum of the RI voltage drops in one direction around a loop equals the algebraic sum of the voltage sources in the same direction around the loop, that is,*

$$\sum RI = \sum V.$$

Going one-by-one through each of the six loops produces the following linear system. Note that every time we encounter a source or resistor in a loop we consider all the loops it is apart of with their orientations. Therefore,

$$\begin{cases} 10I_1 + 20(I_1 - I_2) + 30(I_1 - I_3) = 12 \\ 20(I_2 - I_1) + 5(I_2 - I_3) + 20(I_2 - I_4) = 0 \\ 30(I_3 - I_1) + 5(I_3 - I_2) + 45(I_3 - I_4) = 0 \\ 20(I_4 - I_2) + 45(I_4 - I_3) + 20(I_4 - I_5) + 35(I_4 - I_6) = 0 \\ 20(I_5 - I_4) + 10(I_5 - I_6) + 20I_5 = 0 \\ 35(I_6 - I_4) + 10(I_6 - I_5) + 15I_6 = 0 \end{cases} \sim \begin{cases} 60I_1 - 20I_2 - 30I_3 = 12 \\ -20I_1 + 45I_2 - 5I_3 - 20I_4 = 0 \\ -30I_1 - 5I_2 + 80I_3 - 45I_4 = 0 \\ -20I_2 - 45I_3 + 120I_4 - 20I_5 - 35I_6 = 0 \\ -20I_4 + 50I_5 - 10I_6 = 0 \\ -35I_4 - 10I_5 + 60I_6 = 0 \end{cases}$$

Using `np.linalg.solve(A,b)`, where $A = \begin{bmatrix} 60 & -20 & -30 & 0 & 0 & 0 \\ -20 & 45 & -5 & -20 & 0 & 0 \\ -30 & -5 & 80 & -45 & 0 & 0 \\ 0 & -20 & -45 & 120 & -20 & -35 \\ 0 & 0 & 0 & -20 & 50 & -10 \\ 0 & 0 & 0 & -35 & -10 & 60 \end{bmatrix}$ and $\boldsymbol{b} = \begin{bmatrix} 12 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, we get $\boldsymbol{x} = (0.43427136, 0.29487437, 0.2719598, 0.16120603, -0.08060302, 0.08060302)$. ■

Python Programming

One can form the matrix Vandermonde matrix $V(\boldsymbol{x})$ in Python using `V = np.vander(x)`, where $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ is the vector of x-coordinates for the points we are interpolating.

Likewise, if we are given $\boldsymbol{y}$, the vector of y-coordinates, then we can solve for the coefficient vector $\boldsymbol{c}$ in the linear system $[V(\boldsymbol{x})]\boldsymbol{c} = \boldsymbol{y}$ using `c = np.linalg.solve(A,y)`.

Example 2.4.6. The code:

```
import numpy as np

points = np.array([[-1,3],         # points to be interpolated
                   [-0.5, 0.5625],
                   [0,1],
                   [0.5,3.5625],
                   [1,15]])
x = points[:,0]                    # just the x-coordinates
y= points[:,1]                     # just the y-coordinates

print("x = ",x)
print("y = ", y, "\n")

V = np.vander(x)           # create the Vandermonde matrix, that only depends on the x-
    coordinates
print("V = ", V, "\n")

c = np.linalg.solve(V,y)  # solve the linear system associated to this polynomial-data-fit
print("c = ", c)

f = np.poly1d(c)           # np.poly1d does all the work of solving a polynomial-data-fit
print(f)

import matplotlib.pyplot as plt # matplotlib to print out our results in a graphic form
plt.style.use("seaborn-v0_8-whitegrid")

plt.ion()
domain = np.linspace(-3, 3, 100)
plt.xlim(-3,3)
plt.ylim(-1,25)
plt.scatter(x, y)
plt.plot(domain,f(domain))
plt.show()
```

solves the interpolation problem in Example 2.4.1 by produces a vector of coefficients in descending order. ■

Alternatively, the function `np.polyfit(x, y, n)` will also produce the coefficient vector for a polynomial interpolation in descending order, where $\boldsymbol{x}$ is the vector of x-coordinates, $\boldsymbol{y}$ is the corresponding vector of y-coordinates, and n is the degree of the polynomial.

Exercises

(Go to Solutions)

♠ 1. Michael is going to the store and offers to buy groceries for his friends. His friend, Bailey, says she needs 3 dozen eggs, 1 loaf of bread, and 7 apples. Another friend, Gaston, says he needs 5 dozen eggs, 2 loaves of bread, and 1 apple. His other friend, Janessa, says she needs 1 dozen eggs and 10 apples. After buying everything for his friends, he tells each of them what they should Venmo him. He tells Bailey her total is \$15.49, Gaston his total is \$18.32, and Janessa her total is \$10.29. How much does a loaf of bread, a dozen eggs, and a single apple cost?

For Exercises 2–2, interpolate the function $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ through the given points.

♠ 2. $(1, 1)$, $(2, 3)$, $(4, 0)$, $(-1, -5)$

For Exercises 3–3, interpolate the function $f(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ through the given points.

3. $(-2, 231)$, $(-1, 19)$, $(0, 5)$, $(1, 9)$, $(2, 19)$, $(3, -169)$

For Exercises 4–4, interpolate the function $f(x) = \dfrac{a_2x^2 + a_1x + a_0}{b_2x^2 + b_1x + 1}$ through the given points.

4. $\left(-2, \frac{12}{11}\right)$, $\left(-1, \frac{3}{2}\right)$, $(0, 2)$, $(1, 0)$, $(2, 0)$

5. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/linearapp.ipynb

"If the world's a veil of tears, Smile till rainbows span it." – Lucy Larcom

2.5 Vector Equations

Linear systems can take on many forms. While often represented as lists of linear equations with common variables, we will see that they show up in other ways too. This section focuses on vector equations that are equivalent to linear systems.

2.5.1 Equations and Linear Combinations

Linear combinations are the bread and butter of vectors. Algebraically speaking, this is what we do with vectors, that is, we combine some vectors to construct new vectors. While it is simple to take a list of vector, combine them, and see what pops out, the converse is not as simple. Given a list of vectors $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n \in \mathbb{R}^m$, can we determine if $\boldsymbol{b} \in \mathbb{R}^m$ is a linear combination of these vectors? In other words, we seek to solve the vector equation:

$$x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \ldots + x_n\boldsymbol{a}_n = \boldsymbol{b}. \tag{2.5.1}$$

In Equation (2.5.1), note that the vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n, \boldsymbol{b}$ are fixed and the variables are the scalars $x_1, x_2, \ldots, x_n \in \mathbb{R}$.

To solve the vector equation (2.5.1), suppose that the ith component of $\boldsymbol{a}_j$ is denoted a_{ij}, that is,

$$\boldsymbol{a}_j = \begin{bmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{bmatrix}.$$

Likewise, let the jth component of $\boldsymbol{b}$ be b_j. Then Equation (2.5.1) becomes

$$\begin{aligned}
x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \ldots + x_n\boldsymbol{a}_n &= \boldsymbol{b} \\
x_1\begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2\begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \ldots + x_n\begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} &= \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \\
\begin{bmatrix} a_{11}x_1 \\ a_{21}x_1 \\ \vdots \\ a_{m1}x_1 \end{bmatrix} + \begin{bmatrix} a_{12}x_2 \\ a_{22}x_2 \\ \vdots \\ a_{m2}x_2 \end{bmatrix} + \ldots + \begin{bmatrix} a_{1n}x_n \\ a_{2n}x_n \\ \vdots \\ a_{mn}x_n \end{bmatrix} &= \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \\
\begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \end{bmatrix} &= \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}
\end{aligned}$$

Examining this last equality of vectors reveals a system of linear equations, namely

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \quad\vdots \qquad\quad \vdots \qquad \ddots \qquad \vdots \qquad\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m, \end{cases} \tag{2.5.2}$$

whose augmented matrix is

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} & b_m \end{array}\right].$$

These two problems have the same solution set.

Theorem 2.5.1. *Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_m) \in \mathbb{R}^m$. Then $\boldsymbol{x}$ is a solution to the vector equation* (2.5.1) *if and only if $\boldsymbol{x}$ is a solution to the linear system* (2.5.2).

Example 2.5.2. Let $\boldsymbol{a}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$, $\boldsymbol{a}_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$, and $\boldsymbol{a}_3 = \begin{bmatrix} 3 \\ 0 \\ 5 \\ 7 \end{bmatrix}$. Let $\boldsymbol{b} = \begin{bmatrix} -7 \\ 4 \\ -4 \\ -9 \end{bmatrix}$. Is $\boldsymbol{b}$ a linear combination of $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3$?

To solve the vector equation,

$$x_1 \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + x_2 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_3 \begin{bmatrix} 3 \\ 0 \\ 5 \\ 7 \end{bmatrix} = \begin{bmatrix} -7 \\ 4 \\ -4 \\ -9 \end{bmatrix},$$

we solve the linear system whose augmented matrix is

$$\left[\begin{array}{ccc|c} 1 & 1 & 3 & -7 \\ 2 & 0 & 0 & 4 \\ 3 & 0 & 5 & -4 \\ 4 & 1 & 7 & -9 \end{array}\right].$$

We next row reduce the matrix via the elementary row operations:

$$\left[\begin{array}{ccc|c} \boxed{1} & 1 & 3 & -7 \\ 2 \overset{-2}{} & 0 \overset{-2}{} & 0 \overset{-6}{} & 4 \overset{14}{} \\ 3 \overset{-3}{} & 0 \overset{-3}{} & 5 \overset{-9}{} & -4 \overset{21}{} \\ 4 \overset{-4}{} & 1 \overset{-4}{} & 7 \overset{-12}{} & -9 \overset{28}{} \end{array}\right] \begin{array}{l} \\ (\text{Row } 2 - 2 \text{ Row } 1) \\ (\text{Row } 3 - 3 \text{ Row } 1) \\ (\text{Row } 4 - 4 \text{ Row } 1) \end{array} \sim \left[\begin{array}{ccc|c} \boxed{1} & 1 & 3 & -7 \\ 0 & \boxed{-2} & -6 & 18 \\ 0 & -3 & -4 & 17 \\ 0 & -3 \overset{3}{} & -5 \overset{4}{} & 19 \overset{-17}{} \end{array}\right] \begin{array}{l} \\ \left(-\frac{1}{2}\text{Row } 2\right) \\ \\ (\text{Row } 4 - \text{Row } 3) \end{array}$$

$$\sim \left[\begin{array}{cccc|c} \boxed{1} & 1 & 3 & & -7 \\ 0 & \boxed{1} & 3 & & -9 \\ 0 & -3 & -4 & & 17 \\ 0 & 0 & -1 & & 2 \end{array}\right] \begin{array}{l} \\ \\ \text{(Row 3 + 3 Row 2)} \\ \text{(}-\text{Row 4)} \end{array} \sim \left[\begin{array}{cccc|c} \boxed{1} & 1 & 3 & & -7 \\ 0 & \boxed{1} & 3 & & -9 \\ 0 & 0 & \boxed{5} & & -10 \\ 0 & 0 & 1 & & -2 \end{array}\right]$$

$$\sim \left[\begin{array}{cccc|c} \boxed{1} & 1 & 3 & & -7 \\ 0 & \boxed{1} & 3 & & -9 \\ 0 & 0 & \boxed{1} & & -2 \\ 0 & 0 & 5 & & -10 \end{array}\right] \text{(Row 4 – 5 Row 3)} \sim \left[\begin{array}{cccc|c} \boxed{1} & 1 & 3 & & -7 \\ 0 & \boxed{1} & 3 & & -9 \\ 0 & 0 & \boxed{1} & & -2 \\ 0 & 0 & 0 & & 0 \end{array}\right]$$

This last matrix is in echelon form. We will continue to row reduced echelon form.

$$\sim \left[\begin{array}{cccc|c} \boxed{1} & 1 & 3 & -3 & -7 \\ 0 & \boxed{1} & 3 & -3 & -9 \\ 0 & 0 & \boxed{1} & & -2 \\ 0 & 0 & 0 & & 0 \end{array}\right] \begin{array}{l} \text{(Row 1 – 3 Row 3)} \\ \text{(Row 2 – 3 Row 3)} \\ \\ \end{array} \sim \left[\begin{array}{cccc|c} \boxed{1} & 1 & -1 & 0 & 3 \\ 0 & \boxed{1} & 0 & & -3 \\ 0 & 0 & \boxed{1} & & -2 \\ 0 & 0 & 0 & & 0 \end{array}\right] \text{(Row 1 – Row 2)} \sim \left[\begin{array}{cccc|c} \boxed{1} & 0 & 0 & & 2 \\ 0 & \boxed{1} & 0 & & -3 \\ 0 & 0 & \boxed{1} & & -2 \\ 0 & 0 & 0 & & 0 \end{array}\right]$$

Therefore, the solution is $(2, -3, -2)$. In fact, we see that

$$2\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} - 3\begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} - 2\begin{bmatrix} 3 \\ 0 \\ 5 \\ 7 \end{bmatrix} = \begin{bmatrix} -7 \\ 4 \\ -4 \\ -9 \end{bmatrix}. \qquad \blacksquare$$

Example 2.5.3. Balancing chemical equations is important in science because it allows for an understanding of stoichiometric relations in reactions. A balanced chemical equation allows for calculations necessary to for yields and preparation of solutions. In general, a chemical equation is described by reactants converting to products. Basic conservation laws state that the number of atoms in the reactants is equal to the number of atoms in the products. Consider the chemical reaction:

$$CH_4 + O_2 \to CO_2 + H_2O$$

Here, C, H, and O denote the elements carbon, hydrogen, and oxygen. In this chemical reaction some number of molecules of methane (CH_4) and oxygen (O_2) are combined to produce some number of molecules of carbon dioxide (CO_2) and water (H_2O). Our goal in balancing the chemical equation is to determine how many of each molecule are needed to balance it. As there are three elements in the reaction, we represent each chemical compound as a 3-vector (C, H, O). As there are four compounds, we use the variables $x_1, \ldots, x_4$ to denote the unknown number of molecules of each compound. Thus, the chemical reaction becomes the vector equation:

$$x_1\begin{bmatrix} 1 \\ 4 \\ 0 \end{bmatrix} + x_2\begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} = x_3\begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} + x_4\begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} \sim \begin{cases} x_1 = x_3 \\ 4x_1 = 2x_4 \\ 2x_2 = 2x_3 + x_4 \end{cases} \sim \begin{cases} x_1 - x_3 = 0 \\ 4x_1 - 2x_4 = 0 \\ 2x_2 - 2x_3 - x_4 = 0 \end{cases}$$

$$\sim \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 4 & 0 & 0 & -2 & 0 \\ 0 & 2 & -2 & -1 & 0 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & 0 & 0 & -\frac{1}{2} & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -\frac{1}{2} & 0 \end{array}\right]$$

Therefore, the general solution is $\boldsymbol{x} = t(0.5, 1, 0.5, 1)$. To interpret this, if we have two moles of water $t = 2$, then we need one mole of methane, two moles of oxygen, and one mole of carbon dioxide to balance this chemical reaction. ∎

2.5.2 The Span of Vectors

Definition 2.5.4. Given vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n \in \mathbb{R}^m$, the **(linear) span**[i] of $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$, denoted $\text{Span}\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$, is the set of all linear combinations of $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ inside of $\mathbb{R}^m$. we say that a subset $S \subseteq \mathbb{R}^m$ is **spanned** by $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ if $S = \text{Span}\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ and that $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is a **spanning set** for S.

Asking whether a vector $\boldsymbol{b} \in \text{Span}\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$ is equivalent to asking whether there exists a solution $\boldsymbol{b}$ to the vector equation

$$x_1\boldsymbol{a}_1 + \ldots + x_n\boldsymbol{a}_n = \boldsymbol{b}.$$

Example 2.5.5. Let $\boldsymbol{a}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix}$, $\boldsymbol{a}_2 = \begin{bmatrix} 2 \\ 1 \\ 3 \\ -1 \end{bmatrix}$, $\boldsymbol{a}_3 = \begin{bmatrix} 0 \\ 2 \\ 1 \\ 4 \end{bmatrix}$, and $\boldsymbol{b} = \begin{bmatrix} 7 \\ 0 \\ 2 \\ 0 \end{bmatrix}$. Is $\boldsymbol{b} \in \text{Span}\{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3\}$?

As explained above, answering the question is equivalent to solving the vector equation $x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + x_3\boldsymbol{a}_3 = \boldsymbol{b}$. Likewise, this vector equation can be solved by row reduction on the augmented matrix $\left[\begin{array}{ccc|c} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \boldsymbol{a}_3 & \boldsymbol{b} \end{array}\right]$:

$$\left[\begin{array}{ccc|c} \boxed{1} & 2 & 0 & 7 \\ 0 & 1 & 2 & 0 \\ -1 \overset{+1}{} & 3 \overset{+2}{} & 1 \overset{+0}{} & 2 \overset{+7}{} \\ 2 \overset{-2}{} & -1 \overset{-4}{} & 4 \overset{-0}{} & 0 \overset{-14}{} \end{array}\right] \begin{array}{l} \\ \\ (\text{Row } 3 + \text{Row } 1) \\ (\text{Row } 4 - 2\,\text{Row } 1) \end{array} \sim \left[\begin{array}{ccc|c} \boxed{1} & 2 \overset{-2}{} & 0 \overset{-4}{} & 7 \overset{-0}{} \\ 0 & \boxed{1} & 2 & 0 \\ 0 & 5 \overset{-5}{} & 1 \overset{-10}{} & 9 \overset{-0}{} \\ 0 & -5 \overset{+5}{} & 4 \overset{+10}{} & -14 \overset{+0}{} \end{array}\right] \begin{array}{l} (\text{Row } 1 - 2\,\text{Row } 2) \\ \\ (\text{Row } 3 - 5\,\text{Row } 2) \\ (\text{Row } 4 + 5\,\text{Row } 2) \end{array}$$

$$\sim \left[\begin{array}{ccc|c} \boxed{1} & 0 & -4 \overset{+4}{} & 7 \overset{-4}{} \\ 0 & \boxed{1} & 2 \overset{-2}{} & 0 \overset{+2}{} \\ 0 & 0 & \boxed{-9} & 9 \\ 0 & 0 & 14 \overset{-14}{} & -14 \overset{+14}{} \end{array}\right] \begin{array}{l} (\text{Row } 1 - \frac{4}{9}\text{Row } 3) \\ (\text{Row } 2 + \frac{2}{9}\text{Row } 3) \\ (-\frac{1}{9}\text{Row } 3) \\ (\text{Row } 4 + \frac{14}{9}\text{Row } 3) \end{array} \sim \left[\begin{array}{ccc|c} \boxed{1} & 0 & 0 & 3 \\ 0 & \boxed{1} & 0 & 2 \\ 0 & 0 & \boxed{1} & -1 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

[i] We can by analog define the **affine span** (or an **affine set**) of a set of vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$, denoted $\text{Aff}\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$, as the set of all affine combinations of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$. Recall that an affine combination is a linear combination such that the coefficients sum to 1. Likewise, we can define the **convex span** (or **convex hull**) of a set of vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$, denoted $\text{Con}\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$, as the set of all convex conbinations of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$. Recall that a convex combination is an affine combination such that all coefficients are nonnegative.

Therefore, the solution is $x_1 = 3$, $x_2 = 2$, and $x_3 = -1$, that is,

$$\boldsymbol{b} = 3\boldsymbol{a}_1 + 2\boldsymbol{a}_2 - \boldsymbol{a}_3 = 3\begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix} + 2\begin{bmatrix} 2 \\ 1 \\ 3 \\ -1 \end{bmatrix} - \begin{bmatrix} 0 \\ 2 \\ 1 \\ 4 \end{bmatrix} = \begin{bmatrix} 7 \\ 0 \\ 2 \\ 0 \end{bmatrix}. \qquad \blacksquare$$

Example 2.5.6. Let $\boldsymbol{a}_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 2 \\ -1 \end{bmatrix}$, $\boldsymbol{a}_2 = \begin{bmatrix} 0 \\ 1 \\ 2 \\ -1 \\ 1 \end{bmatrix}$, $\boldsymbol{a}_3 = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 0 \\ 1 \end{bmatrix}$, and $\boldsymbol{b} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 4 \\ 0 \end{bmatrix}$. Is $\boldsymbol{b} \in \text{Span}\{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3\}$?

Like above, we row reduce the augmented matrix $\left[\begin{array}{ccc|c} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \boldsymbol{a}_3 & \boldsymbol{b} \end{array}\right]$:

$$\left[\begin{array}{ccc|c} 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 2 & 5 & 2 & 1 \\ 1 & 0 & 4 & 2 \\ 1 & 1 & 0 & 0 \end{array}\right] \begin{array}{l} \\ \\ \text{(Row 3 } - \text{ Row 1)} \\ \text{(Row 4 } - 2 \text{ Row 1)} \\ \text{(Row 5 } + \text{ Row 1)} \end{array} \sim \left[\begin{array}{ccc|c} \boxed{1} & 0 & 1 & 1 \\ 0 & \boxed{1} & 2 & 1 \\ 0 & 2 & 4 & 1 \\ 0 & -1 & -2 & 2 \\ 0 & 1 & 2 & 1 \end{array}\right] \begin{array}{l} \\ \\ \text{(Row 3 } - 2 \text{ Row 2)} \\ \text{(Row 4 } + \text{ Row 2)} \\ \text{(Row 5 } - \text{ Row 2)} \end{array} \sim \left[\begin{array}{ccc|c} \boxed{1} & 0 & 1 & 1 \\ 0 & \boxed{1} & 2 & 1 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{array}\right],$$

which is now in reduced row echelon form. The third row represents the equation $0 = 1$, which implies that the system is inconsistent. Therefore, $\boldsymbol{b} \notin \text{Span}\{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3\}$. $\blacksquare$

Python Programming

We have seen in Section 2.5.2 that determining whether $\boldsymbol{b} \in \text{Span}\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n\}$ or not is equivalent to whether the vector equation

$$x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \ldots + x_n\boldsymbol{a}_n = \boldsymbol{b}$$

is consistent or inconsistent, which we saw in Section 2.5.1 is in turn equivalent to the consistency of the linear system

$$\left[\begin{array}{cccc|c} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n & \boldsymbol{b} \end{array}\right].$$

If the linear system is consistent, then any solution to this linear system provides the coefficients of the linear combination.

Using `NumPy`, we can check this by the command `np.linalg.solve(A,b)`, as we saw in Section 2.3, but as was seen there, the function `np.linalg.solve(A,b)` only works if A is a square matrix. In many cases, regarding the question of whether $\boldsymbol{b}$ is spanned by $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$, the matrix $A = \left[\begin{array}{cccc} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n \end{array}\right]$ is not square. To avoid this potential error, we may use `np.linalg.lstsq` instead. The function `lstsq` is short for *least squares*, which we learn in Chapter 4. Without attempting to develop the theory of least squares here, in short, `np.linalg.lstsq` can handle matrices of any shape, but if the linear system $\left[\begin{array}{c|c} A & \boldsymbol{b} \end{array}\right]$ has a solution, then the least squares solution will match the linear solution $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$. Hence, when $\left[\begin{array}{c|c} A & \boldsymbol{b} \end{array}\right]$ is consistent, `np.linalg.lstsq(A,b)` will produce the solution.

On the other hand, if $\left[\begin{array}{c|c} A & \boldsymbol{b} \end{array}\right]$ is inconsistent, `np.linalg.lstsq(A,b)` will still produce a "solution"[ii], just not a solution to the linear system. As such, users must be prepared to check this potential solution.

If $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ is the least squares solution to $\left[\begin{array}{c|c} A & \boldsymbol{b} \end{array}\right]$, then we can check if it is the linear solution by computing the method explained in Section 2.1.

[ii] Note that the least squares method always produces something, which will match the linear solution when the linear system is consistent. When the linear system is inconsistent, the least squares solution will be something else, because there is no linear solution so every vector is necessariliy something else.

Exercises

(Go to Solutions)

For Exercises 1-8, express the given system of equations as a vector equation or vice versa.

1. $\begin{cases} x_1 + 7x_2 = 16 \\ 8x_1 - 3x_2 = 6 \end{cases}$

♠ 2. $\begin{cases} 3x_1 + 7x_2 - 5x_3 = -5 \\ -x_1 - 2x_2 + 6x_3 = 4 \end{cases}$

♠ 3. $\begin{cases} -2x_1 + 6x_2 - 11x_3 = 6 \\ x_1 - 2x_2 + 5x_3 = -3 \\ 3x_1 - 19x_2 + 31x_3 = -17 \end{cases}$

4. $\begin{cases} x_1 + 2x_2 + 2x_3 = 1 \\ 3x_1 + x_2 = 4 \\ 5x_1 + 2x_2 + x_3 = 3 \end{cases}$

5. $\begin{cases} x_1 + 3x_2 = 7 \\ 7x_1 + 6x_2 - x_3 = 8 \\ 3x_1 + x_2 + 2x_3 = 3 \end{cases}$

6. $\begin{cases} 2x_1 + 4x_2 - x_3 = 8 \\ x_1 - 6x_2 + 3x_3 = 12 \\ -3x_1 + 5x_2 + 6x_3 = 9 \end{cases}$

7. $x_1 \begin{bmatrix} 9 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 3 \end{bmatrix} + x_3 \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 7 \\ 5 \end{bmatrix}$

8. $x_1 \begin{bmatrix} 7 \\ -3 \\ 4 \\ 2 \end{bmatrix} + x_2 \begin{bmatrix} -3 \\ -1 \\ -1 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 6 \\ 3 \\ -8 \\ 5 \end{bmatrix} + x_4 \begin{bmatrix} 9 \\ 3 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 17 \\ 4 \\ -12 \\ 8 \end{bmatrix}$

For Exercises 9-13, determine if $\boldsymbol{b}$ is a linear combination of the other vectors $\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots\}$. If so, write $\boldsymbol{b}$ as a linear combination. The set of vectors $\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots\}$ will be listed first, followed by the vector $\boldsymbol{b}$. Answers may vary.

9. $\left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 3 \\ -1 \end{bmatrix} \right\}, \begin{bmatrix} 4 \\ 1 \end{bmatrix}$

♠ 10. $\left\{ \begin{bmatrix} -1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ -2 \end{bmatrix} \right\}, \begin{bmatrix} 1 \\ 5 \end{bmatrix}$

♠ 11. $\left\{ \begin{bmatrix} -3 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} \right\}, \begin{bmatrix} 5 \\ 4 \\ 3 \end{bmatrix}$

♠ 12. $\left\{ \begin{bmatrix} 3 \\ -2 \\ 1 \end{bmatrix}, \begin{bmatrix} -2 \\ 3 \\ -3 \end{bmatrix} \right\}, \begin{bmatrix} 1 \\ 6 \\ -9 \end{bmatrix}$

13. $\left\{ \begin{bmatrix} 1 \\ -1 \\ 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ -1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 2 \\ 2 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \\ 2 \\ 2 \end{bmatrix} \right\}, \begin{bmatrix} 6 \\ 3 \\ 14 \\ 8 \end{bmatrix}$

For Exercises 14-16, determine if $\boldsymbol{b} \in \text{Span}\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots\}$. If so, write $\boldsymbol{b}$ as a linear combination of the $\boldsymbol{a}_i$. The set of vectors $\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots\}$ will be listed first, followed by the vector $\boldsymbol{b}$. Answers may vary.

14. $\left\{ \begin{bmatrix} -4 \\ 9 \\ 6 \end{bmatrix}, \begin{bmatrix} 10 \\ 6 \\ 4 \end{bmatrix}, \begin{bmatrix} 7 \\ -12 \\ -8 \end{bmatrix} \right\}, \begin{bmatrix} 12 \\ 4 \\ -6 \end{bmatrix}$

♠ 15. $\left\{ \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}, \begin{bmatrix} -2 \\ -1 \\ -2 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix} \right\}, \begin{bmatrix} -4 \\ -1 \\ -2 \end{bmatrix}$

16. $\left\{ \begin{bmatrix} 1 \\ 2 \\ -1 \\ 3 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 5 \\ 4 \\ -8 \\ 3 \end{bmatrix} \right\}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

17. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/span.ipynb

"Some painters transform the sun into a yellow spot, others transform a yellow spot into the sun."
– Pablo Picasso

2.6 Matrix Equations

In addition to the vector equation seen in Section 2.5, we introduce another problem equivalent to a linear system, this time involving the product of a matrix by a vector. This leads naturally to the discussion of the column space of a matrix and matrix transformations.

2.6.1 Matrix-Vector Multiplication

Definition 2.6.1. Let A be an $m \times n$ matrix and let $\boldsymbol{x} \in \mathbb{R}^n$. Let the **column vectors** of A be $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n \in \mathbb{R}^n$, that is, $A = \left[\begin{array}{cccc} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n \end{array}\right]$. Then the **product** $A\boldsymbol{x}$ is the linear combination of the column vectors of A with coefficients corresponding to the entries of $\boldsymbol{x}$, that is,

$$A\boldsymbol{x} = \left[\begin{array}{cccc} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array}\right] = x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \ldots + x_n\boldsymbol{a}_n. \tag{2.6.1}$$

Note that $A\boldsymbol{x}$ is only defined if the number of columns of A is equal to the number of entries of $\boldsymbol{x}$.

Example 2.6.2.

(a) $$\left[\begin{array}{ccc} 1 & 2 & -3 \\ 0 & -2 & 5 \end{array}\right] \left[\begin{array}{c} 3 \\ 0 \\ 2 \end{array}\right] = 3\left[\begin{array}{c} 1 \\ 0 \end{array}\right] + 0\left[\begin{array}{c} 2 \\ -2 \end{array}\right] + 2\left[\begin{array}{c} -3 \\ 5 \end{array}\right] = \left[\begin{array}{c} 3 \\ 0 \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \end{array}\right] + \left[\begin{array}{c} -6 \\ 10 \end{array}\right] = \left[\begin{array}{c} -3 \\ 10 \end{array}\right].$$

(b) $$\left[\begin{array}{cc} 2 & 0 \\ -6 & 1 \\ 3 & 2 \end{array}\right] \left[\begin{array}{c} 3 \\ 5 \end{array}\right] = 3\left[\begin{array}{c} 2 \\ -6 \\ 3 \end{array}\right] + 5\left[\begin{array}{c} 0 \\ 1 \\ 2 \end{array}\right] = \left[\begin{array}{c} 6 \\ -18 \\ 9 \end{array}\right] + \left[\begin{array}{c} 0 \\ 5 \\ 10 \end{array}\right] = \left[\begin{array}{c} 6 \\ -13 \\ 19 \end{array}\right].$$

■

Theorem 2.6.3. *If A is an $m \times n$ matrix, with column vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, and if $\boldsymbol{b} \in \mathbb{R}^m$, the matrix equation*

$$A\boldsymbol{x} = \boldsymbol{b} \tag{2.6.2}$$

has the same solution set as the vector equation

$$x_1\boldsymbol{a}_1 + \ldots + x_n\boldsymbol{a}_n = \boldsymbol{b}$$

which, in turn, has the same solution set as the system of linear equations whose augmented matrix is

$$\left[\begin{array}{ccc|c} \boldsymbol{a}_1 & \ldots & \boldsymbol{a}_n & \boldsymbol{b} \end{array}\right].$$

Example 2.6.4. The solution set to the system of equation

$$\begin{cases} x_1 + 3x_2 - 7x_3 = 5 \\ \quad - 2x_2 + 11x_3 = 3 \end{cases}$$

is same as the solution set of the vector equation

$$x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 3 \\ -2 \end{bmatrix} + x_3 \begin{bmatrix} -7 \\ 11 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix},$$

which in turn has the same solution set as the matrix equation

$$\begin{bmatrix} 1 & 3 & -7 \\ 0 & -2 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix}. \qquad \blacksquare$$

Example 2.6.5. Solve the matrix equation $\begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 1 \\ 3 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 8 \\ 3 \end{bmatrix}$.

We note that solving this matrix equation is equivalent to solving the linear system associated to the augmented matrix

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 9 \\ 2 & -1 & 1 & 8 \\ 3 & 0 & -1 & 3 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 3 \end{array}\right].$$

Therefore, the solution to the matrix equation is $\boldsymbol{x} = (2, -1, 3)$. Note that

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -1 & 1 \\ 3 & 0 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} = 2 \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} - \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix} + 3 \begin{bmatrix} 3 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} + \begin{bmatrix} -2 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 9 \\ 3 \\ -3 \end{bmatrix} = \begin{bmatrix} 9 \\ 8 \\ 3 \end{bmatrix}. \quad \blacksquare$$

Corollary 2.6.6. *The equation $A\boldsymbol{x} = \boldsymbol{b}$ has a solution if and only if $\boldsymbol{b}$ is a linear combination of the column vectors of A.*

Let A be an $m \times n$ matrix with column vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$. Let $A = \begin{bmatrix} a_{ij} \end{bmatrix}$, that is, let a_{ij} denote the entry of A in the (i, j) position. Let $\boldsymbol{x} \in \mathbb{R}^n$. Then

$$A\boldsymbol{x} = \begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = x_1\boldsymbol{a}_1 + \ldots + x_n\boldsymbol{a}_n = x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \ldots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} x_1a_{11} \\ x_1a_{21} \\ \vdots \\ x_1a_{m1} \end{bmatrix} + \begin{bmatrix} x_2a_{12} \\ x_2a_{22} \\ \vdots \\ x_2a_{m2} \end{bmatrix} + \ldots + \begin{bmatrix} x_na_{1n} \\ x_na_{2n} \\ \vdots \\ x_na_{mn} \end{bmatrix} = \begin{bmatrix} x_1a_{11} + x_2a_{12} + \ldots + x_na_{1n} \\ x_1a_{21} + x_2a_{22} + \ldots + x_na_{2n} \\ \vdots \\ x_1a_{m1} + x_2a_{m2} + \ldots + x_na_{mn} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \cdot \boldsymbol{x} \\ \boldsymbol{b}_2 \cdot \boldsymbol{x} \\ \vdots \\ \boldsymbol{b}_m \cdot \boldsymbol{x} \end{bmatrix},$$

where $\boldsymbol{b}_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,n})$ denotes the ith row vector of A. That is

$$A\boldsymbol{x} = \begin{bmatrix} x_1a_{11} + x_2a_{12} + \ldots + x_na_{1n} \\ x_1a_{21} + x_2a_{22} + \ldots + x_na_{2n} \\ \vdots \\ x_1a_{m1} + x_2a_{m2} + \ldots + x_na_{mn} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \vdots \\ \boldsymbol{b}_m \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} \boldsymbol{b}_1 \cdot \boldsymbol{x} \\ \boldsymbol{b}_2 \cdot \boldsymbol{x} \\ \vdots \\ \boldsymbol{b}_m \cdot \boldsymbol{x} \end{bmatrix}. \tag{2.6.3}$$

This formula often is easier to compute than the original definition of $A\boldsymbol{x}$.

2.6.2 Column Space

Definition 2.6.7. Let $\mathrm{Col}(A)$ denote the set of all linear combinations of column vectors of A, which is called the **column space** of A. In particular, if $A = \begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n \end{bmatrix}$, then $\mathrm{Col}(A) = \mathrm{Span}\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n\}$.

According to Corollary 2.6.6, the equation $A\boldsymbol{x} = \boldsymbol{b}$ is consistent if and only if $\boldsymbol{b} \in \mathrm{Col}(A)$.

Example 2.6.8. Let $A = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 4 & 4 \\ 3 & 5 & 7 \end{bmatrix}$. Compute the column space of A.

The quick response to this example would be to find a spanning set for the column space of A, but, by definition, a spanning set for a column space is never mysterious. Note that

$$\mathrm{Col}(A) = \mathrm{Span}\left\{ \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix} \right\}.$$

While we have technically computed the column space, that is, we have found a description of the set of vectors, it does not leave the decision question of whether a generic vector $\boldsymbol{b} = (b_1, b_2, b_3)$ belongs to $\mathrm{Col}(A)$ well answered, which is really what we want to be able to do. To answer this question, we row reduce the augmented matrix

$$\left[\begin{array}{ccc|c} 1 & 3 & 1 & b_1 \\ 2 & 4 & 4 & b_2 \\ 3 & 5 & 7 & b_3 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 3 & 1 & b_1 \\ 0 & -2 & 2 & b_2 - 2b_1 \\ 0 & -4 & 4 & b_3 - 3b_1 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 3 & 1 & b_1 \\ 0 & 1 & -1 & b_1 - \frac{1}{2}b_2 \\ 0 & 1 & -1 & \frac{1}{4}(3b_1 - b_3) \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 3 & 1 & b_1 \\ 0 & 1 & -1 & b_1 - \frac{1}{2}b_2 \\ 0 & 0 & 0 & -\frac{1}{4}b_1 + \frac{1}{2}b_2 - \frac{1}{4}b_3 \end{array}\right]$$

Therefore, the above equation is consistent if and only if $-\frac{1}{4}b_1 + \frac{1}{2}b_2 - \frac{1}{4}b_3 = 0$. Now, there exists plenty of choices of $\boldsymbol{b}$ such that this equation does not hold. Hence, $A\boldsymbol{x} = \boldsymbol{b}$ is not consistent for all $\boldsymbol{b}$.

$$\mathrm{Col}(A) = \{\boldsymbol{b} \in \mathbb{R}^3 \mid b_1 - 2b_2 + b_3 = 0\}.$$

■

Generalizing the principles seen in the previous example, for any $n \times n$ matrix A, the equation $A\boldsymbol{x} = \boldsymbol{b}$ has a solution for all $\boldsymbol{b}$ if and only if A has a pivot in each row, in which case the solution must be unique.

2.6.3 Matrix Transformations

Let A be an $(m \times n)$ matrix and let $\boldsymbol{x} \in \mathbb{R}^n$. Then the rule $\boldsymbol{x} \mapsto A\boldsymbol{x}$ is a transformation called a *matrix transformation* or a *linear transformation*.

Example 2.6.9. Let $A = \begin{bmatrix} 0 & -2 \\ 1 & -3 \\ 2 & -3 \end{bmatrix}$ and define a transformation $T : \mathbb{R}^2 \to \mathbb{R}^3$ by $T(\boldsymbol{x}) = A\boldsymbol{x}$. Then

$$T(\boldsymbol{x}) = A\boldsymbol{x} = \begin{bmatrix} 0 & -2 \\ 1 & -3 \\ 2 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

(a) Let $\boldsymbol{u} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$. Compute $T(\boldsymbol{u})$.

$$T(\boldsymbol{u}) = A\boldsymbol{u} = \begin{bmatrix} 0(3) - 2(-1) \\ 1(3) - 3(-1) \\ 2(3) - 3(-1) \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 9 \end{bmatrix}.$$

(b) Find an $\boldsymbol{x} \in \mathbb{R}^2$ whose image under T is $\boldsymbol{b} = \begin{bmatrix} -8 \\ -7 \\ -2 \end{bmatrix}$.

We need an x_1 and x_2 such that

$$\begin{bmatrix} 0 & -2 \\ 1 & -3 \\ 2 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -8 \\ -7 \\ -2 \end{bmatrix}.$$

By row reduction, we solve the corresponding linear system:

$$\left[\begin{array}{cc|c} 0 & -2 & -8 \\ 1 & -3 & -7 \\ 2 & -3 & -2 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & -3 & -7 \\ 0 & -2 & -8 \\ 2 & -3 & -2 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & -3 & -7 \\ 0 & -2 & -8 \\ 0 & 3 & 12 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & -3 & -7 \\ 0 & 1 & 4 \\ 0 & 1 & 4 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & -3 & -7 \\ 0 & 1 & 4 \\ 0 & 0 & 0 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & 0 & 5 \\ 0 & 1 & 4 \\ 0 & 0 & 0 \end{array}\right].$$

Let $\boldsymbol{x} = \begin{bmatrix} 5 \\ 4 \end{bmatrix}$. Then, $T(\boldsymbol{x}) = \boldsymbol{b}$. In fact, $\boldsymbol{x}$ is the only vector whose image under T is $\boldsymbol{b}$. ∎

Python Programming

In Python, the product of a $m \times n$ matrix A and an n-vector $\boldsymbol{x}$ is computed using the command `np.matmul(A,x)` or more simply `A @ x`.

Example 2.6.10. The code:

```
import numpy as np

A = np.array([[ 0.7302560710008486, 0.11188723348377394, -6.592021716657104,
    8.042781326400146, 2.788326020264842, 14.242980690632749, 22.1772711500155,
    -21.764066710542092, 8.89271692192214, -22.23818944800199, 4.744977052078607,
    0.12367035830926021, -41.099408442522396, 27.149102210200162, -4.142488864751487,
    -4.848594837393037, 3.6155970577418657, 10.35888549721486, 13.861778110045924,
    -23.249678068081998, 27.658991254296332, 29.51010287533942, 8.486761307345963,
    -20.11094800337007, 14.227938547756157],
[ 34.02030644269619, -5.392793705179254, -1.9815912123337327, 13.398201496987028,
    20.651443364423393, -8.369561212118745, -6.177970127220448, -19.71448076386723,
    -24.51018166297458, -37.45287263414555, 4.777043856150392, 15.141179174454168,
    21.974871194430847, 24.878803819339602, -20.65107364599926, 39.995110717296996,
    35.50465679232937, -8.292448747588992, -8.609864246275453, -15.998142551641175,
    -25.6633365968774, -4.295382679542431, -18.662223974293333, 18.505229362558573,
    -31.44081956986173],
[ 5.37830956936058, -14.186721045286415, -12.502904020704921, -14.297018587946713,
    20.68564836204305, 3.0098462915425745, 15.975980180515492, 22.200401052635513,
    21.44364934018008, -26.363580411581886, -16.11091921761978, 13.137893639998936,
    -23.51301802216577, -4.226881358561691, -6.006547833355228, -7.446509205447858,
    21.31109433063274, 15.012102057159003, -16.633593431762463, 47.54692416180408,
    13.235822935575536, -9.749801412513047, 16.697188522428977, 35.109373447019834,
    -26.893396926881852],
[ -26.475277563815958, 4.998470795329371, -7.320206891019875, -24.668036342158167,
    16.69297382208554, 13.468460274972887, -17.271822515919578, 15.056196711153953,
    11.674294660719564, -14.440788443235917, -7.219273483678648, 1.7451442246726208,
    12.05790870774154, -19.901842965059302, 10.226736831469394, -64.53322586751794,
    1.2181045962120134, -29.468643523363028, -16.23959944452152, -14.999847374763629,
    10.883982868164841, -17.300724696487904, -1.5827094327770244, 22.287208255682323,
    -18.485622451339452],
[ -9.04930900161286, -15.934273592289763, 24.07800673039985, 7.00775935551542,
    0.24836341061532785, 16.555467328209797, 16.304317853652183, -50.17932097991834,
    -3.8855066979399187, 39.58131137216553, -6.509197910053838, 7.439311330481447,
    -23.95171496828803, 22.876945149482616, -2.8028182441321423, -19.46892662779619,
    5.644260187527502, -35.97534273172091, 8.317567299592325, 28.353509612530278,
    -31.822241466616184, -47.64739713473346, 32.05668239546048, 7.1372683990442285,
    -31.813691938414983],
[ 42.302238367070274, 26.939905524354504, 21.401767774860343, 23.919234277259704,
    18.4367498756433, -7.466697733864239, 17.639492629959612, 15.478619695829508,
    27.16088924608266, 31.657716903495455, 19.804028815576544, 12.383477355410353,
    -2.633794276013031, -26.022377210437377, 3.8591900518128543, 20.944266237782603,
    9.508899179806004, -24.105390768971326, 19.940745082196152, 2.2463935444230607,
    -25.612268866467964, -11.247341762547254, 22.725901891061195, 10.096663750069972,
    -26.32987414916184]])

x = np.array([ 4.735074512289696, 1.8856756758220632, 20.295582330466903, 18.06084313184242,
     -3.540684622226605, 15.51170765196704, 0.7365946209910419, 15.254408145859289,
    39.95568459070577, 14.540362132875753, 13.503399138285358, 4.098480643771802,
    -16.968425362418007, -26.147262136771204, 6.93014147962351, 7.878154967153436,
    -14.07961165024581, -4.117724304748075, 1.7631786510366272, 0.21139609795934255,
    36.44690635882133, -0.17642115791687996, -6.411089543000424, -6.099380836888234,
    12.78991511764993])

print(A @ x)
```

records and prints the product of the 6×25 matrix A and the 25-vector $\boldsymbol{x}$. and produces the 6-vector:
`[ 1106.76453642 -4218.55147486 -32.60068391 -188.96421924 -1674.887551 2491.52132973]` ■

Exercises

(Go to Solutions)

For Exercises 1-3, compute the matrix-vector product.

1. $\begin{bmatrix} 4 & 5 & 2 \\ 0 & -1 & 3 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 3 & -1 & 2 \\ 4 & 3 & 7 \\ -2 & 1 & 5 \end{bmatrix} \begin{bmatrix} 2 \\ -3 \\ 1 \end{bmatrix}$

3. $\begin{bmatrix} 2 & 8 & 10 \\ 6 & 3 & 9 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \\ 7 \end{bmatrix}$

For Exercises 4-7, express the given matrix equation as a system of equations or vice versa.

♠ 4. $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ -1 & -2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 2 \\ -9 \end{bmatrix}$

5. $\begin{bmatrix} 21 & 27 & 17 & -32 \\ 4 & 42 & 6 & 19 \\ 103 & -72 & 17 & -8 \\ 2 & 11 & -10 & 13 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 12 \\ -6 \\ 15 \\ 1 \end{bmatrix}$

6. $\begin{cases} x - 2y - 3z = -4 \\ y + 2z = 1 \\ -2x - 2z = 8 \end{cases}$

7. $\begin{cases} 3x_1 + x_3 + 2x_4 = 1 \\ -x_1 + 4x_2 + 5x_3 - 2x_4 = 17 \\ 6x_1 + x_3 = 15 \end{cases}$

For Exercises 8-11, solve the matrix equation.

8. $\begin{bmatrix} 1 & 2 & 3 \\ -1 & -1 & -3 \\ -1 & -2 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ -8 \\ -9 \end{bmatrix}$

♠ 9. $\begin{bmatrix} 4 & 4 & 2 \\ -4 & -3 & -2 \\ -4 & -3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ -3 \\ 3 \end{bmatrix}$

♠ 10. $\begin{bmatrix} 1 & 0 \\ 2 & 5 \\ 3 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 0 \end{bmatrix}$

11. $\begin{bmatrix} 3 & 0 & 1 & 2 \\ -1 & 4 & 5 & -2 \\ 6 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 17 \\ 15 \end{bmatrix}$

For Exercises 12-13, use the transformation $T : \mathbb{R}^4 \to \mathbb{R}^2$ given by the rule:

$$T(x_1, x_2, x_3, x_4) = \begin{bmatrix} 2 & -1 & 0 & 3 \\ 0 & 6 & -2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}.$$

♠ 12. Compute $T(1, 0, 0, 1)$ and $T(1, 2, 3, 4)$.

♠ 13. Find $\boldsymbol{x}$ such that $T(\boldsymbol{x}) = (1, 2)$.

For Exercises 14-15, use the transformation $T : \mathbb{R}^5 \to \mathbb{R}^2$ given by the rule:

$$T(x_1, x_2, x_3, x_4, x_5) = \begin{bmatrix} -1 & 2 & 1 & 3 & 4 \\ 0 & 0 & 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}.$$

14. Compute $T(1, 0, -1, 3, 0)$ and $T(1, 2, 3, 4, 5)$.

15. Find $\boldsymbol{x}$ such that $T(\boldsymbol{x}) = (15, -6)$.

♠ 16. Suppose we are a trader of currencies, e.g. US dollar, Euro, Korean won, Japanese yen, and we work with n different currencies. At a given time, let the vector $\boldsymbol{x}$ be our portfolio of amount holdings in each of the n currencies, that is, x_i is the current value we hold in currency i. Let A be an $n \times n$ matrix where a_{ij} is the exchange rate for exchanging currency j into currency i, that is, a_{ij} is the amount of currency i we would have if we exchange one note of currency j into currency i. This includes any fees associated with the exchange, that is, $a_{ji}a_{ij} < 1$. Note that $a_{ii} = 1$, that is, there is no change when a currency is no exchanged. Interpret the meaning for $A\boldsymbol{x}$.

17. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/column.ipynb

"All mankind... being all equal and independent, no one ought to harm another in his life, health, liberty or possessions." – John Locke

2.7 Linear Independence

The difference between a linear system having multiple solutions or a unique solution has to due with the presense of a free variable. Free variables emerge when certain vectors are optional, meaning they are not needed at all (with respect to a linear span). The notion of linear independence is describing a set of vectors with no such extra vectors, that is, all the vectors are necessary to produce the same span. Linear independence produces an algebraic metric of the shape of an affine set.

2.7.1 Linear Independence of Vectors

Definition 2.7.1. A set of vectors $\{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n\} \subseteq \mathbb{R}^m$ is said to be **linearly independent** if the vector equation

$$x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \ldots + x_n\boldsymbol{a}_n = \boldsymbol{0}$$

has only the trivial solution. Otherwise, the set is said to be **linearly dependent**.

In other words, a set of vectors is linearly dependent if there exists some weights $c_1, c_2, \ldots, c_n$, with at least one not zero, such that

$$c_1\boldsymbol{a}_1 + c_2\boldsymbol{a}_2 + \ldots + c_n\boldsymbol{a}_n = \boldsymbol{0}.$$

The previous equation is then called a **linear dependence relation**.

Expressed another way, suppose $A = \begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n \end{bmatrix}$, then $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$ is linearly independent if and only if $A\boldsymbol{x} = \boldsymbol{0}$ has no nontrivial solution, since

$$A\boldsymbol{x} = \begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \ldots & \boldsymbol{a}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \ldots + x_n\boldsymbol{a}_n = \boldsymbol{0}.$$

The set $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$ is linearly dependent if and only if $A\boldsymbol{x} = \boldsymbol{0}$ has a nontrivial solution.

Example 2.7.2. Let $\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, $\boldsymbol{v}_2 = \begin{bmatrix} 3 \\ 1 \\ -2 \end{bmatrix}$, and $\boldsymbol{v}_3 = \begin{bmatrix} -3 \\ 4 \\ 13 \end{bmatrix}$.

(a) Determine if the set $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is linearly independent.

To determine if the set is linearly independent, it suffices to compute the echelon form of the corresponding homogeneous system. In fact, the final row of zeros is irrelevant.

$$\begin{bmatrix} 1 & 3 & -3 \\ 2 & 1 & 4 \\ 3 & -2 & 13 \end{bmatrix} \sim \begin{bmatrix} 1 & 3 & -3 \\ 0 & -5 & 10 \\ 0 & -11 & 22 \end{bmatrix} \sim \begin{bmatrix} 1 & 3 & -3 \\ 0 & 1 & -2 \\ 0 & 1 & -2 \end{bmatrix} \sim \begin{bmatrix} 1 & 3 & -3 \\ 0 & 1 & -2 \\ 0 & 0 & 0 \end{bmatrix}.$$

Thus, x_3 is a free variable of the system, which implies that the homogeneous equation $A\boldsymbol{x} = \boldsymbol{0}$ does have a nontrivial solution. Therefore, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is linearly dependent.

(b) Find a linear dependence relation among $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, and $\boldsymbol{v}_3$.

To determine a dependence relation, we finish the row reduction from above.

$$\begin{bmatrix} 1 & 3 & -3 \\ 0 & 1 & -2 \\ 0 & 0 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 0 \end{bmatrix}.$$

Therefore,

$$\begin{cases} x_1 + 3x_3 = 0 \\ x_2 - 2x_3 = 0 \\ 0 = 0 \end{cases}$$

Thus, $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -3x_3 \\ 2x_3 \\ x_3 \end{bmatrix} = x_3 \begin{bmatrix} -3 \\ 2 \\ 1 \end{bmatrix}$. Thus, $-3\boldsymbol{v}_1 + 2\boldsymbol{v}_2 + \boldsymbol{v}_3 = \boldsymbol{0}$ (corresponding to $x_3 = 1$). Of course, there are infinitely many dependence relations. For example, $-15\boldsymbol{v}_1 + 10\boldsymbol{v}_2 + 5\boldsymbol{v}_3 = \boldsymbol{0}$ (corresponding to $x_3 = 5$).

We also note that we have shown that $\boldsymbol{v}_3 \in \text{Span}\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ since $-3\boldsymbol{v}_1 + 2\boldsymbol{v}_2 + \boldsymbol{v}_3 = \boldsymbol{0}$ implies $\boldsymbol{v}_3 = 3\boldsymbol{v}_1 - 2\boldsymbol{v}_2$. This observation is always the case for linear dependence. ∎

Example 2.7.3. Determine if the columns of the matrix $A = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 2 & 4 \\ 2 & -4 & -7 \end{bmatrix}$ are linearly independent.

It suffices to show that $A\boldsymbol{x} = \boldsymbol{0}$ has no nontrivial solutions, that is, that an echelon form of A has no zero rows.

$$\begin{bmatrix} \boxed{1} & -1 & -1 \\ -1 & 2 & 4 \\ 2 & -4 & -7 \end{bmatrix} \sim \begin{bmatrix} \boxed{1} & -1 & -1 \\ 0 & \boxed{1} & 3 \\ 0 & -2 & -5 \end{bmatrix} \sim \begin{bmatrix} \boxed{1} & -1 & -1 \\ 0 & \boxed{1} & 3 \\ 0 & 0 & \boxed{1} \end{bmatrix}.$$

Therefore, the columns of A are linearly independent. ∎

These examples illustrate an important point: a set of vector is linearly dependent if and only if the corresponding homogeneous linear system has a free variable. In other words, a set of vectors is linearly independent if and only if the associated coefficient matrix has a pivot in each column. We summarize this observation in the following theorem.

Theorem 2.7.4. *Let A be an $m \times n$ matrix. Then the following are equivalent:*

(i) The columns of A are linearly independent.

(ii) The homogeneous linear system $A\boldsymbol{x} = \boldsymbol{0}$ has a unique solution, namely the trivial solution $\boldsymbol{x} = \boldsymbol{0}$.

(iii) The homogeneous linear system $A\boldsymbol{x} = \boldsymbol{0}$ has no free variables.

(iv) The homogeneous linear system $A\boldsymbol{x} = \boldsymbol{0}$ has pivots in all columns of A.

2.7.2 Dimension

Definition 2.7.5. A **basis** $\mathcal{B}$ of a vector space V is a linearly independent, spanning set of V. The size of the basis, $|\mathcal{B}|$, is called the **dimension** of V, denoted $\dim V$.

Every line through the origin is a one-dimensional subspace and every plane through the origin is two-dimensional. In general, an affine set passing through the origin spanned by p independent vectors (aka, a subspace) has dimension p.

Example 2.7.6. For $\mathbb{R}^n$, the set $\mathcal{E} = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n\}$, where $\boldsymbol{e}_i$ is the vector with a 1 in the ith component and 0's everywhere else, is always a basis. This is known as the **standard basis** of $\mathbb{R}^n$.

On the other hand, the set $\mathcal{B} = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}$ is a non-standard basis for $\mathbb{R}^3$. ∎

We mention some important properties of bases and dimension.

Theorem 2.7.7. *Let $\mathcal{B}$ be a subset of the vector space V. Then the following are equivalent:*

(i) $\mathcal{B}$ is a basis;

(ii) $\mathcal{B}$ is a maximal linearly independent set, that is, $\mathcal{B}$ is linearly independent but $\mathcal{B} \cup \{\boldsymbol{u}\}$ is linearly dependent for any $\boldsymbol{u} \in V$;

(iii) $\mathcal{B}$ is a minimal spanning set of V, that is, $V = \operatorname{Span}(\mathcal{B})$ but $V \neq \operatorname{Span}(\mathcal{B} \setminus \{\boldsymbol{v}\})$ for any $\boldsymbol{v} \in \mathcal{B}$.

(iv) All vectors in V can be uniquely expressed as a linear combination of vectors from $\mathcal{B}$.

2.7.3 Basis for Column Space

Given any $m \times n$ matrix A, there are two fundamental subspaces associated to A: the *column space* $\operatorname{Col}(A)$ which is the span of the column vectors of A (hence, a subspace of $\mathbb{R}^m$) and the *null space* $\operatorname{Nul}(A)$ which is the solution set to the homogeneous system $A\boldsymbol{x} = \boldsymbol{0}$ (hence, a subspace of $\mathbb{R}^n$). The dimension of $\operatorname{Col}(A)$ is called the **rank** of A, and the dimension of $\operatorname{Nul}(A)$ is called the **nullity** of A.

Theorem 2.7.8. *The pivot columns of A form a basis for the column space of A.*

The location of the pivots (or to say, the absence of pivots) in the echelon form of A tells us which column vectors can be expressed as a linear combination of the previous column vectors

Example 2.7.9. Let $A = \begin{bmatrix} 1 & 3 & 3 & 2 & 4 \\ 2 & 7 & 6 & 3 & 9 \\ 1 & 2 & 3 & 3 & 3 \end{bmatrix}$. Construct a basis for $\operatorname{Col}(A)$.

We begin by computing an echelon form of A:

$$\begin{bmatrix} \boxed{1} & 3 & 3 & 2 & 4 \\ 2 & 7 & 6 & 3 & 9 \\ 1 & 2 & 3 & 3 & 3 \end{bmatrix} \sim \begin{bmatrix} \boxed{1} & 3 & 3 & 2 & 4 \\ 0 & \boxed{1} & 0 & -1 & 1 \\ 0 & -1 & 0 & 1 & -1 \end{bmatrix} \sim \begin{bmatrix} \boxed{1} & 3 & 3 & 2 & 4 \\ 0 & \boxed{1} & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

We see now that this matrix is in echelon form, for which we can now easily identify the pivot columns, namely the first and second. This indicates that the third and fourth column vectors are both linear combinations of the first two column vectors. Therefore, a basis of $\operatorname{Col} A$ is $\left\{ \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 3 \\ 7 \\ 2 \end{bmatrix} \right\}$ and $\operatorname{rank}(A) = 2$. ■

Example 2.7.10. Let $A = \begin{bmatrix} 1 & 3 & 3 & 2 & -9 \\ -2 & -2 & 2 & -8 & 2 \\ 2 & 3 & 0 & 7 & 1 \\ 3 & 4 & -1 & 11 & -8 \end{bmatrix}$. Find a basis for the column space of A.

Since $A \sim \begin{bmatrix} \boxed{1} & 0 & -3 & 5 & 0 \\ 0 & \boxed{1} & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & \boxed{1} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$, we see that the third and fourth column vectors are linear combinations of the first and second column vectors. Thus, the set $\left\{ \begin{bmatrix} 1 \\ -2 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 3 \\ -2 \\ 3 \\ 4 \end{bmatrix}, \begin{bmatrix} -9 \\ 2 \\ 1 \\ -8 \end{bmatrix} \right\}$ is a basis for $\operatorname{Col} A$, and the $\operatorname{rank}(A) = \dim(\operatorname{Col}(A)) = 3$. ■

Python Programming

We have seen that a set of vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\} \subseteq \mathbb{R}^m$ is linearly independent if and only if the $m \times n$ matrix $A = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_n \end{bmatrix}$ has **full rank**, that is, $\operatorname{rank} A = n$ (see Theorem 2.7.4). The `NumPy` function `np.linalg.matrix_rank(A)` computes $\operatorname{rank} A$. Note that n is the number of columns of A, which is equal to `len(A[0])`, the length of the first row of A.[i] Hence, the set of vectors is linearly independent if and only if `len(A[0])==np.linalg.matrix_rank(A)` is `True`. It is linearly dependent otherwise.

Determining a dependence relation is a bit more challenging. A set of vector is linear dependent if and only if one vector is a linear combination of the others. We can find one such linear combination by examining the pivots in the RREF. From this, we can construct a dependency relation. The following example shows how one could do this.

Example 2.7.11. Recall in Example 2.7.2 that the set of vectors $\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, $\boldsymbol{v}_2 = \begin{bmatrix} 3 \\ 1 \\ -2 \end{bmatrix}$, and $\boldsymbol{v}_3 = \begin{bmatrix} -3 \\ 4 \\ 13 \end{bmatrix}$ is linearly dependent. Suppose we do not know the RREF of the matrix

$$A = \begin{bmatrix} 1 & 3 & -3 \\ 2 & 1 & 4 \\ 3 & -2 & 13 \end{bmatrix}$$

because we did not compute it, but instead determined that the columns of A are linearly dependent because we observed that $\operatorname{rank} A = 2 \neq 3$. How could we find a dependence relation? It must be true that either $\boldsymbol{v}_3 \in \operatorname{Span}\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$, $\boldsymbol{v}_2 \in \operatorname{Span}\{\boldsymbol{v}_1, \boldsymbol{v}_3\}$, or $\boldsymbol{v}_1 \in \operatorname{Span}\{\boldsymbol{v}_2, \boldsymbol{v}_3\}$. If we investigate $\boldsymbol{v}_3 \in \operatorname{Span}\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$, consider the following Python code:[ii]

```
import numpy as np

A = np.column_stack([[1,2,3],
                     [3,1,-2],
                     [-3,4,13]])
if len(A[0])==np.linalg.matrix_rank(A): #does number of pivots equal columns?
  print('linearly independent')
else:
  print('linearly dependent')

Asub = A[:,:-1] # Is v[2] a linear combination of v[0] and v[1]?
x = np.linalg.lstsq(Asub,A[:,2],rcond=None)[0] #solve the linear system
print(Asub@x)                          #notice this product is 1*v[2]
print(np.round(x,0),'\n')              #3*v[0] - 2*v[1] = 1*v[0]
```

which outputs:

```
linearly dependent
[-3.  4.  13.]
[ 3.  -2.]
```

This tells us that $3\boldsymbol{v}_1 - 2\boldsymbol{v}_2 = 1\boldsymbol{v}_3$. Therefore, $-3\boldsymbol{v}_1 + 2\boldsymbol{v}_2 + \boldsymbol{v}_3 = \boldsymbol{0}$ is a dependency relation. ■

[i] Conversely, `len(A)` is the number of rows in A, aka m.

[ii] Note that this code is based upon solving overdetermined linear systems using `np.linalg.lstsq`, as seen in Section 2.5).

Exercises

(Go to Solutions)

For Exercises 1-5, determine if the set of vectors $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p\}$ is linearly independent or dependent. If linearly dependent, provide a nontrivial dependency relation.

1. $\left\{ \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \\ 4 \end{bmatrix}, \begin{bmatrix} 3 \\ 6 \\ 5 \end{bmatrix} \right\}$

♠ 2. $\left\{ \begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ -2 \\ 6 \end{bmatrix}, \begin{bmatrix} -2 \\ 3 \\ -2 \end{bmatrix} \right\}$

3. $\left\{ \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}, \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 \\ 1 \\ 3 \end{bmatrix} \right\}$

♠ 4. $\left\{ \begin{bmatrix} 1 \\ -2 \\ -1 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 2 \\ 2 \\ 4 \end{bmatrix}, \begin{bmatrix} 2 \\ -3 \\ -3 \\ 2 \end{bmatrix} \right\}$

5. $\left\{ \begin{bmatrix} 1 \\ 2 \\ -1 \\ 3 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ -1 \\ 1 \\ 0 \end{bmatrix} \right\}$

6. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/independence.ipynb

"After every storm the sun will smile; for every problem there is a solution, and the soul's indefeasible duty is to be of good cheer." – William R. Alger

2.8 Solution Sets of Linear Systems

In this section, we will see that the general solution of a homogeneous linear system is just the null space of the associated coefficient matrix. Generalizing this, we will see that the general solution to a nonhomogeneous linear system will be the general solution of the associated homogeneous solution and some particular solution. This particular solution is generally found by setting all the free variables, if any, equal to zero. In particular, the solution set of a linear system is an affine set, and any affine set can be the solution of a linear system.

2.8.1 The Null Space

Definition 2.8.1. Let A be an $m \times n$ matrix. Then the **null space** (or **kernel**) of A, denoted $\text{Nul}\, A$, is the set of all solutions of the homogeneous system $A\boldsymbol{x} = \boldsymbol{0}$. The **nullity** of A, denoted $\text{nullity}(A)$, is the number of non-pivot columns in A. This value counts the number of free variables in the homogeneous system and essentially measures the "size" of null space.

Example 2.8.2. For the matrix $A = \begin{bmatrix} 1 & 3 & -3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & -2 & 13 & 1 \end{bmatrix}$, find a spanning set for its null space.

Note that when solving a homogeneous linear system, the augmented column is just the zero vector and no row operation will ever transform a column of zeros. As such, we often omit the zero column when solving homogeneous systems. In fact, solving homogeneous systems is essentially the same techniques as determining whether a set of vectors is linearly independent (compare Example 2.7.2).

$$\begin{bmatrix} 1 & 3 & -3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & -2 & 13 & 1 \end{bmatrix} \sim \begin{bmatrix} 1 & 3 & -3 & 4 \\ 0 & -5 & 10 & -5 \\ 0 & -11 & 22 & -11 \end{bmatrix} \sim \begin{bmatrix} 1 & 3 & -3 & 4 \\ 0 & 1 & -2 & 1 \\ 0 & 1 & -2 & 1 \end{bmatrix} \sim \begin{bmatrix} 1 & 3 & -3 & 4 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & 3 & 1 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

The associated reduced linear system would then be

$$\begin{cases} x_1 + 3x_3 + x_4 = 0 \\ x_2 - 2x_3 + x_4 = 0 \\ 0 = 0 \end{cases} \sim \begin{cases} x_1 = -3x_3 - x_4 \\ x_2 = 2x_3 - x_4 \end{cases}$$

Thus, the general solution to the homogeneous system $A\boldsymbol{x} = \boldsymbol{0}$ is

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -3x_3 - x_4 \\ 2x_3 - x_4 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -3x_3 \\ 2x_3 \\ x_3 \\ 0 \end{bmatrix} + \begin{bmatrix} -x_4 \\ -x_4 \\ 0 \\ x_4 \end{bmatrix} = x_3 \begin{bmatrix} -3 \\ 2 \\ 1 \\ 0 \end{bmatrix} + x_4 \begin{bmatrix} -1 \\ -1 \\ 0 \\ 1 \end{bmatrix}.$$

Let $\boldsymbol{u} = (-3, 2, 1, 0)$ and $\boldsymbol{v} = (-1, -1, 0, 1)$. Hence, $\text{Nul}(A) = \text{Span}\{\boldsymbol{u}, \boldsymbol{v}\}$. ∎

A homogeneous system $A\boldsymbol{x} = \boldsymbol{0}$ always has a solution, namely $\boldsymbol{x} = \boldsymbol{0}$. This can be thought of as the **trivial solution**. Any other solution to a homogeneous system is a **nontrivial solution**. Thus, the general solution to the homogeneous system can be expressed as $\boldsymbol{x} = x_1\boldsymbol{v}_1 + x_2\boldsymbol{v}_2 + \ldots + x_n\boldsymbol{v}_n$, where $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ is a basis for $\text{Nul}(A)$. Additionally, $\boldsymbol{x}$ is a nontrivial solution if and only if at least one $x_i \neq 0$.

2.8.2 Basis for Null Space

Theorem 2.8.3. *If two matrices A and B are row equivalent, then* $\operatorname{Nul} A = \operatorname{Nul} B$.

Because of the previous theorem, one can use the row-reduced echelon form of A to find a basis for $\mathrm{Nul}(A)$. This will be the the same technique we applied in Example 2.8.2 to find a spanning set for $\mathrm{Nul}(A)$, which was already a basis. The basis vectors will be derived from the non-pivot columns of A's row-reduced echelon form, that is, those columns which correspond to free variables of the linear system.

Example 2.8.4. Let $A = \begin{bmatrix} 1 & 3 & 3 & 2 & 4 \\ 2 & 7 & 6 & 3 & 9 \\ 1 & 2 & 3 & 3 & 3 \end{bmatrix}$. Construct a basis for $\mathrm{Nul}(A)$.

Remember that $\operatorname{Nul} A$ is the solution set to $A\boldsymbol{x} = \mathbf{0}$, which we now solve by row reducing the matrix A. We might recognize that this is the same matrix from Example 2.8.4. As such, we have already seen that

$$[\ A \mid \mathbf{0}\ ] \sim \left[\begin{array}{ccccc|c} \boxed{1} & 3 & 3 & 2 & 4 & 0 \\ 0 & \boxed{1} & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right].$$

From echelon form, we can identify the pivot and non-pivot columns. Thus, we see that $\mathrm{nullity}(A) = 3$. To find a specific basis, we are advantaged to continue to reduce the matrix to row-reduced echelon form, as shown below:

$$[\ A \mid \mathbf{0}\ ] \sim \left[\begin{array}{ccccc|c} \boxed{1} & 0 & 3 & 5 & 1 & 0 \\ 0 & \boxed{1} & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right].$$

As we have seen many times, this augmented matrix corresponds to a linear system, namely

$$\begin{cases} x_1 + 3x_3 + 5x_4 + x_5 = 0 \\ x_2 - x_4 + x_5 = 0. \end{cases}$$

In usual tradition, we solve for the two dependent variables, x_1, x_2, in terms of the three free variables, x_3, x_4, x_5. From this we see

$$\begin{cases} x_1 = -3x_3 - 5x_4 - x_5 \\ x_2 = x_4 - x_5, \end{cases}$$

which provides the parametric equations to the flat which is the solution set to $A\boldsymbol{x} = \mathbf{0}$. Converting these parametric equations into a single vector equation, we see

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} -3x_3 - 5x_4 - x_5 \\ x_4 - x_5 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = x_3 \begin{bmatrix} -3 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + x_4 \begin{bmatrix} -5 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} + x_5 \begin{bmatrix} -1 \\ -1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = x_3\boldsymbol{u} + x_4\boldsymbol{v} + x_5\boldsymbol{w}. \quad (2.8.1)$$

Then $\mathrm{Nul}(A) = \mathrm{Span}\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}\}$. Thus, $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}\}$ is a spanning set for $\mathrm{Nul}(A)$. On the other hand, the construction of $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}\}$ guarantees that they are linearly independent since

$$x_3\boldsymbol{u} + x_4\boldsymbol{v} + x_5\boldsymbol{w} = \mathbf{0}$$

implies that x_3, x_4, and x_5 are all zero. To see this, consider the 3rd, 4th, and 5th components of $\boldsymbol{x}$ in (2.8.1). As $\boldsymbol{u}$ contains a 1 in the 3rd component and $\boldsymbol{v}$ and $\boldsymbol{w}$ both contain 0, any combination of $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ to form $\boldsymbol{x}$ must have the coefficient of $\boldsymbol{u}$ be x_3. The same is also true that the coefficients of $\boldsymbol{v}$ and $\boldsymbol{w}$ must be x_4 and x_5, respectively. Hence, if $\boldsymbol{x}$ were $\mathbf{0}$, then $x_3 = x_4 = x_5 = 0$, proving the claim. Therefore, $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}\}$ is a basis for $\text{Nul}(A)$. ∎

2.8.3 Particular Solutions to Linear Systems

Like a homogeneous system, a non-homogeneous system of linear equations with multiple solutions can have their solutions expressed in parametric form.

Example 2.8.5. Describe all solutions of $A\boldsymbol{x} = \boldsymbol{b}$, where

$$A = \begin{bmatrix} 4 & 3 & -11 \\ 1 & -8 & 6 \\ -1 & -10 & 12 \end{bmatrix} \text{ and } \boldsymbol{b} = \begin{bmatrix} 17 \\ -22 \\ -32 \end{bmatrix}.$$

Row reducing the augmented matrix, we can see that

$$\left[\begin{array}{ccc|c} 4 & 3 & -11 & 17 \\ 1 & -8 & 6 & -22 \\ -1 & -10 & 12 & -32 \end{array}\right] \sim \left[\begin{array}{ccc|c} 4 & 3 & -11 & 17 \\ 0 & -18 & 18 & -54 \\ 3 & -7 & 1 & -15 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 10 & -12 & 32 \\ 0 & 1 & -1 & 3 \\ 3 & -7 & 1 & -15 \end{array}\right]$$

$$\sim \left[\begin{array}{ccc|c} 1 & 0 & -2 & 2 \\ 0 & 1 & -1 & 3 \\ 0 & 37 & -37 & -111 \end{array}\right] \sim \left[\begin{array}{ccc|c} \boxed{1} & 0 & -2 & 2 \\ 0 & \boxed{1} & -1 & 3 \\ 0 & 0 & 0 & 0 \end{array}\right].$$

Thus, the related system is

$$\begin{cases} x_1 - 2x_3 = 2 \\ x_2 - x_3 = 3 \\ 0 = 0. \end{cases}$$

and the solutions are all of the form:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 + 2x_3 \\ 3 + x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix},$$

where $(2, 3, 0)$ is a particular solution to the system and $\text{Nul}(A) = \text{Span}\{(2, 1, 1)\}$. ∎

Theorem 2.8.6. *Suppose the equation $A\boldsymbol{x} = \boldsymbol{b}$ is consistent for some given $\boldsymbol{b}$, and let $\boldsymbol{x}_0$ be a (particular) solution. Then the solution set of $A\boldsymbol{x} = \boldsymbol{b}$ is the set of all vectors of the form $\boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{x}_n$, where $\boldsymbol{x}_n$ is any solution of the homogeneous equation $A\boldsymbol{x} = \mathbf{0}$ and can be expressed in parametric form.*

From a geometric perspective, a solution set to a homogeneous system is the null space of its coefficient matrix A. As such, it is a subspace spanned by n independent vectors, where $n = \text{nullity}(A)$. Theorem 2.8.6 tells us that a solution set to a non-homogeneous system generically is an affine span of n independent vectors, in other words, it is an n-flat.

Exercises

(Go to Solutions)

For Exercises 1-11, let A be the provided matrix. Find a basis for $\text{Col}(A)$ consisting of column vectors of A and a basis for $\text{Nul}(A)$. Find the rank and nullity of A. Answers may vary.

1. $\begin{bmatrix} 1 & 15 & 8 \\ 0 & 9 & 6 \\ 0 & 0 & 2 \end{bmatrix}$

2. $\begin{bmatrix} 1 & 1 & 2 \\ 2 & 3 & 5 \\ 4 & 2 & 4 \end{bmatrix}$

3. $\begin{bmatrix} 1 & 1 & 1 & 0 \\ 2 & 3 & 4 & 2 \\ 3 & 1 & 3 & 1 \end{bmatrix}$

4. $\begin{bmatrix} 5 & 4 & 2 & 1 \\ 4 & 4 & 2 & 8 \\ 2 & 5 & 6 & 3 \end{bmatrix}$

♠ 5. $\begin{bmatrix} 8 & -3 & -13 & 15 & 17 \\ -2 & 1 & 3 & -3 & -3 \end{bmatrix}$

♠ 6. $\begin{bmatrix} -1 & 2 & 1 & 0 \\ 7 & -14 & -7 & -8 \\ -3 & 6 & 3 & 2 \end{bmatrix}$

7. $\begin{bmatrix} 1 & 2 & 5 & 9 \\ 3 & 4 & 7 & 1 \end{bmatrix}$

8. $\begin{bmatrix} 1 & 4 & 8 & -1 \\ 2 & 4 & 8 & 0 \\ 3 & 2 & 4 & 1 \end{bmatrix}$

9. $\begin{bmatrix} 5 & 0 & 0 & 5 & 6 & 5 \\ 0 & 4 & 0 & 4 & 4 & 6 \\ 9 & 0 & 0 & 9 & 9 & 0 \end{bmatrix}$

10. $\begin{bmatrix} 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 4 & 3 & 2 & 1 & 6 \\ 2 & 4 & 6 & 8 & 2 & 4 \\ 0 & 1 & 0 & 1 & 2 & 6 \end{bmatrix}$

11. $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 2 & -2 & 4 & 4 & 8 \end{bmatrix}$

For Exercises 12-15, describe all solutions to the matrix equation $A\boldsymbol{x} = \boldsymbol{b}$. The matrix A will be listed first, followed by the vector $\boldsymbol{b}$. Answers may vary.

12. $\begin{bmatrix} -4 & 0 & -4 \\ 3 & -4 & 3 \\ 2 & 0 & 2 \end{bmatrix}, \begin{bmatrix} -4 \\ 6 \\ 2 \end{bmatrix}$

♠ 13. $\begin{bmatrix} 9 & -4 & -6 \\ -1 & 6 & -16 \\ -1 & -2 & 8 \end{bmatrix}, \begin{bmatrix} 1 \\ 11 \\ -5 \end{bmatrix}$

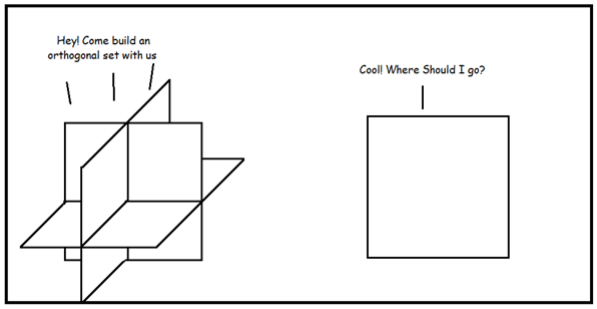
♠ 14. $\begin{bmatrix} 2 & -2 & 2 & 0 \\ 3 & 4 & -5 & 0 \\ 0 & 0 & 6 & 0 \\ -2 & -1 & 4 & 0 \end{bmatrix}, \begin{bmatrix} 4 \\ -4 \\ 18 \\ 8 \end{bmatrix}$

15. $\begin{bmatrix} 2 & -6 & 8 & 0 \\ 1 & -3 & 4 & 1 \\ 0 & 0 & 0 & 1 \\ -1 & 3 & -4 & 1 \end{bmatrix}, \begin{bmatrix} 10 \\ -3 \\ -8 \\ -13 \end{bmatrix}$

16. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/solutionset.ipynb

"To be trusted is a greater compliment than being loved." – George MacDonald

2.9 Orthogonality



2.9.1 Orthogonal Vectors

Definition 2.9.1. Two nonzero[i] vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ are **orthogonal** if $\boldsymbol{u} \cdot \boldsymbol{v} = 0$.

Example 2.9.2. Given the vectors $\boldsymbol{u} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, $\boldsymbol{v} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$, and $\boldsymbol{w} = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix}$, determine if $\boldsymbol{u}$ is orthogonal to $\boldsymbol{v}$ or $\boldsymbol{w}$.

$$\boldsymbol{u} \cdot \boldsymbol{v} = 1(1) + 2(1) + 3(-1) = 3 - 3 = 0.$$

Thus, $\boldsymbol{u}$ is orthogonal to $\boldsymbol{v}$.

$$\boldsymbol{u} \cdot \boldsymbol{w} = 1(2) + 2(3) + 3(5) = 23 \neq 0.$$

Thus, $\boldsymbol{u}$ is not orthogonal to $\boldsymbol{w}$.

$$\boldsymbol{v} \cdot \boldsymbol{w} = 1(2) + 1(3) - 1(5) = 5 - 5 = 0.$$

Thus, $\boldsymbol{v}$ is orthogonal to $\boldsymbol{w}$. ∎

Theorem 2.9.3 (Pythagorean Theorem)**.** *If two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ are orthogonal, then*

$$\|\boldsymbol{u} + \boldsymbol{v}\|^2 = \|\boldsymbol{u}\|^2 + \|\boldsymbol{v}\|^2.$$

Proof. By assumption, $\boldsymbol{u} \cdot \boldsymbol{v} = 0 = \boldsymbol{v} \cdot \boldsymbol{u}$. Note that $\|\boldsymbol{u}+\boldsymbol{v}\|^2 = (\boldsymbol{u}+\boldsymbol{v}) \cdot (\boldsymbol{u}+\boldsymbol{v}) = \boldsymbol{u} \cdot \boldsymbol{u} + \boldsymbol{u} \cdot \boldsymbol{v} + \boldsymbol{v} \cdot \boldsymbol{u} + \boldsymbol{v} \cdot \boldsymbol{v} = \boldsymbol{u} \cdot \boldsymbol{u} + 0 + 0 + \boldsymbol{v} \cdot \boldsymbol{v} = \|\boldsymbol{u}\|^2 + \|\boldsymbol{v}\|^2$. □

We call the above equation the Pythagorean Theorem because how it resembles the trigonometric equation, after all it states that a sum of squares is equal to a square under certain conditions, but there is a deeper geometry connection. First of all, if two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are linearly independent, then they span a plane in $\mathbb{R}^n$, but they also form a triangle with sides $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{u} + \boldsymbol{v}$, where the three points correspond to the tail of $\boldsymbol{u}$ (which is also the tail of $\boldsymbol{u} + \boldsymbol{v}$), the head of $\boldsymbol{u}$ (which is also the tail of $\boldsymbol{v}$), and the head of $\boldsymbol{v}$ (which is also the head of $\boldsymbol{u} + \boldsymbol{v}$). The norms of the three sides of this triangle is $\|\boldsymbol{u}\|$, $\|\boldsymbol{v}\|$, and $\|\boldsymbol{u} + \boldsymbol{v}\|$. Therefore, the above theorem is saying that the sums of squares of two sides of the a triangle equal the square of the other side. This is exactly the statement of the Pythagorean Theorem from Trigonometry, but furthermore the Trigonometric Pythagorean Theorem guarantees this can only happen if there is a right

[i] When it comes to defining orthogonality, we have to exclude the zero vector. Note that $\boldsymbol{0} \cdot \boldsymbol{v} = 0$ for every vector $\boldsymbol{v}$. As such, if we allowed the zero vector to be orthogonal then it would be orthogonal to every vector, including itself, which would provide needless counterexamples to theorems and geometric interpretations that will follow.

angle present, that is, the angle between the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ must be a right angle. This proves the notion that two vectors are orthogonal if and only if the angle between the two vectors is $90°$.

An alternative way of expressing a plane in $\mathbb{R}^3$ is by orthogonality. For example, we can construct a plane in $\mathbb{R}^3$ by taking all vectors in $\mathbb{R}^3$ that are orthogonal to some fixed vector $\boldsymbol{n} = (a,\ b,\ c)$, called the **normal vector** of the plane. If $\boldsymbol{x} = (x,\ y,\ z)$ is a vector on the plane, then

$$\boldsymbol{n} \cdot \boldsymbol{x} = ax + by + cz = 0.$$

This produces the equation of a plane which passes through the origin (notice that $\mathbf{0}$ is a solution here). In order to have the plane pass through the particular vector $\boldsymbol{x}_0$ we replace $\boldsymbol{x}$ with $\boldsymbol{x} - \boldsymbol{x_0}$, that is,

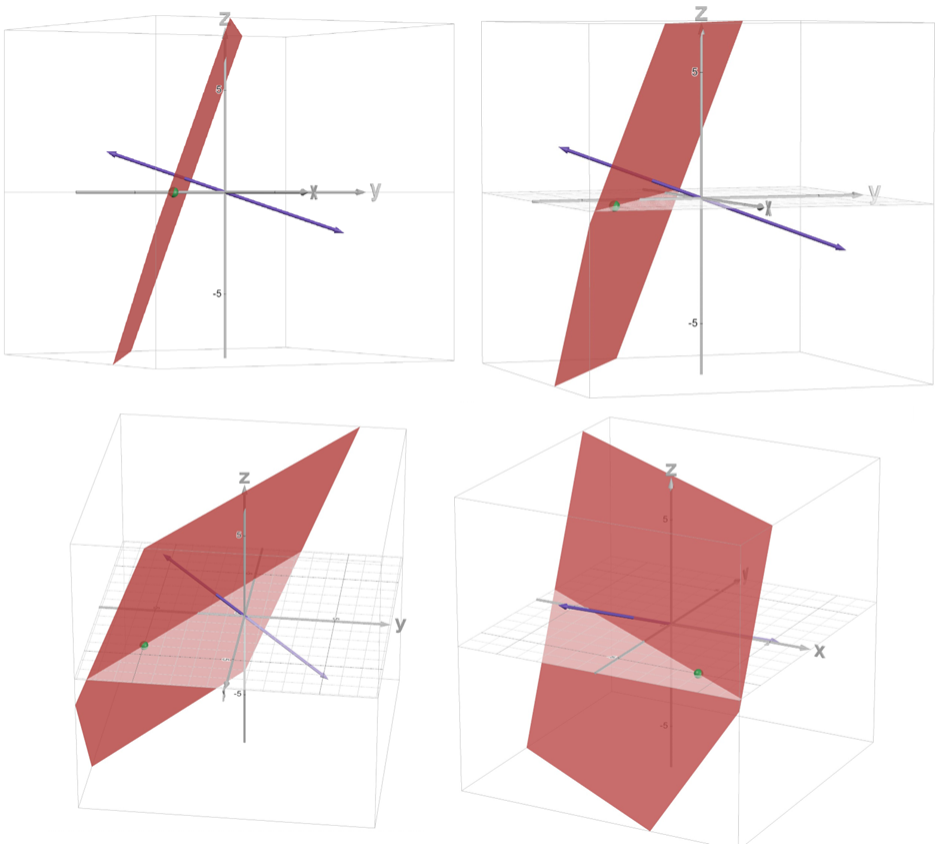
$$\begin{aligned} \boldsymbol{n} \cdot (\boldsymbol{x} - \boldsymbol{x_0}) &= 0 \\ a(x - x_0) + b(y - y_0) + c(z - z_0) &= 0 \\ ax + by + cz &= d \end{aligned}$$

gives an equation of the plane in $\mathbb{R}^3$ through the point (x_0, y_0, z_0) and orthogonal to $\boldsymbol{n}$.

Example 2.9.4. The solutions to the equation

$$3(x-4) + 5(y+5) - 2z = 0 \quad \Rightarrow \quad 3x + 5y - 2z = -13$$

is the plane containing the point $(4, -5, 0)$ and is perpendicular to $(3, 5, -2)$.



■

Similarly, this strategy will create a hyperplane in $\mathbb{R}^n$. In particular, the hyperplane containing $\boldsymbol{x}_0$ and is orthogonal to $\boldsymbol{n}$ will be the solutions $\boldsymbol{x}$ to the linear equation, given in **point-normal form**,

$$\boldsymbol{n} \cdot (\boldsymbol{x} - \boldsymbol{x}_0) = 0. \tag{2.9.1}$$

2.9.2 Orthogonal Sets

Definition 2.9.5. A set of nonzero vectors $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_r\} \subseteq \mathbb{R}^n$ is called an **orthogonal set** if each pair of distinct vectors is orthogonal, that is, $\boldsymbol{v}_i \cdot \boldsymbol{v}_j = 0$ whenever $i \neq j$. An orthogonal set of vectors is called **orthonormal** if each vector in the set is also a unit vector.

Of course, any orthogonal set can be transformed into an orthonormal set by normalizing each vector, that is, replacing each vector $\boldsymbol{v}$ with $\dfrac{1}{\|\boldsymbol{v}\|}\boldsymbol{v}$.

Example 2.9.6. Let $\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \boldsymbol{v}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, \boldsymbol{v}_3 = \begin{bmatrix} -5 \\ 4 \\ -1 \end{bmatrix}$. Then the set $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is orthogonal with respect to the dot product. To see this, we check the three dot products:

$$\begin{aligned} \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 &= 1 + 2 - 3 = 0 \\ \boldsymbol{v}_1 \cdot \boldsymbol{v}_3 &= -5 + 8 - 3 = 0 \\ \boldsymbol{v}_2 \cdot \boldsymbol{v}_3 &= -5 + 4 + 1 = 0. \end{aligned}$$

On the other hand, $\|\boldsymbol{v}_1\| = \sqrt{14}$, $\|\boldsymbol{v}_2\| = \sqrt{3}$, and $\|\boldsymbol{v}_3\| = \sqrt{42}$. Hence, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is not orthonormal, since the elements are not unit vectors. This can be easily corrected by normalizing each vector. Therefore, $\left\{ \dfrac{1}{\sqrt{14}}\boldsymbol{v}_1, \dfrac{1}{\sqrt{3}}\boldsymbol{v}_2, \dfrac{1}{\sqrt{42}}\boldsymbol{v}_3 \right\}$ is an orthonormal set of vectors. ■

Theorem 2.9.7. *If $S = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p\} \subseteq \mathbb{R}^n$ is an orthogonal set of nonzero vectors, then S is linearly independent.*

The requirement that an orthogonal set contain only nonzero vectors is critical. Note that $\mathbf{0} \cdot \boldsymbol{v} = 0$ for any vector $\boldsymbol{v}$. Thus, as we defined an orthogonal pair, the zero vector is orthogonal to every vector. This is a truly exceptional quality shared by no other vector. For this reason, the zero vector must be treated definitely in the theory of orthogonality and it is thus excluded from orthogonal sets. While this may seem somewhat arbitrary, this is truly consequence driven. The main consequence is that Theorem 2.9.7 would fail if $\mathbf{0}$ were allowed in. It should also be noted that for a singleton $\{\boldsymbol{v}\}$, this set is orthogonal if and only if $\boldsymbol{v} \neq 0$, the same condition that guarantees that $\{\boldsymbol{v}\}$ is linearly independent. Notice that the condition that all pairs be orthogonal is vacuously true in this case as there are no pairs which are not orthogonal (since there are no pairs at all). This also holds for the empty set $\emptyset$. It has no pairs of distinct vectors which are not orthogonal, hence, the empty set is an orthogonal set.

An orthogonal (orthonormal) spanning set of $\mathbb{R}^n$ is necessarily a basis, called an **orthogonal (orthonormal) basis**. For example, the standard basis in $\mathbb{R}^n$ is an orthonormal basis.

2.9.3 Orthogonal Projections

Theorem 2.9.8 (Parseval's Identity)**.** *Let $S = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p\} \subseteq \mathbb{R}^n$ be an orthogonal subset. If $\boldsymbol{y}$ is a linear combination of the vectors in S, that is,*

$$\boldsymbol{y} = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \ldots + c_p\boldsymbol{v}_p,$$

then

$$c_i = \frac{\boldsymbol{v}_i \cdot \boldsymbol{y}}{\boldsymbol{v}_i \cdot \boldsymbol{v}_i}, \quad \textit{called the } \textbf{\textit{Fourier coefficients}}.$$

Proof. Taking the dot product, we get

$$\boldsymbol{v}_i \cdot \boldsymbol{y} = \boldsymbol{v}_i \cdot (c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \ldots + c_p\boldsymbol{v}_p) = c_i(\boldsymbol{v}_i \cdot \boldsymbol{v}_i).$$

Since $\boldsymbol{v}_i \cdot \boldsymbol{v}_i \neq 0$, dividing both sides by $\boldsymbol{v}_i \cdot \boldsymbol{v}_i$ gives the formula. □

Example 2.9.9. Let $S = \{\boldsymbol{v}_1 = (1,2,3), \boldsymbol{v}_2 = (1,1,-1), \boldsymbol{v}_3 = (-5,4,-1)\}$, which was shown in Example 2.9.6 to be an orthogonal subset of $\mathbb{R}^3$. By Theorem 2.9.7, we see that S is, in fact, an orthogonal basis for $\mathbb{R}^3$. Let $\boldsymbol{y} = (-4,8,10)$. Since $\boldsymbol{y} \in \mathbb{R}^3$, $\boldsymbol{y}$ is a linear combination of the vectors of S. By Theorem 2.9.8, we have

$$\boldsymbol{y} = \left(\frac{\boldsymbol{v}_1 \cdot \boldsymbol{y}}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1}\right)\boldsymbol{v}_1 + \left(\frac{\boldsymbol{v}_2 \cdot \boldsymbol{y}}{\boldsymbol{v}_2 \cdot \boldsymbol{v}_2}\right)\boldsymbol{v}_2 + \left(\frac{\boldsymbol{v}_3 \cdot \boldsymbol{y}}{\boldsymbol{v}_3 \cdot \boldsymbol{v}_3}\right)\boldsymbol{v}_3 = \frac{42}{14}\boldsymbol{v}_1 + \frac{-6}{3}\boldsymbol{v}_2 + \frac{42}{42}\boldsymbol{v}_3 = 3\boldsymbol{v}_1 - 2\boldsymbol{v}_2 + \boldsymbol{v}_3$$

$$= 3\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} - 2\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + \begin{bmatrix} -5 \\ 4 \\ -1 \end{bmatrix} = \begin{bmatrix} -4 \\ 8 \\ 10 \end{bmatrix}. \quad \blacksquare$$

Definition 2.9.10. Let $\boldsymbol{y} \in \mathbb{R}^n$ and $W \leq \mathbb{R}^n$. Let $\{\boldsymbol{u}_1,\ \boldsymbol{u}_2,\ \ldots, \boldsymbol{u}_r\}$ be an orthogonal basis for W. Then the **orthogonal projection** of $\boldsymbol{y}$ onto W is

$$\widehat{\boldsymbol{y}} = \text{proj}_W \boldsymbol{y} = \sum_{i=1}^{r} \frac{\boldsymbol{u}_i \cdot \boldsymbol{y}}{\boldsymbol{u}_i \cdot \boldsymbol{u}_i}\boldsymbol{u}_i = \frac{\boldsymbol{u}_1 \cdot \boldsymbol{y}}{\boldsymbol{u}_1 \cdot \boldsymbol{u}_1}\boldsymbol{u}_1 + \frac{\boldsymbol{u}_2 \cdot \boldsymbol{y}}{\boldsymbol{u}_2 \cdot \boldsymbol{u}_2}\boldsymbol{u}_2 + \ldots + \frac{\boldsymbol{u}_r \cdot \boldsymbol{y}}{\boldsymbol{u}_r \cdot \boldsymbol{u}_r}\boldsymbol{u}_r.$$

When the subspace W is clear from context, $\text{proj}_W(\boldsymbol{y})$ is often abbreviated as $\widehat{\boldsymbol{y}}$.

Let $\boldsymbol{y},\ \boldsymbol{u} \in \mathbb{R}^n$ and $\boldsymbol{u} \neq \boldsymbol{0}$. Then the **orthogonal projection** of $\boldsymbol{y}$ onto $\boldsymbol{u}$ is $\text{proj}_{\boldsymbol{u}} \boldsymbol{y} = \text{proj}_W \boldsymbol{y}$, where $W = \text{Span}(\boldsymbol{u})$.

Example 2.9.11. Let $\boldsymbol{y} = \begin{bmatrix} 1 \\ 7 \\ 3 \end{bmatrix}$ and $\boldsymbol{u} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$. Then

$$\widehat{\boldsymbol{y}} = \text{proj}_{\boldsymbol{u}}(\boldsymbol{y}) = \frac{\boldsymbol{u} \cdot \boldsymbol{y}}{\boldsymbol{u} \cdot \boldsymbol{u}}\boldsymbol{u} = \frac{12}{6}\begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix}.$$

Note that

$$\boldsymbol{y} - \widehat{\boldsymbol{y}} = \begin{bmatrix} 1 \\ 7 \\ 3 \end{bmatrix} - \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} -3 \\ 5 \\ 1 \end{bmatrix}, \qquad \boldsymbol{y} = \widehat{\boldsymbol{y}} + (\boldsymbol{y} - \widehat{\boldsymbol{y}}) = \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix} + \begin{bmatrix} -3 \\ 5 \\ 1 \end{bmatrix},$$

and that

$$\widehat{\boldsymbol{y}} \cdot (\boldsymbol{y} - \widehat{\boldsymbol{y}}) = \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} -3 \\ 5 \\ 1 \end{bmatrix} = -12 + 10 + 2 = 0.$$

Therefore, $\boldsymbol{y} - \widehat{\boldsymbol{y}}$ and $\widehat{\boldsymbol{y}}$ are orthogonal. $\blacksquare$

Theorem 2.9.12. *Let $\boldsymbol{y} \in \mathbb{R}^n$ and let $W \leq \mathbb{R}^n$. Then*

$$\text{proj}_W \boldsymbol{y} \cdot (\boldsymbol{y} - \text{proj}_W \boldsymbol{y}) = 0.$$

Exercises

(Go to Solutions)

For Exercises 1-6, determine whether the two given vectors are orthogonal or not. If not, adjust a value in one of the vectors to make them orthogonal. Answers may vary.

1. $\begin{bmatrix} 4 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 \\ -8 \end{bmatrix}$

2. $\begin{bmatrix} 6 \\ 1 \end{bmatrix}, \begin{bmatrix} 8 \\ -24 \end{bmatrix}$

3. $\begin{bmatrix} -3 \\ -5 \end{bmatrix}, \begin{bmatrix} 5 \\ -3 \end{bmatrix}$

4. $\begin{bmatrix} 2 \\ 8 \end{bmatrix}, \begin{bmatrix} 4 \\ 1 \end{bmatrix}$

5. $\begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}, \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}$

6. $\begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}, \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix}$

For Exercises 7-8, determine whether the set of vectors is orthogonal or not.

7. $\left\{ \begin{bmatrix} 3 \\ 5 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ -2 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 \\ 0 \\ -5 \end{bmatrix} \right\}$

8. $\left\{ \begin{bmatrix} 1 \\ 1 \\ 1 \\ 7 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} \right\}$

For Exercises 9-12, find a hyperplane in $\mathbb{R}^n$ containing the vector $\boldsymbol{x}_0$, listed first, and whose normal vector is $\boldsymbol{n}$, listed second.

♠ 9. $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 3 \\ -2 \\ 0 \end{bmatrix}$

♠ 10. $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \\ 2 \\ -1 \end{bmatrix}$

♠ 11. $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}, \begin{bmatrix} -3 \\ -6 \\ 3 \\ 2 \\ -5 \end{bmatrix}$

12. $\begin{bmatrix} 10 \\ -8 \\ 6 \\ -4 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 3 \\ 5 \\ 7 \\ 9 \end{bmatrix}$

For Exercises 13-17, use inner products to express the vector $\boldsymbol{y}$, listed first, as a linear combination of the orthogonal basis $\mathcal{B}$, listed second.

♠ 13. $\begin{bmatrix} -2 \\ -6 \end{bmatrix}, \left\{ \begin{bmatrix} 1 \\ 3 \end{bmatrix} \right\}$

♠ 14. $\begin{bmatrix} -1 \\ 10 \\ 11 \end{bmatrix}, \left\{ \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \right\}$

♠ 15. $\begin{bmatrix} 17 \\ -8 \\ 4 \end{bmatrix}, \left\{ \begin{bmatrix} 5 \\ 4 \\ -2 \end{bmatrix}, \begin{bmatrix} 2 \\ -2 \\ 1 \end{bmatrix} \right\}$

♠ 16. $\begin{bmatrix} -6 \\ 9 \\ 8 \\ -5 \end{bmatrix}, \left\{ \begin{bmatrix} 3 \\ 4 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 3 \\ -1 \\ -5 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \right\}$

♠ 17. $\begin{bmatrix} -19 \\ 18 \\ -12 \\ -15 \\ -13 \end{bmatrix}, \left\{ \begin{bmatrix} 1 \\ 6 \\ -6 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 7 \\ -2 \\ 0 \\ 5 \\ 5 \end{bmatrix} \right\}$

18. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/ortho.ipynb

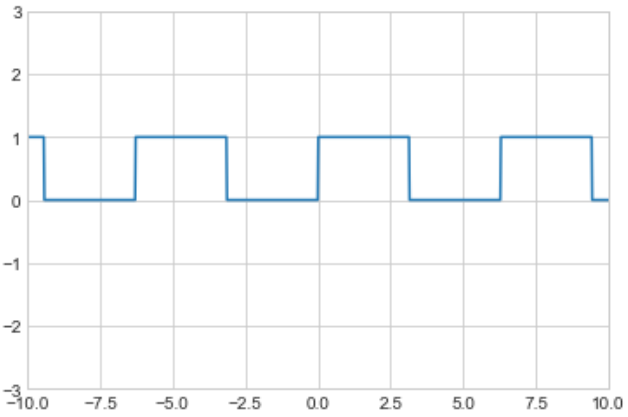
Python Projects

1. A Fourier series is an infinite sum of the form

$$g(x) = c_0 + \sum_{n=1}^{\infty}(a_n \cos(nx) + b_n \sin(nx)).$$

They have many applications in mathematics, including the study of periodic functions. Consider the square-wave function defined recursively as

$$f(x) = \begin{cases} 1 & 0 \le x < \pi \\ 0 & \pi \le x < 2\pi \\ f(x+2\pi) & = f(x) \end{cases}.$$



The graph of the square-wave is given to the right. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/fourier.py

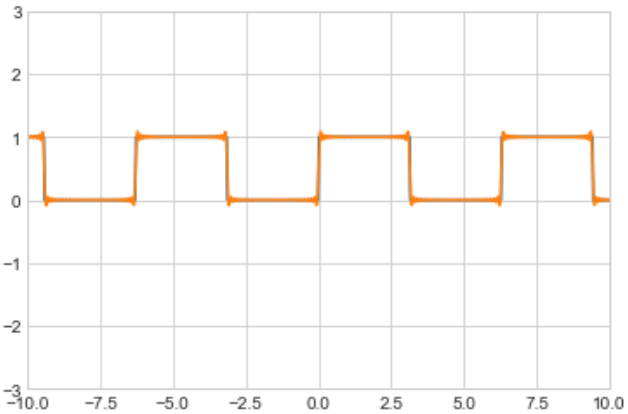
Using Calculus, one can show that

$$g(x) = \frac{1}{2} + c_1 \sin(x) + c_2 \sin(3x) + c_3 \sin(5x) + c_4 \sin(7x) + \ldots + c_{25} \sin(49x)$$

will make an excellent Fourier approximation for $f(x)$, assuming we can determine the coefficients c_i. While Calculus also provides a method for computing these coefficients, we will approximate the Calculus-method by setting up and solving a linear system, such as explained in Section 4.2.

(a) Choose 25 points on the graph of f and set up a linear system that will fit $g(x)$ to these 25 points.

(b) Graph your Fourier approximation on the same screen as the graph of the square-wave to determine how good your estimate is. Below you will find the graph of the Fourier approximation of f using the Calculus-method alluded to above. You will seek to produce such a plot, although I do not expect yours to be as good as this one, as you will NOT be using the Calculus-method here.



2. In this project, you will write Python functions to implement the three elementary row operations, such as explained in Section 2.2.1. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/elementary_row_operations.py

(a) One of the three elementary row operation is *scaling*, which scales a row R_i of an augmented matrix $[A \mid \boldsymbol{b}]$ by a nonzero scalar c, namely $\mathbb{R}_i \mapsto cR_i$. Write a Python function `row_scaling(coefficient_matrix, augmented_matrix)` which takes two input parameters:[ii] `coefficient_matrix` is A and `augmented_matrix` is $\boldsymbol{b}$ from a linear system $[A \mid \boldsymbol{b}]$, in which we will scale R_i by `scalar`, and returns two output values: `scaled_coefficient_matrix` and `scaled_augmented_matrix`, which are the original ma-

[ii] Note that the function takes in two input and produces two output to recognize the partition inside an augmented matrix. While this may make the current implementation slightly more difficult than ignoring the partition, this will make future code more efficient since we need not row reduce beyond the partition.

trices A and $\boldsymbol{b}$ now with $\mathbb{R}_i \mapsto cR_i$. Verify your code on the example:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{array}\right] \quad 3(\text{Row } 2) \quad \sim \quad \left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 15 & 18 & 21 & 24 \\ 9 & 10 & 11 & 12 \end{array}\right].$$

(b) One of the three elementary row operation is *replacement*, which replaces a row R_i of an augmented matrix $[A \mid \boldsymbol{b}]$ with $R_i + cR_j$, namely $R_i \mapsto R_i + cR_j$. Write a Python function `row_replacement(coefficient_matrix, augmented_matrix, scalar, i, j)` which takes five input parameters:[ii] `coefficient_matrix` is A and `augmented_matrix` is $\boldsymbol{b}$ from a linear system $[A \mid \boldsymbol{b}]$, in which we will scale R_j by scalar and it to R_i, and returns two output values: `replaced_coefficient_matrix` and `replaced_augmented_matrix`, which are the original matrices A and $\boldsymbol{b}$ now with $R_i \mapsto R_i + cR_j$. Verify your code on the example:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{array}\right] \quad \text{Row } 3 - 9(\text{Row } 1) \quad \sim \quad \left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 0 & -8 & -16 & -24 \end{array}\right].$$

(c) One of the three elementary row operation is *interchange*, which interchanges the order of rows R_i and R_j of an augmented matrix $[A \mid \boldsymbol{b}]$, namely $R_i \leftrightarrow R_j$. Write a Python function `row_interchange(coefficient_matrix, augmented_matrix, i, j)` which takes two input parameters:[ii] `coefficient_matrix` is A and `augmented_matrix` is $\boldsymbol{b}$ from a linear system $[A \mid \boldsymbol{b}]$, in which we interchange R_i and R_j, and returns two output values: `interchanged_coefficient_matrix` and `interchanged_augmented_matrix`, which are the original matrices A and $\boldsymbol{b}$ now with $R_i \leftrightarrow R_j$. Verify your code on the example:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{array}\right] \quad \sim \quad \left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \\ 5 & 6 & 7 & 8 \end{array}\right].$$

3. In this project, you continue the program of the previous project and write Python functions to implement row reduction, such as explained in Sections 2.3 and 2.7.1. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/row_reduction.py

(a) The Forward Phase Algorithm transforms an augmented matrix into echelon form. This is done recursively using three procedures. The first is to locate the next pivot (we will not consider this procedure in this project). The second is, once the pivot position is located, to place a nonzero entry in the pivot position if there is a zero there. Lastly, the third procedure is to place a zero into every entry below the pivot. Write a Python function `zero_below(coefficient_matrix, constant_matrix, pivot)` which takes input parameters:[ii] `coefficient_matrix` is A and `constant_matrix` is $\boldsymbol{b}$ from a linear system $[A \mid \boldsymbol{b}]$, as well as the location of the current `pivot` position (i, j), entered as an ordered pair. First, the function needs to check to see whether $a_{i,j} = 0$ or not. If true, then it must interchange with a nonzero row below the pivot position, using `row_interchange`. Next, the function will use `row_replacement` to make each entry below the pivot a zero. After this, the function will output the modified versions of A and $\boldsymbol{b}$. Verify your code on the example:

$$\left[\begin{array}{ccc|c} \boxed{0} & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{array}\right] \sim \left[\begin{array}{ccc|c} \boxed{5} & 6 & 7 & 8 \\ 0 & 2 & 3 & 4 \\ 0 & -0.8 & -1.6 & -2.4 \end{array}\right].$$

(b) The Backward Phase Algorithm transforms an echelon matrix into row reduced echelon form. This is done recursively using three procedures. The first is to locate the next pivot (we will not consider this procedure in this project). The second is, once the pivot position is located, to scale the leading entry into a 1. Lastly, the third procedure is to place a zero into every entry above the pivot. Write a Python function `zero_above(coefficient_matrix, constant_matrix, pivot)` which takes input parameters:[ii] `coefficient_matrix` is A and `constant_matrix` is $\boldsymbol{b}$ from a linear system $[A \mid \boldsymbol{b}]$, as well as the location of the current `pivot` position (i, j), entered as an ordered pair. First, the function needs to scale the pivot row, using `row_scaling`, so that the leading entry is 1. Next, the function will use `row_replacement` to make each entry above the pivot a zero. After this, the function will output the modified versions of A and $\boldsymbol{b}$. Verify your code on the example:

$$\left[\begin{array}{ccc|c} 5 & 6 & 7 & 8 \\ 0 & 2 & 3 & 4 \\ 0 & -0.8 & \boxed{-1.6} & -2.4 \end{array}\right] \sim \left[\begin{array}{ccc|c} 5 & 2.5 & 0 & -2.5 \\ 0 & 0.5 & 0 & -0.5 \\ 0 & 0.5 & \boxed{1} & 1.5 \end{array}\right].$$

4. In this project, you continue the program of the previous project and write Python functions to compute an echelon form and determine linear independence, such as explained in Section 2.3.1. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/echelon_form.py

(a) The Forward Phase Algorithm transforms an augmented matrix into echelon form. This is done recursively using three procedures. The first is to locate the next pivot. The second is, once the pivot position is located, to place a nonzero entry in the pivot position if there is a zero there. Lastly, the third procedure is to place a zero into every entry below the pivot. Write a Python function `forward_phase(coefficient_matrix, constant_matrix)` which takes input parameters:[ii] `coefficient_matrix` is A and `constant_matrix` is $\boldsymbol{b}$ from a linear system $[A \mid \boldsymbol{b}]$. The function will return three objects: `reduced_coefficient_matrix`, `reduced_costant_matrix`, and `pivots`. `reduced_coefficient_matrix` and `reduced_constant_matrix` will be row-reduced versions of `coefficient_matrix` and `constant_matrix`, respectively, accomplished by repeated usage of `zero_below` for each pivot column. Note that `reduced_coefficient_matrix` must be in echelon form, but not necessarily row-reduced echelon form. `pivots` will be the set of pivot positions, listed as ordered pairs of the leading entries in `reduced_coefficient_matrix`. Verify your code on the example:

$$\left[\begin{array}{ccc} \boxed{1} & 1 & -5 \\ 2 & 1 & 4 \\ 3 & -1 & -1 \end{array}\right] \sim \left[\begin{array}{ccc} \boxed{1} & 1 & -5 \\ 0 & -1 & 14 \\ 0 & -4 & 14 \end{array}\right] \sim \left[\begin{array}{ccc} \boxed{1} & 1 & -5 \\ 0 & -1 & 14 \\ 0 & 0 & -42 \end{array}\right].$$

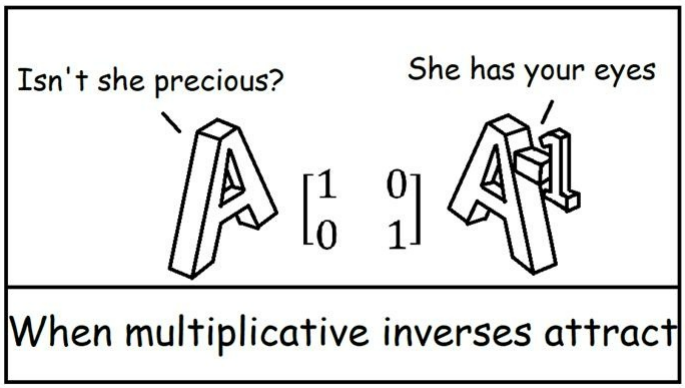
(b) Write a Python function `IsLinearIndep(vectors)` which takes input parameter: `vectors` a set of n-vectors. The function will return `True` if the set is linearly independent and will return `False` if the set is linearly dependent. This will be accomplished using the `forward_phase` function just written. This function must use your `forward_phase` function. Verify your code on Example 2.7.5.

Chapter 3

The Algebra and Geometry of Matrices

Up to this point, we have used matrices for one purpose, to encode information about linear objects. We first introduced the augmented matrix in Section 2.2 to do exactly that. In Chapter 1 we started using matrices to represent a set of vectors in a slightly more compact way. This happened in Section 2.6, where we introduce the matrix-vector product so that the matrix equation (2.6.1) could encode the vector equation (2.5.1). This allowed us to define linear transformations using matrices.

But we also introduced the the column and null spaces of a matrix. These vector spaces came about essentially by solving problems about spanning and linear independence of vectors but we attributed the spaces to the matrix, not a collection of vectors. This was our first inkling that matrices deserve to be studied in their own right, not just as tools to better understand vectors. In fact, we will turn this paradigm on its head. Viewing column vectors just as $n \times 1$ matrices, all we have studied about vectors can actually be viewed as a subset of the theory of matrices. We can add and scale matrices just like column vectors. For this reason, we can actually view matrices as just beefier versions of column vectors, which can lead to discussions about vector spaces of matrices. But matrices can transcend the vector operations as we will introduce a general matrix multiplication, which will be the foundation for this chapter on matrices.



"The thing that scares us the most is when familiar things operate in unfamiliar ways." – Noah Hawley

3.1 Matrix Operations

In many ways, matrices behave just like vectors, that is, we can add them and scale them. In that vein, we can view matrices as block vectors. On the other hand, the 2-dimensional shape of matrices offers new opeations, such as matrix multiplications, which make the algebra of matrices so much more involved compared to just vectors.

3.1.1 Linear Combinations of Matrices

Viewing an $m \times n$ matrix as a block m-vector with all entries equal to n-vectors, that is, a matrix is an array of its row vectors, we can define operations on matrices analogous to the extended vector operations onto block vectors.

We say that two $m \times n$ matrices $A = \begin{bmatrix} a_{ij} \end{bmatrix}$ and $B = \begin{bmatrix} b_{ij} \end{bmatrix}$ are **equal** if $a_{ij} = b_{ij}$ for all i and j (for all $a_{ij}, b_{ij} \in \mathbb{R}$). We can **add** matrices term-wise, that is,

$$A + B = \begin{bmatrix} a_{ij} + b_{ij} \end{bmatrix}.$$

Finally, we can **multiply** matrices by a **scalar** term-wise, that is,

$$cA = \begin{bmatrix} ca_{ij} \end{bmatrix} \qquad \text{for all } c \in \mathbb{R}.$$

Example 3.1.1. Let $A = \begin{bmatrix} 3 & 9 & 1 \\ -2 & 4 & 6 \end{bmatrix}$, $B = \begin{bmatrix} 0 & 5 & 6 \\ 3 & 1 & 1 \end{bmatrix}$, and $C = \begin{bmatrix} 2 & 1 \\ 0 & 4 \end{bmatrix}$.

Then

$$A + B = \begin{bmatrix} 3+0 & 9+5 & 1+6 \\ -2+3 & 4+1 & 6+1 \end{bmatrix} = \begin{bmatrix} 3 & 14 & 7 \\ 1 & 5 & 7 \end{bmatrix}.$$

Now, $A + C$ is not possible since the matrices have different sizes.

Next,

$$2B = 2\begin{bmatrix} 0 & 5 & 6 \\ 3 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 10 & 12 \\ 6 & 2 & 2 \end{bmatrix},$$

and

$$A - 2B = \begin{bmatrix} 3 & 9 & 1 \\ -2 & 4 & 6 \end{bmatrix} - \begin{bmatrix} 0 & 10 & 12 \\ 6 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & -1 & -11 \\ -8 & 2 & 4 \end{bmatrix}. \qquad \blacksquare$$

Because we can add and scale matrices, we can actually view matrices as vector themselves. Let $\mathbb{R}$ be a field. Then $\mathbb{R}^{m \times n}$ will denote the set of $m \times n$ matrices with entries from $\mathbb{R}$, which is a vector space. When working with matrices, we will let $0_{m,n}$ denote an $m \times n$ matrix with all entries equal to zero. This is called the **zero matrix**. Likewise, we will let $J_{m,n}$ denote the $m \times n$ matrix with all entries equal to one. When the context is clear, these subscripts may be dropped.

Let E_{ij} be the matrix whose entry in the (i, j)th entry is a one and all other entries are zero. Then $\mathcal{E} = \{E_{1,1}, E_{1,2}, \ldots, E_{1,n}, E_{2,1}, E_{2,2}, \ldots, E_{2,n}, \ldots, E_{m,1}, E_{m,2}, \ldots, E_{m,n}\}$ forms the **standard basis** of $\mathbb{R}^{m \times n}$.

The matrices $E_{i,j}$ are often called the unit matrices. Using coordinate vectors, we see that $\mathbb{R}^{m\times n} \cong \mathbb{R}^{mn}$, since $|\mathcal{E}| = mn$.

Definition 3.1.2. The **diagonal entries** in an $m \times n$ matrix $A = \begin{bmatrix} a_{ij} \end{bmatrix}$ are the entries a_{11}, a_{22}, $a_{33}, \ldots$, that is, the entries a_{ii}, and they form the **main diagonal**.

If A is an $n \times n$ matrix, then we say that A is a **square matrix**. A **diagonal matrix** is a square matrix whose only nonzero entries are on the main diagonal.

Let I_n denote the $n \times n$ matrix whose entries are 1's across the diagonal and 0's everywhere else. This matrix is known as the **identity matrix**.

Example 3.1.3. The identity matrices I_2, I_3, and I_4 are listed below, respectively.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad \blacksquare$$

Similar to how we defined block vectors, one can also define **block matrices**, which are matrices whose entries are themselves matrices. As matrices themselves can be viewed as a block vector (a matrix is a vector of row (or column) vectors), this is a natural generalization. Unlike block vectors though, as a matrix is a 2-dimensional array special care is required on the shapes of individual blocks in the matrix to make sure the block matrix is rectangular in shape also.

Defining a norm on a matrix is a bit more difficult because there are several different notions of the norm for matrices. As we have discussed before with vectors, from the norm is derived the distance function, so it is important to establish a norm to that distance and other statistics can be calculated on matrices like we did with vectors. So, we will adopt essentially the same norm that we had on vectors for matrices (called the **Frobenius norm**), understanding that this norm could be replaced with a better norm if the circumstances warrant such. More specifically,

$$\|A\| = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{i,j}^2}.$$

Note that this is the same norm on the $m \times n$ matrix A if we instead viewed A as a block mn-vector. It is satisfies all the previous properties we have seen for norms as it extends the previous setting with vectors toward the collection of matrices.

3.1.2 Matrix Multiplication

Definition 3.1.4. If A is an $m \times n$ matrix and B is an $n \times p$ matrix with column vectors

$$B = \begin{bmatrix} \boldsymbol{b}_1 & \boldsymbol{b}_2 & \ldots & \boldsymbol{b}_p \end{bmatrix},$$

then the **matrix product**

$$AB = A\begin{bmatrix} \boldsymbol{b}_1 & \boldsymbol{b}_2 & \ldots & \boldsymbol{b}_p \end{bmatrix} = \begin{bmatrix} A\boldsymbol{b}_1 & A\boldsymbol{b}_2 & \ldots & A\boldsymbol{b}_p \end{bmatrix}.$$

The matrix AB is a $m \times p$ matrix.

If A is an $n \times n$ matrix, then $A^k = \underbrace{A \cdots A}_{k}$. We let $A^0 = I_n$.

Example 3.1.5. Compute AB with the given matrices $A = \begin{bmatrix} 2 & 1 & -1 \\ 0 & 4 & -2 \end{bmatrix}$ and $B = \begin{bmatrix} 9 & -5 & -3 \\ 3 & 9 & 1 \\ -2 & 4 & 6 \end{bmatrix}$.

$$AB = \begin{bmatrix} A\begin{bmatrix} 9 \\ 3 \\ -2 \end{bmatrix} & A\begin{bmatrix} -5 \\ 9 \\ 4 \end{bmatrix} & A\begin{bmatrix} -3 \\ 1 \\ 6 \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} 2(9)+1(3)-1(-2) & 2(-5)+1(9)-1(4) & 2(-3)+1(1)-1(6) \\ 0(9)+4(3)-2(-2) & 0(-5)+4(9)-2(4) & 0(-3)+4(1)-2(6) \end{bmatrix}$$

$$= \begin{bmatrix} 18+3+2 & -10+9-4 & -6+1-6 \\ 0+12+4 & 0+36-8 & 0+4-12 \end{bmatrix} = \begin{bmatrix} 23 & -5 & -11 \\ 16 & 28 & -8 \end{bmatrix},$$

which is a 2×3 matrix. ∎

Matrix multiplication can also be defined with the seemingly complicated formula,

$$AB = \begin{bmatrix} \sum_{k=1}^{n} a_{ik}b_{kj} \end{bmatrix} = \begin{bmatrix} a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj} \end{bmatrix} \qquad \text{for } A = \begin{bmatrix} a_{ik} \end{bmatrix} \text{ and } B = \begin{bmatrix} b_{kj} \end{bmatrix},$$

which is none other than the "finger-multiplication" we learned with the matrix-vector product.

Example 3.1.6. Compute AB with the given matrices $A = \begin{bmatrix} 3 & -2 \\ 5 & 3 \end{bmatrix}$ and $B = \begin{bmatrix} -1 & 4 \\ 7 & -6 \end{bmatrix}$.

$$\begin{bmatrix} 3 & -2 \\ 5 & 3 \end{bmatrix}\begin{bmatrix} -1 & 4 \\ 7 & -6 \end{bmatrix} = \begin{bmatrix} 3(-1)-2(7) & \\ & \end{bmatrix}$$

$$\begin{bmatrix} 3 & -2 \\ 5 & 3 \end{bmatrix}\begin{bmatrix} -1 & 4 \\ 7 & -6 \end{bmatrix} = \begin{bmatrix} -17 & 3(4)-2(-6) \\ & \end{bmatrix}$$

$$\begin{bmatrix} 3 & -2 \\ 5 & 3 \end{bmatrix}\begin{bmatrix} -1 & 4 \\ 7 & -6 \end{bmatrix} = \begin{bmatrix} -17 & 24 \\ 5(-1)+3(7) & \end{bmatrix}$$

$$\begin{bmatrix} 3 & -2 \\ 5 & 3 \end{bmatrix}\begin{bmatrix} -1 & 4 \\ 7 & -6 \end{bmatrix} = \begin{bmatrix} -17 & 24 \\ 16 & 5(4)+3(-6) \end{bmatrix}$$

Hence,

$$\begin{bmatrix} 3 & -2 \\ 5 & 3 \end{bmatrix}\begin{bmatrix} -1 & 4 \\ 7 & -6 \end{bmatrix} = \begin{bmatrix} -17 & 24 \\ 16 & 2 \end{bmatrix}.$$

∎

Example 3.1.7. Let $A = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & -3 \\ 1 & 1 \end{bmatrix}$. Then

$$AB = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 0 & -3 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ 3 & 6 \end{bmatrix} \quad \text{and} \quad BA = \begin{bmatrix} 0 & -3 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix} = \begin{bmatrix} 3 & -9 \\ 0 & 5 \end{bmatrix}.$$

Therefore, $AB \neq BA$. ∎

In general, matrix multiplication is a noncommutative operations.

3.1.3 The Transpose of a Matrix

Definition 3.1.8. Let $A = \begin{bmatrix} a_{ij} \end{bmatrix}$ be an $m \times n$ matrix. Then the **transpose** of A, denoted $A^\top$, is the $n \times m$ matrix given by $A^\top = \begin{bmatrix} a_{ji} \end{bmatrix}$, that is, the matrix whose columns are formed from the corresponding rows of A. We say a square matrix A is **symmetric** if $A^\top = A$.

Example 3.1.9. Let

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 1 \\ 3 & 5 & 7 \end{bmatrix}, \text{ and } C = \begin{bmatrix} 2 & -3 \\ -3 & 1 \end{bmatrix}.$$

Then

$$A^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}, \quad B^\top = \begin{bmatrix} 1 & 3 \\ 1 & 5 \\ 1 & 7 \end{bmatrix}, \text{ and } C^\top = \begin{bmatrix} 2 & -3 \\ -3 & 1 \end{bmatrix}.$$

Note that $C^\top = C$, hence C is symmetric. ∎

Theorem 3.1.10. *Let A and B be matrices with sizes such that the indicated sums and products are defined. Let $c \in \mathbb{R}$.*

(i) $(A + B)^\top = A^\top + B^\top$

(ii) $(cA)^\top = cA^\top$

(iii) $(A^\top)^\top = A$

(iv) $(AB)^\top = B^\top A^\top$

Definition 3.1.11. Let A be an $m \times n$ matrix. Then its **Gram matrix** is $G = A^\top A$, which is necessarily an $n \times n$ matrix.

Note that a Gram matrix is always symmetric, even if A is not. To see this, note

$$G^\top = (A^\top A)^\top = A^\top (A^\top)^\top = A^\top A = G.$$

Example 3.1.12. Let $A = \begin{bmatrix} 1 & -2 & 4 \\ 3 & 0 & -5 \end{bmatrix}$. Then

$$A^\top A = \begin{bmatrix} 1 & 3 \\ -2 & 0 \\ 4 & -5 \end{bmatrix} \begin{bmatrix} 1 & -2 & 4 \\ 3 & 0 & -5 \end{bmatrix} = \begin{bmatrix} 10 & -2 & -11 \\ -2 & 4 & -8 \\ -11 & -8 & 41 \end{bmatrix}$$ ■

3.1.4 The Trace of a Matrix

Definition 3.1.13. If A is a square matrix, then the **trace** of A, denoted $\mathrm{tr}(A)$, is the sum of the diagonal entries of A.

Example 3.1.14. Let $A = \begin{bmatrix} 2 & 0 \\ 1 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 9 & 3 & -2 & 3 \\ 28 & 9 & 4 & 0 \\ 16 & 1 & 6 & 3 \\ 2 & 5 & 2 & 1 \end{bmatrix}$. Then

$$\mathrm{tr}(A) = 2 + 4 = \boxed{6}, \qquad \mathrm{tr}(B) = 9 + 9 + 6 + 1 = \boxed{25}.$$ ■

Python Programming

We add to the matrix commands we saw previously, such as in Section 2.2. For example, the same Python commands for matrix-vector multiplication can be used for the generalized matrix multiplication, i.e. `A @ B` produces the product of the matrices A and B. In the special case that A is a square matrix and the product is A^k, one can use `np.linalg.matrix_power(A,k)`.

In Python, we can implement the $n \times n$ identity matrix I_n by the command `np.identity(n)`. Likewise, we can implement the all zeros matrix $0_{m,n}$ and the all ones matrix $J_{m,n}$ of shape $m \times n$ with the commands `np.zeros((m,n))` and `np.ones((m,n))`, respectively. Do not forget the double parentheses. They are important for Python. We can generate the $n \times n$ diagonal matrix whose diagonal entries are given by $\boldsymbol{d} = (d_1, d_2, \ldots, d_n)$ using the command `np.diag(d)`. Conversely, if A is a matrix, then `np.diag(A)` returns the diagonal entries of A as an n-vector.

In Python, to compute the transpose of a matrix A use the commands `A.T` or `np.transpose(A)`. To compute the trace of A use `np.trace(A)`.

In Python, a matrix can be constructed using a block decomposition. When generating the matrix, instead of using `np.array(...)` use `np.block(...)`. Inside the parenthesis, include a matrix where each entry in the matrix is itself a matrix.

Example 3.1.15. The code:

```
import numpy as np

B = np.array([[1,2,3,4],[5,6,7,8]]) #2x4 block
C = np.array([[9,10],[11,12]]) #2x2 block
D = np.array([[13,14,15,16],[17,18,19,20],[21,22,23,24]]) #3x4 block
E = np.array([[25,26],[27,28],[29,30]]) #3x2 block

# 2x2 block matrix
A = np.block([[B,C],[D,E]])
print(A.shape,"\n", A, "\n")
print(A.T.shape,"\n", A.T, "\n")
print("Main Diagonal = ", np.diag(A), "\n")

A_2x15 = np.reshape(A,(2,15))
print(A_2x15.shape,"\n", A_2x15, "\n")
```

produces the output:

```
(5, 6)
[[ 1  2  3  4  9 10]
 [ 5  6  7  8 11 12]
 [13 14 15 16 25 26]
 [17 18 19 20 27 28]
 [21 22 23 24 29 30]]

(6, 5)
[[ 1  5 13 17 21]
 [ 2  6 14 18 22]
 [ 3  7 15 19 23]
 [ 4  8 16 20 24]
 [ 9 11 25 27 29]
 [10 12 26 28 30]]

Main Diagonal = [ 1 6 15 20 29]

(2, 15)
[[ 1  2  3  4  9 10  5  6  7  8 11 12 13 14 15]
 [16 25 26 17 18 19 20 27 28 21 22 23 24 29 30]]
```

■

Exercises

(Go to Solutions)

1. Write the matrix $E_{1,3} \in \mathbb{R}^{3\times 3}$.

For Exercises 2-11, using the matrices listed below, perform the matrix calculation:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & -3 & 0 \\ 1 & -2 & -1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 5 & 6 \\ -5 & -5 & 2 \\ 2 & 0 & -3 \end{bmatrix}, \quad C = \begin{bmatrix} 3 & 0 & -1 & 2 \\ 1 & 7 & 8 & -3 \\ 3 & 3 & 2 & -4 \end{bmatrix}, \quad D = \begin{bmatrix} 2 & 3 & -2 \\ -5 & 0 & 7 \\ 0 & -2 & 4 \\ 1 & 2 & 3 \end{bmatrix}.$$

♠ 2. $2A + B^\top$ ♠ 3. $4A - 3B$ ♠ 4. AC ♠ 5. $A^\top$ ♠ 6. $C^\top$ ♠ 7. $D^\top$

♠ 8. $\text{tr}(A)$ ♠ 9. $\text{tr}(B)$ ♠ 10. $DB^\top$ ♠ 11. $A^2 + 2A - 5I_3$

For Exercises 12-15, using the matrices listed below, perform the matrix calculation:

$$A = \begin{bmatrix} -1 & 0 & 2 \\ 1 & 3 & 5 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 1 \\ 2 & 2 \\ 3 & 4 \end{bmatrix}, \quad C = \begin{bmatrix} -2 & 0 \\ 3 & -2 \end{bmatrix}.$$

12. $2A^\top + B$ 13. AB 14. BC 15. ABC

For Exercises 16-18, for the matrix A provided, find $\text{tr}(A)$, $A^\top$, and $\text{tr}(A^\top)$.

16. $\begin{bmatrix} 8 & -7 & 2 \\ 0 & -2 & 4 \\ 3 & 2 & 1 \end{bmatrix}$

17. $\begin{bmatrix} 1/8 & -7/8 & 2/5 & -12/13 \\ 9/8 & -1/2 & 1/4 & 1/4 \\ 1 & -12/15 & 2 & 1/5 \\ 1/4 & 8/15 & -3/8 & 3/8 \end{bmatrix}$

18. $\begin{bmatrix} 5 & 1 & 6 & 2 \\ 3 & 0 & 3 & 4 \\ 5 & 2 & 1 & 1 \\ 2 & 3 & 0 & 0 \end{bmatrix} \pmod 7$

19. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/matrix.ipynb

"I actually work better within restrictions. When you leave everything wide open, things tend to get a little convoluted. So when you give me those restrictions and I start to use my brain creatively to work around those, that's when things get interesting." – Kenny Omega

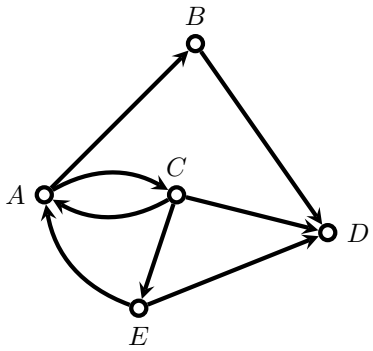
3.2 Matrix Applications

We have seen many examples previously where data naturally takes the shape of a matrix, such as tables, images, and arrays of vectors. It is natural to represent data via a matrix whenever there are two parameters in play, i.e. $1 \leq i \leq m$ and $1 \leq j \leq n$. This could be weather patterns are multiple locations, a bill of materials from multiple suppliers, or a stock portfolio for multiple clients. It is too difficult to list all the possible uses of a matrix in data representation.

3.2.1 Graphs and Adjacency Matrices

Definition 3.2.1. Let $\Gamma = (\{v_1, v_2, \ldots, v_n\}, E)$ be a digraph. Let $A(\Gamma) = [a_{ij}]$ be the $n \times n$ matrix such that a_{ij} is the number of edges from v_i to v_j. This matrix is called the **adjacency matrix** of Γ.

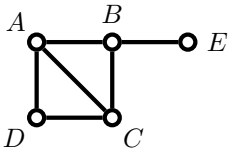
Example 3.2.2. Consider the graph Γ, illustrated below. Its adjacency matrix is displayed below right.



$$A(\Gamma) = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

■

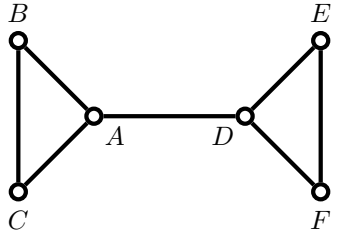
Example 3.2.3. Consider the graph Γ, illustrated below. Its adjacency matrix is displayed below right.



$$A(\Gamma) = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

■

Example 3.2.4. Consider the graph Γ, illustrated below. Its adjacency matrix is displayed below right.



$$A(\Gamma) = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

■

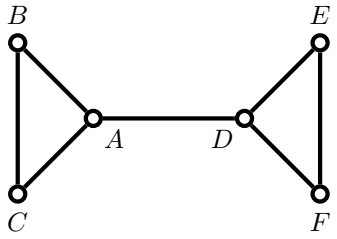
By construction, an adjacency matrix is necessarily symmetric.

Definition 3.2.5. Let Γ be a graph. A **walk** on Γ is a sequence of adjacent vertices.

Example 3.2.6. Consider the graph Γ, illustrated to the right. Then some examples of walks starting at A are:

$ABCAD$, ADF, $ADADADADE$, $ACABACB$.

The sequence $ADBCA$ is not a walk since there is no edge from D to B.



■

Proposition 3.2.7. *The number of walks from v_i to v_j in a graph Γ of length k is equal to (i,j)-entry in A^k, where $A = A(\Gamma)$.*

Example 3.2.8. Consider the graph Γ from Example 3.2.4. Its adjacency matrix A and some of its powers: A^2, A^3, A^4, are displayed below.

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 3 & 1 & 1 & 0 & 1 & 1 \\ 1 & 2 & 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 1 & 3 & 1 & 1 \\ 1 & 0 & 0 & 1 & 2 & 1 \\ 1 & 0 & 0 & 1 & 1 & 2 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 2 & 4 & 4 & 5 & 1 & 1 \\ 4 & 2 & 3 & 1 & 1 & 1 \\ 4 & 3 & 2 & 1 & 1 & 1 \\ 5 & 1 & 1 & 2 & 4 & 4 \\ 1 & 1 & 1 & 4 & 2 & 3 \\ 1 & 1 & 1 & 4 & 3 & 2 \end{bmatrix}, \quad A^4 = \begin{bmatrix} 13 & 6 & 6 & 4 & 6 & 6 \\ 6 & 7 & 6 & 6 & 2 & 2 \\ 6 & 6 & 7 & 6 & 2 & 2 \\ 4 & 6 & 6 & 13 & 6 & 6 \\ 6 & 2 & 2 & 6 & 7 & 6 \\ 6 & 2 & 2 & 6 & 6 & 7 \end{bmatrix}.$$

These matrices tell us that for walks from A to B there is 1 walk of length 1 (AB), 1 walk of length 2 (ACB), four walks of length 3 ($ABAB$, $ACAB$, $ADAB$, and $ABCB$), and six walks of length 4 ($ABCAB$, $ACBAB$, $ADACB$, $ABACB$, $ACACB$, $ACBCB$) ■

3.2.2 Selectors

Definition 3.2.9. A **permutation matrix** is a square matrix formed by some rearrangement of the rows of I_n. A **selector matrix** is some submatrix of a permutation matrix.

Example 3.2.10. Let $P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_2^\top \\ \boldsymbol{e}_4^\top \\ \boldsymbol{e}_1^\top \\ \boldsymbol{e}_3^\top \end{bmatrix}$ is a permutation matrix. Note that for $\boldsymbol{x}^\top = (x_1, x_2, x_3, x_4)$, we have

$$P\boldsymbol{x} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_2 \\ x_4 \\ x_1 \\ x_3 \end{bmatrix}.$$

Thus, multiplication by a permutation matrix on a vector has the affect of rearranging the entries of the vector $\boldsymbol{x}$ in the same manner as the rearrangement of the rows of the identity matrix.

The *reverser matrix* is the permutation matrix that puts the standard basis in reverse order, namely

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_n^\top \\ \boldsymbol{e}_{n-1}^\top \\ \vdots \\ \boldsymbol{e}_2^\top \\ \boldsymbol{e}_1^\top \end{bmatrix}.$$

Note that

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} x_n \\ x_{n-1} \\ \vdots \\ x_2 \\ x_1 \end{bmatrix}. \qquad \blacksquare$$

Example 3.2.11. Let $\boldsymbol{t}$ be an n-time series, such as an audio file. A *down-sample* by a factor of k is the subvector of $\boldsymbol{t}$ where only the kth entries are retained , where $n = jk$. This is captured by multiplying by the $j \times n$ matrix

$$\begin{bmatrix} \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \boldsymbol{e}_k & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \boldsymbol{e}_{2k} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \boldsymbol{e}_{jk} \end{bmatrix}^\top$$

For example, if $\boldsymbol{t} = (t_1, t_2, t_3, t_4, t_5, t_6)^\top$,

then the 2-down sample would be

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{bmatrix} = \begin{bmatrix} t_2 \\ t_4 \\ t_6 \end{bmatrix},$$

and the 3-down sample would be

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{bmatrix} = \begin{bmatrix} t_3 \\ t_6 \end{bmatrix}. \qquad \blacksquare$$

Example 3.2.12. Let $\boldsymbol{x}$ be an n-vector. Then the $k \times n$ matrix $\begin{bmatrix} 0 & I_k & 0 \end{bmatrix}$ is a *cropping matrix* which only retains the middle k-entries of $\boldsymbol{x}$ (based upon the zero matrices before and after I_k). For example,

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix}. \qquad \blacksquare$$

3.2.3 Vector Convolution

Definition 3.2.13. The **convolution** of an m-vector $\boldsymbol{a}$ and an n-vector $\boldsymbol{b}$ is the $(m+n-1)$-vector $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$, with entries

$$c_k = \sum_{i+j=k+1} a_i b_j.$$

For example,

$$\begin{aligned} c_1 &= a_1 b_1 \\ c_2 &= a_1 b_2 + a_2 b_1 \\ c_3 &= a_1 b_3 + a_2 b_2 + a_3 b_1 \\ c_4 &= a_1 b_4 + a_2 b_3 + a_3 b_2 + a_4 b_1 \\ c_5 &= a_1 b_5 + a_2 b_4 + a_3 b_3 + a_4 b_2 + a_5 b_1 \\ &\vdots \end{aligned}$$

Convolution on the surface seems like a strange operation to introduce, but believe it or not. It is a very natural calculation. For example, if $p(x) = a_1 + a_2x + \ldots + a_mx^{m-1}$ and $q(x) = b_1 + b_2x + \ldots + b_nx^{n-1}$, then we can represent these polynomials as vectors: $\boldsymbol{a} = (a_0, a_1, \ldots, a_{m-1})$, $\boldsymbol{b} = (b_0, b_1, \ldots, b_{n-1})$. Then the vector for the coefficients of the product $p(x)q(x)$ is exactly $\boldsymbol{a} * \boldsymbol{b}$. As such $\boldsymbol{a} * \boldsymbol{b} = \boldsymbol{b} * \boldsymbol{a}$, $\boldsymbol{a} * (\boldsymbol{b} * \boldsymbol{c}) = (\boldsymbol{a} * \boldsymbol{b}) * \boldsymbol{c}$, and $\boldsymbol{a} * \boldsymbol{b} = \boldsymbol{0}$ if and only if $\boldsymbol{a} = \boldsymbol{0}$ or $\boldsymbol{b} = \boldsymbol{0}$.

Example 3.2.14. Let $\boldsymbol{a} = (1, 0, -1)$ and $\boldsymbol{b} = (2, 1, -1)$. Then

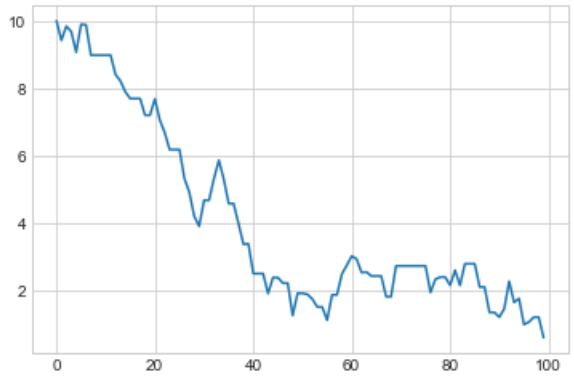
$$a * b = (1(2), 1(1) + 0(2), 1(-1) + 0(1) - 1(2), 0(-1) - 1(1), -1(-1)) = (2, 1, -3, -1, 1).$$

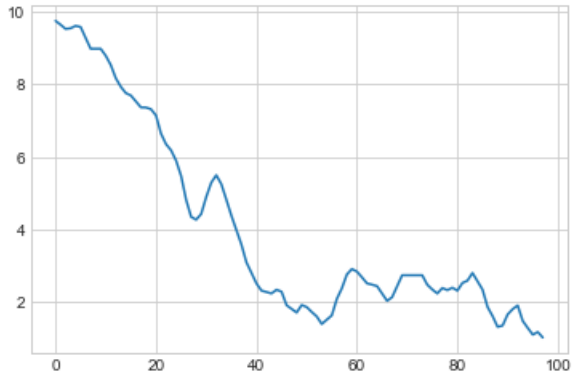
Conversely, if $p(x) = 1 - x^2$ and $q(x) = 2 + x - x^2$, then their respective coefficient vectors are $\boldsymbol{a}$ and $\boldsymbol{b}$. Likewise,

$$p(x)q(x) = (1 - x^2)(2 + x - x^2) = (2 + x - x^2) - (2x^2 + x^3 - x^4) = 2 + x - 3x^2 - x^3 + x^4.$$

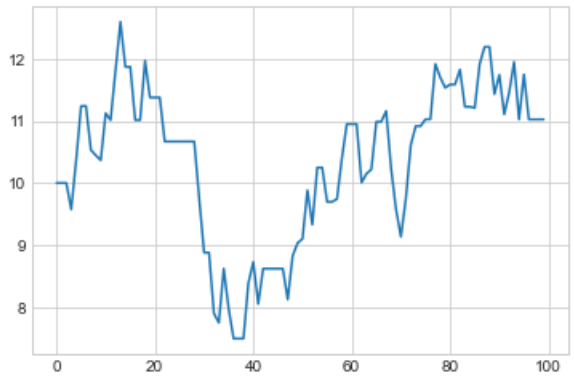
Note that the coefficient vector of $p(x)q(x)$ is $(2, 1, -3, -1, 1) = \boldsymbol{a} * \boldsymbol{b}$. ∎

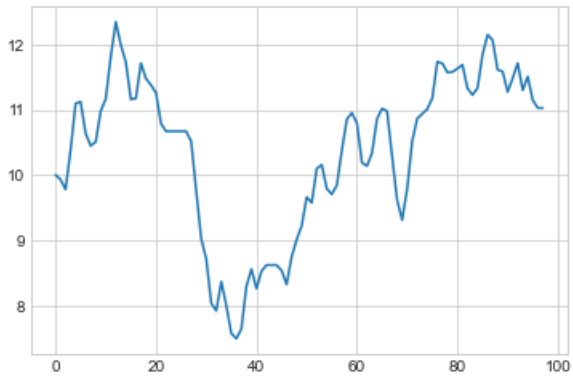
Example 3.2.15. Let $\boldsymbol{t}$ be an time series vector and let $\boldsymbol{p}$ be a probability vector (that is, $p_i \geq 0$ and $p_1 + p_2 + \ldots + p_m = 1$. Then $\boldsymbol{t} * \boldsymbol{p}$ is a smoothing of the time series using weights from $\boldsymbol{p}$ to smooth out the time series. A common probability vector to use is $\boldsymbol{p} = (1/3, 1/3, 1/3)$ which replaces t_i with the mean of t_{i-1}, t_i, t_{i+1}. Such as in the following diagram.





Other probability can be used of course. Consider instead $\boldsymbol{p} = (1/6, 2/3, 1/6)$ which places twice the weight on the concurrent value.





You will likely want to slice off the first and last entries (or so) because there might not be any elements to compare the entire probability vector against. ∎

We can see that convolution is a special case of matrix multiplication. Let $T(\boldsymbol{a})$ be the $(m + n - 1) \times n$ *Toeplitz matrix* given by the formula

$$T(\boldsymbol{a})_{ij} = \begin{cases} a_{i-j+1} & 1 \leq i - j + 1 \leq m \\ 0 & \text{otherwise.} \end{cases}$$

Then $T(\boldsymbol{a})\boldsymbol{b} = \boldsymbol{a} * \boldsymbol{b}$.

Example 3.2.16. Let $\boldsymbol{a} = (a_1, a_2, a_3, a_4)$. Then the 6×4 Toeplitz matrix $T(\boldsymbol{a})$ is given as

$$T(\boldsymbol{a}) = \begin{bmatrix} a_1 & 0 & 0 \\ a_2 & a_1 & 0 \\ a_3 & a_2 & a_1 \\ a_4 & a_3 & a_2 \\ 0 & a_4 & a_3 \\ 0 & 0 & a_4 \end{bmatrix}. \qquad \blacksquare$$

Python Programming

The convolution of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ can be computed by the command `np.convolve(a,b)`.

Example 3.2.17. The following code:

```
import numpy as np

X = [10]                                # initialize X

# add 99 more random points weighted near to the last point.
# this is so that we can create a simulated trend without needing to rely on real data.
for i in range(99):
    X.append(X[i]+np.random.random()*(np.random.randint(3)-1))

# convolve the data to make it softer by probability vector [1/3, 1/3, 1/3]
# 'same' is a mode of np.convolve(a,b), which returns output of length max(len(a),len(b))
# the by default mode is 'full', which gives the full convolution of length len(a)+len(b)-1
# 'same' trims off positions from the start and end of 'full' where points do not overlap
# since the probability vector is here length 3, 'same' trims off index 0 and -1
Y = np.convolve(X,[1/3, 1/3, 1/3],'same')

# when using 'same', boundary effects are still visible.
# we can correct these boundary effects by anchoring to X[0] and X[-1]
Y[0],Y[-1] = X[0],X[-1]

import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-whitegrid') # use 'seaborn-whitegrid' if error with plt

fig, axs = plt.subplots(2, 1, figsize=(8, 6))# makes it so we can see both graphs at once

# First (pre-convolved) plot
axs[0].plot(range(100), X)
axs[0].set_title("Original Graph (paramter 1 'X')")

# Second (convolved) plot
axs[1].plot(range(100),Y)
axs[1].set_title("Smoothed Graph")

plt.tight_layout()
plt.show()
```

generated the line graphs of time series and their convolutions in this section. ∎

Exercises

(Go to Solutions)

♠ 1. Given an n-vector $\boldsymbol{x}$, find an $(n-2)\times n$ matrix A which *trims the vectors*, that is, $A\boldsymbol{x} = (x_2, \ldots, x_{n-1})$.

♠ 2. Given a time series $\boldsymbol{t}$ of even length n, find an $\dfrac{n}{2} \times n$ matrix A which *down samples* the time series by a factor of 2, that is, $A\boldsymbol{t} = (t_2, t_4, t_6, \ldots, t_n)$.

♠ 3. Given a time series $\boldsymbol{t}$ of length n, find a $(2n-1) \times n$ matrix A which *up samples* the time series by a factor of 2, that is, $A\boldsymbol{t} = \left(t_1, \dfrac{t_1+t_2}{2}, t_2, \dfrac{t_2+t_3}{2}, t_3, \ldots, t_{n-1}, \dfrac{t_{n-1}+t_n}{2}, t_n\right)$.

4. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/matrixapp.ipynb

"My happiness grows in direct proportion to my acceptance, and in inverse proportion to my expectations."
– Michael J. Fox

3.3 Matrix Inverses

When can we divide by a matrix? Does it even make sense to talk about matrix division? It depends on the matrix. In this section, we explore this very question, leading to the notion of a nonsingular matrix.

3.3.1 Nonsingular Matrices

Definition 3.3.1. An $n \times n$ matrix A is **nonsingular** (or **invertible**) if there exists an $n \times n$ matrix B such that

$$AB = BA = I_n.$$

In this case, B is called an **inverse** of A. If A is not nonsingular, then it is **singular**.

Example 3.3.2. Let $A = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$ and $C = \begin{bmatrix} 5 & -3 \\ -3 & 2 \end{bmatrix}$. Then

$$AC = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} 5 & -3 \\ -3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and

$$CA = \begin{bmatrix} 5 & -3 \\ -3 & 2 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Thus, A is invertible with inverse C. ∎

We should mention that inverses are unique. Suppose that A is invertible with inverses B and C. Thus,

$$B = BI_n = B(AC) = (BA)C = I_nC = C.$$

We will denote the inverse of A as A^{-1}. Therefore,

$$AA^{-1} = A^{-1}A = I_n.$$

Theorem 3.3.3. *Let A and B be $n \times n$ invertible matrices, let m be a positive integer, and c is a nonzero scalar. Then A^{-1}, AB, $A^\top$, A^m, and cA are also invertible with:*

(i) $(A^{-1})^{-1} = A$ *(ii)* $(AB)^{-1} = B^{-1}A^{-1}$ *(iii)* $(A^\top)^{-1} = (A^{-1})^\top$

(iv) $(A^m)^{-1} = (A^{-1})^m := A^{-m}$ *(v)* $(cA)^{-1} = \frac{1}{c}A^{-1}$.

Note that (ii) is another example of the *Shoe-Sock Principle*.[i]

Corollary 3.3.4. *If A is invertible, then $AA^\top$ and $A^\top A$ are likewise invertible.*

[i] To remember this formula, think of the following proverb:

Put Your Socks On, Then Your Shoes
Take Your Shoes Off, Then Your Socks

3.3.2 The Inversion Algorithm

Theorem 3.3.5 (Inversion Algorithm)**.** *Let A be an invertible $n \times n$ matrix A. Then any sequence of elementary row operations that reduces A to I_n also transforms I_n into A^{-1}.*

The Inversion Algorithm is a process for compute matrix inverses. Consider the matrix $[A \mid I_n]$. By row reduction,

$$[A \mid I_n] \sim [I_n \mid A^{-1}]. \tag{3.3.1}$$

Thus, row reduction saves the day again!

Example 3.3.6. Find the inverse of $A = \begin{bmatrix} 0 & 1 & -3 \\ 1 & -2 & 5 \\ -5 & 4 & 3 \end{bmatrix}$.

Using the method suggested above, we will row reduce A and apply these same row operations to I_n.

$$\left[\begin{array}{ccc|ccc} \boxed{0} & 1 & -3 & 1 & 0 & 0 \\ 1 & -2 & 5 & 0 & 1 & 0 \\ -5 & 4 & 3 & 0 & 0 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|ccc} \boxed{1} & -2 & 5 & 0 & 1 & 0 \\ 0 & 1 & -3 & 1 & 0 & 0 \\ -5 & 4 & 3 & 0 & 0 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|ccc} 1 & -2 & 5 & 0 & 1 & 0 \\ 0 & \boxed{1} & -3 & 1 & 0 & 0 \\ 0 & -6 & 28 & 0 & 5 & 1 \end{array}\right]$$

$$\sim \left[\begin{array}{ccc|ccc} 1 & -2 & 5 & 0 & 1 & 0 \\ 0 & 1 & -3 & 1 & 0 & 0 \\ 0 & 0 & \boxed{10} & 6 & 5 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|ccc} 1 & -2 & 5 & 0 & 1 & 0 \\ 0 & 1 & -3 & 1 & 0 & 0 \\ 0 & 0 & \boxed{1} & \frac{3}{5} & \frac{1}{2} & \frac{1}{10} \end{array}\right] \sim \left[\begin{array}{ccc|ccc} 1 & -2 & 0 & -3 & -\frac{3}{2} & -\frac{1}{2} \\ 0 & 1 & -3 & 1 & 0 & 0 \\ 0 & 0 & \boxed{1} & \frac{3}{5} & \frac{1}{2} & \frac{1}{10} \end{array}\right]$$

$$\sim \left[\begin{array}{ccc|ccc} 1 & -2 & 0 & -3 & -\frac{3}{2} & -\frac{1}{2} \\ 0 & \boxed{1} & 0 & \frac{14}{5} & \frac{3}{2} & \frac{3}{10} \\ 0 & 0 & 1 & \frac{3}{5} & \frac{1}{2} & \frac{1}{10} \end{array}\right] \sim \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{13}{5} & \frac{3}{2} & \frac{1}{10} \\ 0 & 1 & 0 & \frac{14}{5} & \frac{3}{2} & \frac{3}{10} \\ 0 & 0 & 1 & \frac{3}{5} & \frac{1}{2} & \frac{1}{10} \end{array}\right]$$

Therefore, $A^{-1} = \begin{bmatrix} \frac{13}{5} & \frac{3}{2} & \frac{1}{10} \\ \frac{14}{5} & \frac{3}{2} & \frac{3}{10} \\ \frac{3}{5} & \frac{1}{2} & \frac{1}{10} \end{bmatrix} = \frac{1}{10}\begin{bmatrix} 26 & 15 & 1 \\ 28 & 15 & 3 \\ 6 & 5 & 1 \end{bmatrix}$. ∎

3.3.3 Left Inverses

Definition 3.3.7. Let A be an $m \times n$ matrix. If there exists an $n \times m$ matrix B such that

$$BA = I_n$$

then we say that A is **left-invertible**.[ii] The matrix B is called the **left inverse** of A.

[ii] We say that A is **right-invertible** if there exists an $n \times m$ matrix B suh that $AB = I_n$, in which case B is called a **right inverse** of A. The theory of right inverses is essentially the same as left inverses since A is left-invertible if and only if $A^\top$ is right-invertible, in which case every right inverse of $A^\top$ is $B^\top$ where B is a left inverse of A.

Example 3.3.8. Let $A = \begin{bmatrix} -3 & -4 \\ 4 & 6 \\ 1 & 1 \end{bmatrix}$. Then let $B = \frac{1}{9}\begin{bmatrix} -11 & -10 & 16 \\ 7 & 8 & -11 \end{bmatrix}$. Then

$$BA = \frac{1}{9}\begin{bmatrix} -11 & -10 & 16 \\ 7 & 8 & -11 \end{bmatrix}\begin{bmatrix} -3 & -4 \\ 4 & 6 \\ 1 & 1 \end{bmatrix} = \frac{1}{9}\begin{bmatrix} 33-40+16 & 44-60+16 \\ -21+32-11 & -28+48-11 \end{bmatrix} = \frac{1}{9}\begin{bmatrix} 9 & 0 \\ 0 & 9 \end{bmatrix} = I_2.$$

Thus, B is a left inverse of A. On the other hand, let $C = \frac{1}{2}\begin{bmatrix} 0 & -1 & 6 \\ 0 & 1 & -4 \end{bmatrix}$. Then

$$CA = \frac{1}{2}\begin{bmatrix} 0 & -1 & 6 \\ 0 & 1 & -4 \end{bmatrix}\begin{bmatrix} -3 & -4 \\ 4 & 6 \\ 1 & 1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0-4+6 & 0-6+6 \\ 0+4-4 & 0+6-4 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = I_2.$$

Therefore, C is also a left inverse of A, which shows that when a left inverse exists then it is not necessarily unique. ∎

Example 3.3.9. Let $\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ be a column vector such that $\boldsymbol{x} \neq \boldsymbol{0}$. Then there exists some i such that $x_i \neq 0$. Then $\left(\frac{1}{x_i}\right)\boldsymbol{e}_i^\top \boldsymbol{x} = 0 + \ldots + 0 + \frac{x_i}{x_i} + 0 \ldots 0 = 1$. Therefore, $\left(\frac{1}{x_i}\right)\boldsymbol{e}_i^\top$ is a left inverse for $\boldsymbol{x}$. ∎

Suppose that linear system $A\boldsymbol{x} = \boldsymbol{b}$ is consistent, that is, it has a solution. If A is left-invertible, say with left inverse C, then the linear system $A\boldsymbol{x} = \boldsymbol{b}$ will have a unique solution, namely $\boldsymbol{x} = C\boldsymbol{b}$. To see this, note that if we multiply both sides of $A\boldsymbol{x} = \boldsymbol{b}$ on the left by C we get $C(A\boldsymbol{x}) = C\boldsymbol{b}$. Simplifying the left hand side, we get $(CA)\boldsymbol{x} = I_n\boldsymbol{x} = \boldsymbol{x}$. Additionally, if B is another left inverse of A, we still get the same solution, that is, $B\boldsymbol{b} = C\boldsymbol{b}$. Of course, if $A\boldsymbol{x} = \boldsymbol{b}$ is inconsistent, then $C\boldsymbol{b}$ is not a solution then. We can acutally check then that $A(C\boldsymbol{x}) \neq \boldsymbol{b}$. Thus, for left-invertible matrices, there is a simple check to see if the linear system is consistent or inconsistent.

Theorem 3.3.10. *An $m \times n$ matrix A is left-invertible if and only if the column vectors of A are linearly independent. In particular, if $m < n$, then A is not left-invertible.*

Let A be an $m \times n$ matrix with linearly independent columns. Then A must be left-invertible, but how do we find a left inverse of A? As there could be multiple left inverses, it is important to know that we do not need all of them, just one. We know that the Gram matrix $A^\top A$ is necessarily nonsingular in this case. Hence, $(A^\top A)^{-1}$ exists.

Definition 3.3.11. Let A be a matrix with linearly independent columns. Then its **(Moore-Penrose) pseudo-inverse**[iii] is given as

$$A^\dagger = (A^\top A)^{-1}A^\top.$$

[iii] In the case where A has linearly independent rows, A is then right-invertible and its right inverse is given as $A^\dagger = A^\top(AA^\top)^{-1}$.

Note that

$$A^\dagger A = [(A^\top A)^{-1}A^\top]A = (A^\top A)^{-1}(A^\top A) = I.$$

Hence, $A^\dagger$ is a left inverse of A in this case.

Let us now return to the situation that A is an $n \times n$ matrix. If A is left-invertible, then its left inverse B is likewise $n \times n$. Thus, $BA = I_n$. What about AB? We saw in Theorem 3.3.10, that A is left-invertible if and only if the columns of A are independent. Thus, the column vectors of A form a basis for $\mathbb{R}^n$. In particular, the unit vector $\boldsymbol{e}_i$ can be expressed as a linear combination of the columns of A. Say $\boldsymbol{e}_i = A\boldsymbol{c}_i$ for some coefficient vector $\boldsymbol{c}_i \in \mathbb{R}^n$. Let $C = \begin{bmatrix} \boldsymbol{c}_1 & \boldsymbol{c}_2 & \cdots & \boldsymbol{c}_n \end{bmatrix}$. Then $AC = \begin{bmatrix} A\boldsymbol{c}_1 & A\boldsymbol{c}_2 & \cdots & A\boldsymbol{c}_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_1 & \boldsymbol{e}_2 & \cdots & \boldsymbol{e}_n \end{bmatrix} = I_n$. This means that A is right-invertible, that is, it has a right inverse C since $AC = I_n$. Like observed above, if a matrix A has a left inverse B and a right inverse C, then $B = C$, that is, the left and right inverses are the same. As the left and right inverses in the above argument were arbitrary, this also means that A has only one left inverse, only one right inverse, and these matrices are the same. Thus, a left-invertible (or a right-invertible) square matrix is nonsingular and $A^\dagger = A^{-1}$.

Theorem 3.3.12 (The Nonsingular Matrix Theorem)**.** *A square matrix is nonsingular if and only if its column vectors are linearly independent if and only if its row vectors are linearly independent.*

Python Programming

The inverse A^{-1} of a nonsingular matrix A can be computed by the command `np.linalg.inv(A)`. The pseudo-inverse $A^{\dagger}$ of an $m \times n$ matrix A with linearly independent columns can be computed by the command `np.linalg.pinv(A)`. The error "`Singular Matrix`" will appear if the columns of A are not linearly independent in either case.

Exercises

(Go to Solutions)

For Exercises 1-3, solve linear system $A\boldsymbol{x} = \boldsymbol{b}$ using the matrix inverse.

1. $A = \begin{bmatrix} 4 & 8 \\ 2 & 6 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$

♠ 2. $A = \begin{bmatrix} 1 & 2 & 3 \\ -1 & -1 & 1 \\ 2 & 1 & -5 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} 0 \\ 3 \\ -2 \end{bmatrix}$, $A^{-1} = \begin{bmatrix} 4 & 13 & 5 \\ -3 & -11 & -4 \\ 1 & 3 & 1 \end{bmatrix}$

♠ 3. $A = \begin{bmatrix} 4 & 26 & 31 & -15 \\ -3 & -21 & -25 & 12 \\ 1 & 7 & 8 & -4 \\ 0 & -1 & -1 & 1 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$, $A^{-1} = \begin{bmatrix} 3 & 4 & 1 & 1 \\ -1 & 0 & 4 & 1 \\ 0 & -1 & -3 & 0 \\ -1 & -1 & 1 & 2 \end{bmatrix}$

For Exercises 4-6, solve linear system $A\boldsymbol{x} = \boldsymbol{b}$ using the left-inverse.

4. $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ -1 & 2 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}$, $A^{\dagger} = \frac{1}{2}\begin{bmatrix} 1 & 0 & -1 \\ -2 & 1 & 1 \end{bmatrix}$

5. $A = \begin{bmatrix} 2 & 4 \\ 0 & 8 \\ 4 & 12 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} -2 \\ -24 \\ -16 \end{bmatrix}$, $A^{\dagger} = \frac{1}{12}\begin{bmatrix} 2 & -4 & 2 \\ 6 & 0 & -3 \end{bmatrix}$

6. $A = \begin{bmatrix} -1 & 0 & -1 \\ 0 & 2 & 1 \\ 2 & -2 & 0 \\ -1 & 0 & 2 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} -4 \\ 7 \\ -2 \\ 5 \end{bmatrix}$, $A^{\dagger} = \frac{1}{88}\begin{bmatrix} -22 & 22 & 22 & -22 \\ -7 & 31 & -13 & -19 \\ -16 & 8 & 8 & 32 \end{bmatrix}$

For Exercises 7-11, find the inverse matrix of the provided matrix, if possible. Verify your answer.

7. $\begin{bmatrix} 6 & 2 \\ 12 & 4 \end{bmatrix}$

8. $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 2 & 16 \end{bmatrix}$

9. $\begin{bmatrix} 4 & -3 & -1 \\ 1 & -1 & 0 \\ -2 & -1 & 2 \end{bmatrix}$

♠ 10. $\begin{bmatrix} 10 & -1 & -6 \\ -11 & 2 & 9 \\ -3 & 1 & 3 \end{bmatrix}$

11. $\begin{bmatrix} \frac{1}{2} & -1 & -\frac{3}{2} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{3}{2} \\ -\frac{3}{2} & 3 & -\frac{7}{2} \end{bmatrix}$

12. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/nonsingular.ipynb

"Don't spend time beating on a wall, hoping to transform it into a door." – Coco Chanel

3.4 Linear Transformations

The next important linear structure we have seen before (primarily in calculus) is the notion of a linear operator *or a* ***linear transformation****. This is a function on vectors which preserve linear combinations.*

3.4.1 Linear Transformations

Definition 3.4.1. Let X and Y be vector space over a field $\mathbb{R}$. A function $T : X \to Y$ is a **linear transformation** if:

(i) $T(\boldsymbol{u} + \boldsymbol{v}) = T(\boldsymbol{u}) + T(\boldsymbol{v})$, for all vectors $\boldsymbol{u}, \boldsymbol{v} \in X$;

(ii) $T(c\boldsymbol{u}) = cT(\boldsymbol{u})$, for all vectors $\forall \boldsymbol{u} \in X$ and scalars $c \in \mathbb{R}$.

Recall that X and Y are called the **domain** and **codomain** of T, respectively. For any $\boldsymbol{x} \in X$, we call $T(\boldsymbol{x})$ the **image** of $T\boldsymbol{x}$, and we call $\text{im}(T) = \{T(\boldsymbol{x}) \mid \boldsymbol{x} \in X\}$ (the set of images) the **image** (or **range**) of T. The **kernel** of a linear transformation T is the set of vectors which map to the zero vector, that is, $\ker T = \{\boldsymbol{x} \mid T(\boldsymbol{x}) = \boldsymbol{0}\}$, denoted $\ker(T)$.

Example 3.4.2. In calculus, there are many linear operators. For example, the derivative is a linear transformation since

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}[f(x)] + \frac{d}{dx}[g(x)] \qquad \text{and} \qquad \frac{d}{dx}[cf(x)] = c\frac{d}{dx}[f(x)].$$

Note that $\ker\left(\dfrac{d}{dx}\right)$ is the set of constant functions. Likewise, limits are linear operators:

$$\lim_{x\to a}[f(x) + g(x)] = \lim_{x\to a}[f(x)] + \lim_{x\to a}[g(x)] \qquad \text{and} \qquad \lim_{x\to a}[cf(x)] = c\lim_{x\to a}[f(x)].$$

Likewise, antiderivatives, definite integrals, indefinite integrals, and series are all linear operators on functions. Linear algebra was everywhere in calculus, we just didn't know it! ∎

Proposition 3.4.3. *Let $T : X \to Y$ be a linear transformation. Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in X$ and $c_1, \ldots, c_n \in \mathbb{R}$. Then*

$$T(c_1\boldsymbol{x}_1 + \ldots + c_n\boldsymbol{x}_n) = c_1T(\boldsymbol{x}_1) + \ldots + c_nT(\boldsymbol{x}_n). \tag{3.4.1}$$

The converse also is true, that is, T is a linear transformation if it preserves linear combinations. In particular, $T(\boldsymbol{0}) = \boldsymbol{0}$.

Example 3.4.4. Some examples of linear transformations are:

- *the negation function*: $T : \mathbb{R}^n \to \mathbb{R}^n : T(\boldsymbol{x}) = -\boldsymbol{x}$,
- *reversal function*: $T : \mathbb{R}^n \to \mathbb{R}^n : T(x_1, x_2, \ldots, x_n) = (x_n, \ldots, x_2, x_1)$,
- *the running sum function*: $T : \mathbb{R}^n \to \mathbb{R}^n : T(x_1, x_2, x_3, \ldots, x_n) = (x_1, x_1 + x_2, x_1 + x_2 + x_3, \ldots, x_1 + x_2 + x_3 + \ldots + x_n)$,
- *the de-meaning function*: $T : \mathbb{R}^n \to \mathbb{R}^n : T(\boldsymbol{x}) = \boldsymbol{x} - \text{avg}(\boldsymbol{x})\boldsymbol{1} = \widetilde{\boldsymbol{x}}$,
- *linear functionals*[i]: for a fixed $\boldsymbol{a} \in \mathbb{R}^n$, we define the linear function $f_{\boldsymbol{a}} : \mathbb{R}^n \to \mathbb{R} : f_{\boldsymbol{a}}(\boldsymbol{x}) = \boldsymbol{a} \cdot \boldsymbol{x}$. ∎

[i]Sometimes called *linear forms*, *1-forms*, or *covectors*. It is important to note that every linear transformation of the form $T : \mathbb{R}^n \to \mathbb{R}$ is equivalent to $f_{\boldsymbol{a}}$ for some vector $\boldsymbol{a}$. This is a special case of the matrix representation theorem we see below.

Example 3.4.5. Some examples of functions which are NOT:

- *the norm function*: $T : \mathbb{R}^n \to \mathbb{R} : T(\boldsymbol{x}) = \|\boldsymbol{x}\|$, not that in general $\|\boldsymbol{x} + \boldsymbol{y}\| \neq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$, such as
$$\|(1,0)\| + \|(0,1)\| = 1 + 1 = 2 \neq \sqrt{2} = \|(1,1)\|,$$
- *max function*: $T : \mathbb{R}^n \to \mathbb{R} : T(x_1, x_2, \ldots, x_n) = \max(x_n, \ldots, x_2, x_1)$,
$$\max(1,0) + \max(0,2) = 1 + 2 = 3 \neq 2 = \max(1,2),$$
- *the sorting function*: $T : \mathbb{R}^n \to \mathbb{R}^n : T(x_1, x_2, \ldots, x_n) = \text{sort}(x_n, \ldots, x_2, x_1)$,
$$\text{sort}(1,0) + \text{sort}(0,2) = (0,1) + (0,2) = (0,3) \neq (1,2) = \text{sort}(1,2),$$
- *standard deviation*: $T : \mathbb{R}^n \to \mathbb{R} : T(\boldsymbol{x}) = \text{std}(\boldsymbol{x})$,
- *the squaring function*: $T : \mathbb{R}^n \to \mathbb{R}^n : T(x_1, \ldots, x_n) = (x_1^2, \ldots, x_n^2)$. ∎

3.4.2 The Standard Matrix of a Linear Transformation

Example 3.4.6. Let $\boldsymbol{e}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\boldsymbol{e}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Suppose that $T : \mathbb{R}^2 \to \mathbb{R}^3$ is a linear transformation such that

$$T(\boldsymbol{e}_1) = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \text{and} \quad T(\boldsymbol{e}_2) = \begin{bmatrix} 3 \\ -5 \\ 0 \end{bmatrix}.$$

With no additional information, find a formula for the image of an arbitrary vector $\boldsymbol{x} \in \mathbb{R}^2$.

Since $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1\boldsymbol{e}_1 + x_2\boldsymbol{e}_2$ and since T is a linear transformation, it must hold that

$$\begin{aligned} T(\boldsymbol{x}) &= T(x_1\boldsymbol{e}_1 + x_2\boldsymbol{e}_2) = x_1T(\boldsymbol{e}_1) + x_2T(\boldsymbol{e}_2) \\ &= x_1\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + x_2\begin{bmatrix} 3 \\ -5 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 + 3x_2 \\ 2x_1 - 5x_2 \\ 3x_1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & -5 \\ 3 & 0 \end{bmatrix}\boldsymbol{x}. \end{aligned}$$

In particular, T can be represented as a matrix transformation. ∎

The argument applied in the previous example works in general.

Theorem 3.4.7. *Let $\boldsymbol{e}_i$ be the vector in $\mathbb{R}^n$ such that the ith entry is 1 and all other entries are 0. Let $T : \mathbb{R}^n \to \mathbb{R}^m$ be a linear transformation and let $A = \begin{bmatrix} T(\boldsymbol{e}_1) & T(\boldsymbol{e}_2) & \ldots & T(\boldsymbol{e}_n) \end{bmatrix}$. Let $\boldsymbol{x} \in \mathbb{R}^n$. Then*

$$T(\boldsymbol{x}) = A\boldsymbol{x}.$$

*This matrix A is called the **standard matrix for the linear transformation** T and is often denoted $[T]$.*

Example 3.4.8. Find the standard matrix A for the linear transformation $T(x_1,\ x_2) = (3x_1 + x_2,\ 2x_1 - 4x_2)$.

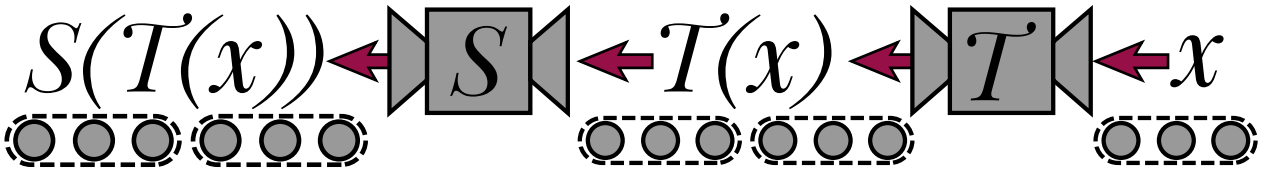
Since $T(\boldsymbol{e}_1) = (3,\ 2)$ and $T(\boldsymbol{e}_2) = (1,\ -4)$, we have that $[T] = A = \begin{bmatrix} 3 & 1 \\ 2 & -4 \end{bmatrix}$ and

$$T(\boldsymbol{x}) = A\boldsymbol{x} = \begin{bmatrix} 3 & 1 \\ 2 & -4 \end{bmatrix} \boldsymbol{x}.$$

■

3.4.3 Composition of Linear Transformations

Recall that if $S : \mathbb{R}^m \to \mathbb{R}^p$ and $T : \mathbb{R}^n \to \mathbb{R}^m$ are functions, then their composite $S \circ T$ is the function given by the rule $(S \circ T)(\boldsymbol{x}) = S(T(\boldsymbol{x}))$ where $\boldsymbol{x} \in \mathbb{R}^n$. Functions are often viewed, by parable, as machines on a conveyor belt. As an element $\boldsymbol{x}$ in the domain of T rolls into the machine T, it is transformed into the element $T(\boldsymbol{x})$. The composite $S \circ T$ is then the machine formed by placing the machines S and T next each other on the conveyor belt. The following theorem states that matrix representations are compatible with function composition.



Theorem 3.4.9. *Let $S : \mathbb{R}^m \to \mathbb{R}^p$ and $T : \mathbb{R}^n \to \mathbb{R}^m$ be linear transformations with standard matrices A and B, respectively. Let $\boldsymbol{x} \in \mathbb{R}^n$. Then*

$$S \circ T(\boldsymbol{x}) = S(T(\boldsymbol{x})) = AB\boldsymbol{x},$$

that is, the matrix which represents the composite of two linear transformations is the product of their matrix representations.

Exercises

(Go to Solutions)

For Exercises 1-3, use the transformation $T : \mathbb{R}^3 \to \mathbb{R}^2$ given by the rule:

$$T(x, y, z) = (x + y - 2z, -y + z).$$

1. Show that T is a linear transformation.

♠ 2. Compute $T(1, 2, 3)$ and $T(1, 0, -2)$.

♠ 3. Is $\boldsymbol{b} = (3, -1)$ in the image of T? Explain.

For Exercises 4-6, use the transformation $T : \mathbb{R}^3 \to \mathbb{R}^3$ given by the rule:

$$T(x, y, z) = (2z + y, 2x + 4, -2y).$$

4. Compute $T(1, 3, 2)$.

5. Is $\boldsymbol{b} = (1, 6, 2)$ in the image of T? Explain.

6. Show that T is NOT a linear transformation.

For Exercises 7-12, for the linear transformations T find its standard matrix $[T]$.

7. $T : \mathbb{R}^2 \to \mathbb{R}^2 : T(x_1, x_2) = (x_1 + 2x_2, x_2)$

8. $T : \mathbb{R}^2 \to \mathbb{R}^2 : T(x_1, x_2) = (2x_1 + 2x_2, x_1 + 3x_2)$

♠ 9. $T : \mathbb{R}^2 \to \mathbb{R}^2 : T(x_1, x_2) = (2x_1 - x_2, -4x_2)$

10. $T : \mathbb{R}^2 \to \mathbb{R}^2 : T(x_1, x_2, x_3) = (x_1 + x_2, 2x_1 + 3x_3, x_1 + 2x_2 + 2x_3)$

♠ 11. $T : \mathbb{R}^3 \to \mathbb{R}^3 :$

$$T\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 + 2x_2 + 3x_3 \\ 4x_1 + 5x_3 + 6x_2 \\ 7x_2 + 8x_1 + 9x_3 \end{bmatrix}$$

12. $T : \mathbb{R}^3 \to \mathbb{R}^4 :$

$$T\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 + 2x_2 + x_3 \\ 3x_1 + 14x_3 - 5x_2 \\ 3x_2 - 5x_1 - 18x_3 \\ 19x_2 - 7x_1 - 40x_3 \end{bmatrix}$$

For Exercises 13-14, find the standard matrix of the composite function $S \circ T$.

13. $T : \mathbb{R}^2 \to \mathbb{R}^2 : T(x_1, x_2) = (x_1 + 2x_2)$ and $S : \mathbb{R}^2 \to \mathbb{R}^2 : S(y_1, y_2) = (2y_1 + 2y_2, y_1 + 3y_2)$

♠ 14. $T : \mathbb{R}^2 \to \mathbb{R}^3 : T(x_1, x_2) = (2x_2, 3x_1, 2x_1 - 3x_2)$ and $S : \mathbb{R}^3 \to \mathbb{R}^2 : S(y_1, y_2, y_3) = (y_3 - 2y_2, y_1 - 2y_3)$

15. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/transform.ipynb

"Life is a mirror and will reflect back to the thinker what he thinks into it." – Ernest Holmes

3.5 Linear Transformations on $\mathbb{R}^2$

In this section, we will explore how multiplication by a nonsingular matrix transforms geometry in $\mathbb{R}^2$.

3.5.1 Stretches and Reflections

Let $a, b \in \mathbb{R}$ such that $a, b > 0$.

We can **stretch** $\mathbb{R}^2$ **horizontally** by multiplying the (scaling) matrix

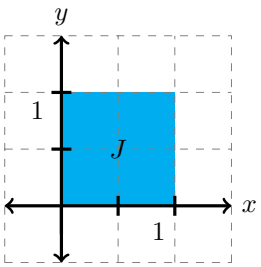
$$\begin{bmatrix} a & 0 \\ 0 & 1 \end{bmatrix}.$$

We can **stretch** $\mathbb{R}^2$ **vertically** by multiplying the matrix

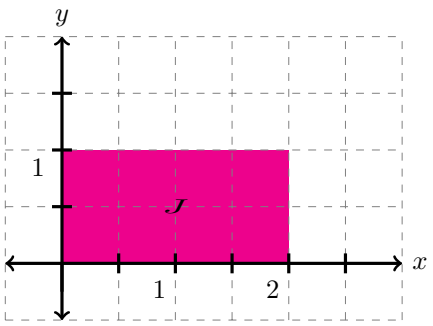
$$\begin{bmatrix} 1 & 0 \\ 0 & b \end{bmatrix}.$$

Thus, multiplication by a diagonal matrix will rescale the x- and y-axes by the diagonal entries. When $a < 1$ or $b < 1$, we say the geometry is **compressed** since the scale becomes smaller than the original.

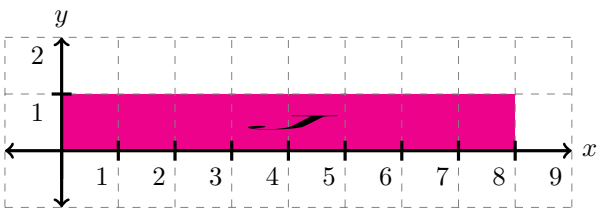
Example 3.5.1. The **unit square** J is the rectangle with vertices at $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$. The unit square is displayed below in cyan. It will be a useful tool to visualize how matrix multiplication distorts the geometry by seeing how a matrix transforms the unit square J.



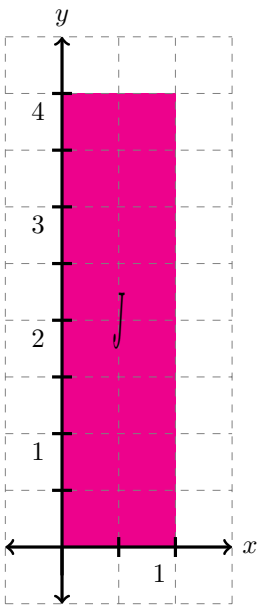
(a) Multiplication by the matrix $\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ horizontally stretches the plane by a factor of two.



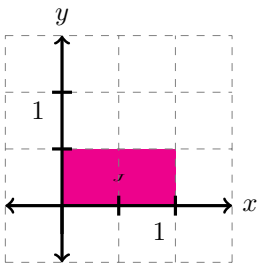
(b) Multiplication by the matrix $\begin{bmatrix} 8 & 0 \\ 0 & 1 \end{bmatrix}$ horizontally stretches the plane by a factor of eight.



(c) Multiplication by the matrix $\begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$ vertically stretches the plane by a factor of four.



(d) Multiplication by the matrix $\begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$ vertically comprresses the plane by a factor of two.



■

Example 3.5.2. The matrix $\begin{bmatrix} 1/2 & 0 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}$ will vertically stretch the four vertices of J by a factor of 3 and will horizontally compress the vertices by a factor of 2 (or we can say that they are horizontally stretched by a factor of 1/2). In particular,

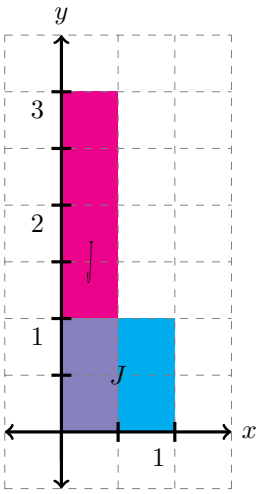
$$\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ 3 \end{bmatrix}$$

The image of J is displayed to the right in magenta.

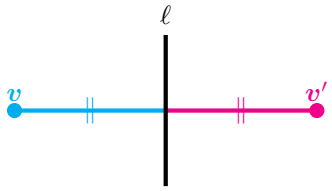


■

Reflections across the x- and y-axis are accomplished by multiplying by the scaling (diagonal) matrices

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix},$$

respectively.



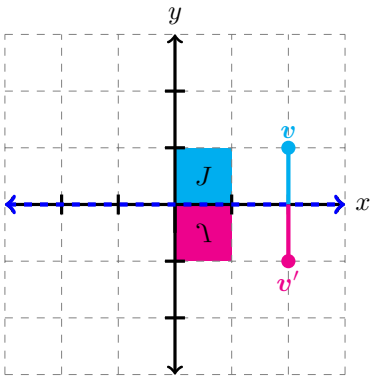
The interchange (permutation) matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ produces a reflection across the line $y = x$. The composite of reflections across the x- and y-axis gives a reflection through the origin. General reflections will be discussed in the homework.

Example 3.5.3. Consider the reflections of point $\boldsymbol{v} = (2, 1)$ across:

(a) the x-axis.

The image of $\boldsymbol{v}$ under this reflection is

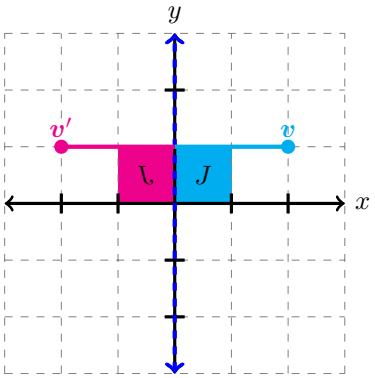
$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}.$$



(b) the y-axis.

The image of $\boldsymbol{v}$ under this reflection is

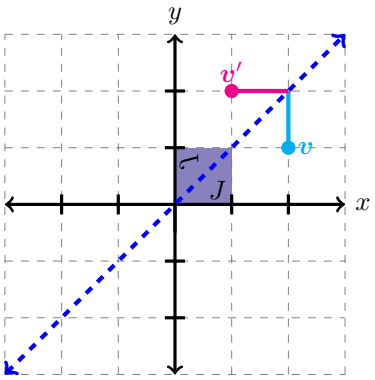
$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \end{bmatrix}.$$



(c) $y = x$.

The image of $\boldsymbol{v}$ under this reflection is

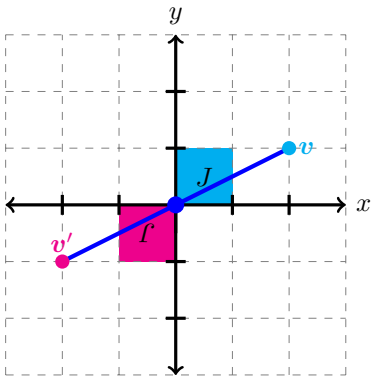
$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$



(d) the origin.

The image of $\boldsymbol{v}$ under this reflection is

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix}.$$

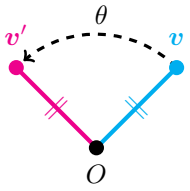


■

3.5.2 Rotations

A counterclockwise rotation in $\mathbb{R}^2$ by angle θ is captured by the matrix

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$



For a clockwise rotation by θ, take the previous matrix's inverse, $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}^{-1} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$.

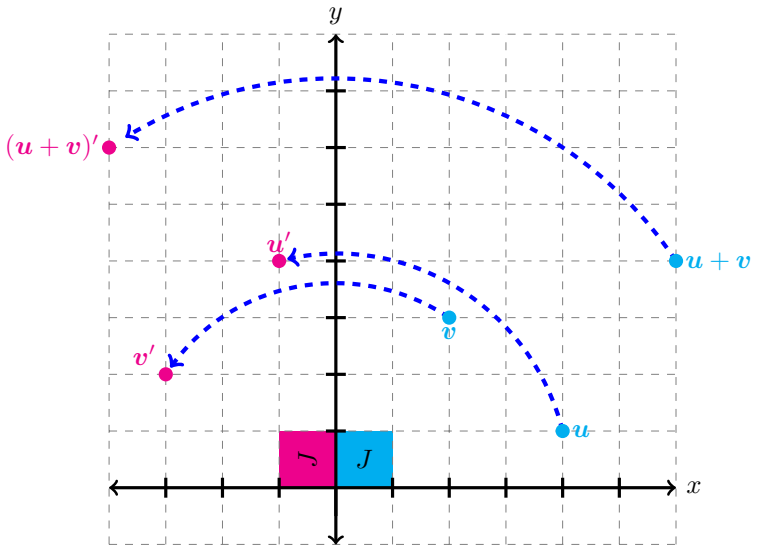
Example 3.5.4. Let $T : \mathbb{R}^2 \to \mathbb{R}^2$ be the linear transformation given as counterclockwise rotation by $\frac{\pi}{2}$ (or 90°). Find the images under T of $\boldsymbol{u} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$, $\boldsymbol{v} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$, and $\boldsymbol{u}+\boldsymbol{v} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}$.

Note that $[T] = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$. Then

$$T(\boldsymbol{u}) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 4 \end{bmatrix}$$

$$T(\boldsymbol{v}) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$$

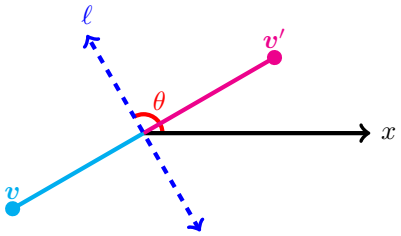
$$T(\boldsymbol{u}+\boldsymbol{v}) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 6 \\ 4 \end{bmatrix} = \begin{bmatrix} -4 \\ 6 \end{bmatrix}$$



■

Utilizing rotations, we can see that reflections across the line ℓ that passes through the origin forming the angle θ is accomplished by multiplying by matrix

$$\begin{bmatrix} \cos(2\theta) & \sin(2\theta) \\ \sin(2\theta) & -\cos(2\theta) \end{bmatrix}.$$



One last linear transformation family on $\mathbb{R}^2$, called *projections*, will be addressed in Section C.2.

A similar analysis may be used to describe the geometry of matrix transformation in $\mathbb{R}^3$ (and higher).

3.5.3 Affine Transformations

Let $\boldsymbol{b} \in \mathbb{R}^n$ be any vector in the vector space. We can define a transformation $T : \mathbb{R}^n \to \mathbb{R}^n$ by the rule $T(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{b}$, called a **translation** by $\boldsymbol{b}$. If $\boldsymbol{b} \neq \boldsymbol{0}$, then translation is not a linear map since $T(\boldsymbol{0}) = \boldsymbol{b} \neq 0$. On the other hand, $\text{dist}(T(\boldsymbol{x}), T(\boldsymbol{y})) = \|(\boldsymbol{x}+\boldsymbol{b})-(\boldsymbol{y}+\boldsymbol{b})\| = \|\boldsymbol{x}+\boldsymbol{b}-\boldsymbol{y}-\boldsymbol{b}\| = \|\boldsymbol{x}-\boldsymbol{y}\| = \text{dist}(\boldsymbol{x}, \boldsymbol{y})$. If we allow for translations in a vector space, we must broaden our type of transformations to affine transformations.

Definition 3.5.5. Let $T : \mathbb{R}^n \to \mathbb{R}^m$ be a function. We say that T is an **affine transformation** if there exists a matrix $A \in \mathbb{R}^{m\times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^m$ such that $T(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$ for all $\boldsymbol{x} \in \mathbb{R}^n$.

Of course, a linear transformation is affine (when $\boldsymbol{b} = \boldsymbol{0}$) and every translation is affine (when $A = I_n$). Affine transformations are very important in linear geometry. As translation maps are ALWAYS one-to-one (onto), an affine transformation $\boldsymbol{x} \mapsto A\boldsymbol{x} + \boldsymbol{b}$ is one-to-one (onto) if and only if the matrix transformation $\boldsymbol{x} \mapsto A\boldsymbol{x}$ is one-to-one (onto).

Example 3.5.6. Consider the affine transformation $T : \mathbb{R}^3 \to \mathbb{R}^3$ associated to the matrix $A = \begin{bmatrix} 3 & -1 & 1 \\ -3 & 2 & 0 \\ 6 & -3 & 2 \end{bmatrix}$ and the translation vector $\boldsymbol{b} = (1, 2, 3)$. Then

$$T\left(\begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix}\right) = \begin{bmatrix} 3 & -1 & 1 \\ -3 & 2 & 0 \\ 6 & -3 & 2 \end{bmatrix}\begin{bmatrix} 2 \\ 0 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ -6 \\ 10 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ -4 \\ 13 \end{bmatrix}.$$

Is $\boldsymbol{y} = (2, -2, 4) \in \text{im}(T)$? To answer this question, we solve the matrix equation $A\boldsymbol{x} + \boldsymbol{b} = \boldsymbol{y} \Rightarrow A\boldsymbol{x} = \boldsymbol{y} - \boldsymbol{b} = (1, -4, 1)$:

$$\left[\begin{array}{ccc|c} 3 & -1 & 1 & 1 \\ -3 & 2 & 0 & -4 \\ 6 & -3 & 2 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -4 \end{array}\right].$$

Therefore, $T(2, 1, -4) = (2, -2, 4)$. ∎

Over $\mathbb{R}^2$ there are four types of isometries: rotation around a point in the plane, reflection across a line in the plane, translation, and glide reflection (the composition of a reflection and a translation, like footprints in the sand).

Python Programming

Example 3.5.7. The following code:

```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-whitegrid') #Use 'seaborn-whitegrid' if error occurs.

dilate = lambda a, b: np.array([[a, 0], [0, b]])

rotate = lambda theta: np.array([[np.cos(theta), -np.sin(theta)],
[np.sin(theta), np.cos(theta)]])

reflect = lambda theta: np.array([[np.cos(2*theta), np.sin(2*theta)],
[np.sin(2*theta), -np.cos(2*theta)]])

project = lambda theta: 1/2*np.array([[1+np.cos(2*theta), np.sin(2*theta)],
[np.sin(2*theta), 1-np.cos(2*theta)]])

#create a scatter plot for the unit square
points = np.concatenate([[[i*0.01,j*0.01] for i in range(101)] for j in range(101)])

transform = "dilate"
n=3
n_it = n            #The number of iterations you want to rotate; set to 2 for single rotate
theta= 2*np.pi/n    #Controls angle of rotation, reflection, and projection
a, b = 2, 0.5       #Controls horizontal and vertical scalars for dilation
scal = 2            #Controls the scale of the gridlines

plt.figure(figsize=(8, 8), dpi=80)
plt.ion()
plt.axis([-1*scal,scal,-1*scal,scal])

if transform == "dilate":
    T = dilate(a,b)
    #Now rotate them
    tpoints = np.array([T @ p for p in points])

    #Show the two sets of points
    plt.scatter([c[0] for c in points], [c[1] for c in points])
    plt.scatter([c[0] for c in tpoints],[c[1] for c in tpoints], alpha=0.1)

if transform == "rotate":
    T = [rotate(k*theta) for k in range(n_it)]

    #Now rotate them
    tpoints = [np.array([T[k] @ p for p in points]) for k in range(n_it)]

    #Show the two sets of points

    #plt.scatter([c[0] for c in points], [c[1] for c in points])
    for k in range(n_it):
        plt.scatter([c[0] for c in tpoints[k]],[c[1] for c in tpoints[k]], alpha=0.1)

if transform == "reflect":
    T = reflect(theta)

    #Now rotate them
    tpoints = np.array([T @ p for p in points])

    #Show the two sets of points
    plt.scatter([c[0] for c in points], [c[1] for c in points])
    plt.scatter([c[0] for c in tpoints],[c[1] for c in tpoints], alpha=0.1)

    t = np.linspace(-2, 2, 100)
    plt.plot(t,t*np.tan(theta), "-.r")

if transform == "project":
    T = project(theta)

    #Now rotate them
    tpoints = np.array([T @ p for p in points])
```

```
69
70      #Show the two sets of points
71      plt.scatter([c[0] for c in points], [c[1] for c in points])
72      plt.scatter([c[0] for c in tpoints],[c[1] for c in tpoints], alpha=0.1)
73
74      t = np.linspace(-2, 2, 100)
75      plt.plot(t,t*np.tan(theta), "-.r")
76
77  plt.show()
```

can be used to implement the transformations of dilation, reflection, rotation, and projection. ■

Exercises

(Go to Solutions)

For Exercises 1-9, find the matrix for the operator that performs the stated geometric transformation(s).

1. Rotates through an angle of $\frac{\pi}{2}$ radians about the origin, then reflects across the x-axis, and then expands by a factor of 4 in the x-direction.

♠ 2. Reflect across the y-axis, stretch by a factor of 2 in the x-direction, then expand by a factor of 3 in the y-direction.

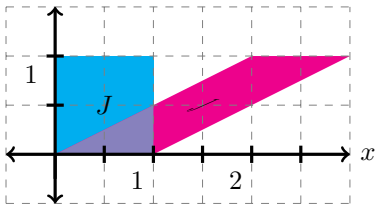
♠ 3. Reflects about the line $y = x$, then rotates through an angle of $\frac{\pi}{4}$ radians about the origin, and then contracts by a factor of 2 in the y-direction.

♠ 4. Reflects across the x-axis, then reflects across the line $y = x$, and then expands by a factor of 2 in the y-direction.

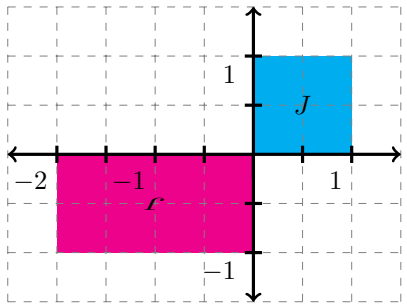
5. Rotates through an angle of $\frac{\pi}{3}$ radians clockwise about the origin, then expands by a factor of 2 in the x-direction, and then reflects across the y-axis.

6. Expands by a factor of 3 in both the x- and y-directions, then reflects across the y-axis, and then rotates through an angle of π radians about the origin.

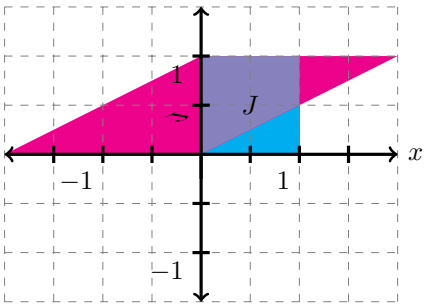
7.



8.



9.



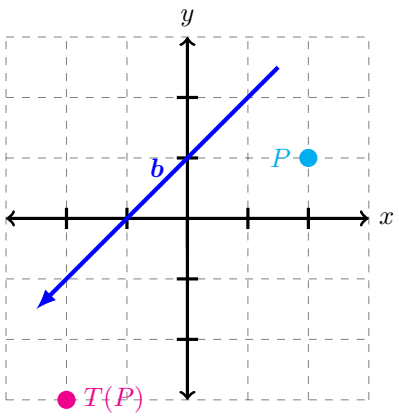
For Exercises 10-14, let $T : \mathbb{R}^n \to \mathbb{R}^n : \boldsymbol{x} \mapsto \boldsymbol{x} + \boldsymbol{b}$ be translation by $\boldsymbol{b}$, listed first. Compute the image $T(P)$, where P is the second listed vector. When 2-dimensional, draw P and $T(P)$ on the same gridlines.

♠ 10. $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ ♠ 11. $\begin{bmatrix} 0 \\ -2 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ ♠ 12. $\begin{bmatrix} -1 \\ -1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ ♠ 13. $\begin{bmatrix} -2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}$

14. $\begin{bmatrix} 0 \\ -7 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

For Exercises 15-15, let $T : \mathbb{R}^2 \to \mathbb{R}^2 : \boldsymbol{x} \mapsto \boldsymbol{x} + \boldsymbol{b}$ be translation by $\boldsymbol{b}$, illustrated below. The vector $P = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ and its image $T(P)$ are likewise illustrated. Find the translation vector $\boldsymbol{b}$.

15.



For Exercises 16-17, consider the affine transformation $T : \mathbb{R}^3 \to \mathbb{R}^3 : \boldsymbol{x} \mapsto A\boldsymbol{x} + \boldsymbol{b}$ associated to the matrix

$$A = \begin{bmatrix} 3 & -1 & 1 \\ -3 & 2 & 0 \\ 6 & -3 & 2 \end{bmatrix} \text{ and the vector } \boldsymbol{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

16. Compute $T(3, 1, -5)$.

17. Find a vector $\boldsymbol{x} \in \mathbb{R}^3$ such that $T(\boldsymbol{x}) = (6, 1, 3)$.

For Exercises 18-19, consider the affine transformation $T : \mathbb{R}^3 \to \mathbb{R}^3 : \boldsymbol{x} \mapsto A\boldsymbol{x} + \boldsymbol{b}$ associated to the matrix

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 2 & 2 \end{bmatrix} \text{ and the vector } \boldsymbol{b} = \begin{bmatrix} 1 \\ -5 \\ 2 \end{bmatrix}.$$

18. Compute $T(4, 6, 13)$.

19. Find a vector $\boldsymbol{x} \in \mathbb{R}^3$ such that $T(\boldsymbol{x}) = (4, 6, 13)$.

For Exercises 20-21, consider the affine transformation $T : \mathbb{R}^3 \to \mathbb{R}^3 : \boldsymbol{x} \mapsto A\boldsymbol{x} + \boldsymbol{b}$ associated to the matrix

$$A = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 2 & 2 \\ 4 & 1 & -3 \end{bmatrix} \text{ and the vector } \boldsymbol{b} = \begin{bmatrix} -5 \\ 6 \\ 8 \end{bmatrix}.$$

♠ 20. Compute $T(5, -2, 4)$.

♠ 21. Find a vector $\boldsymbol{x} \in \mathbb{R}^3$ such that $T(\boldsymbol{x}) = (-6, 8, 12)$.

For Exercises 22-23, consider the affine transformation $T : \mathbb{R}^3 \to \mathbb{R}^3 : \boldsymbol{x} \mapsto A\boldsymbol{x} + \boldsymbol{b}$ associated to the matrix

$$A = \begin{bmatrix} 6 & 4 & 6 \\ 2 & 2 & 2 \\ 10 & 8 & 6 \end{bmatrix} \text{ and the vector } \boldsymbol{b} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}.$$

22. Compute $T(1, 2, 1)$.

23. Find a vector $\boldsymbol{x} \in \mathbb{R}^3$ such that $T(\boldsymbol{x}) = (7, 6, 5)$.

24. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/geotransform.ipynb

"Anyone who imagines that bliss is normal is going to waste a lot of time running around shouting that he has been robbed." – Jenkins Lloyd Jones

3.6 The Gram-Schmidt Algorithm

Using orthogonal projections, we can construct an orthogonal basis from a given basis. If desired, we can also construct an orthonormal basis by normalizing.

3.6.1 The Gram-Schmidt Algorithm

Theorem 3.6.1 (The Gram-Schmidt Process)**.** *Let $\mathcal{B} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p\}$ be a basis for a subspace $W \leq \mathbb{R}^n$. Define recursively the vectors*

$$\begin{aligned}
\boldsymbol{v}_1 &= \boldsymbol{x}_1 \\
\boldsymbol{v}_2 &= \boldsymbol{x}_2 - \frac{\boldsymbol{v}_1 \cdot \boldsymbol{x}_2}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1}\boldsymbol{v}_1 \\
\boldsymbol{v}_3 &= \boldsymbol{x}_3 - \frac{\boldsymbol{v}_1 \cdot \boldsymbol{x}_3}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1}\boldsymbol{v}_1 - \frac{\boldsymbol{v}_2 \cdot \boldsymbol{x}_3}{\boldsymbol{v}_2 \cdot \boldsymbol{v}_2}\boldsymbol{v}_2 \\
&\vdots \\
\boldsymbol{v}_p &= \boldsymbol{x}_p - \frac{\boldsymbol{v}_1 \cdot \boldsymbol{x}_p}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1}\boldsymbol{v}_1 - \ldots - \frac{\boldsymbol{v}_{p-1} \cdot \boldsymbol{x}_p}{\boldsymbol{v}_{p-1} \cdot \boldsymbol{v}_{p-1}}\boldsymbol{v}_{p-1}
\end{aligned}$$

Then $\mathcal{C} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p\}$ is an orthogonal basis of W. In addition,

$$\text{Span}\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\} = \text{Span}\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\} \qquad \forall k.$$

Example 3.6.2. Let $\boldsymbol{x}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$, $\boldsymbol{x}_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$, and $\boldsymbol{x}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$, which as a set is linearly independent. Let $W = \text{Span}\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3\}$. Then using the Gram-Schmidt procedure, we can construct an orthogonal basis of W.

$$\boldsymbol{v}_1 = \boldsymbol{x}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\boldsymbol{v}_2 = \boldsymbol{x}_2 - \frac{\boldsymbol{v}_1 \cdot \boldsymbol{x}_2}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1}\boldsymbol{v}_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} - \frac{3}{4}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -3/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

$$\boldsymbol{v}_3 = \boldsymbol{x}_3 - \frac{\boldsymbol{v}_1\cdot\boldsymbol{x}_3}{\boldsymbol{v}_1\cdot\boldsymbol{v}_1}\boldsymbol{v}_1 - \frac{\boldsymbol{v}_2\cdot\boldsymbol{x}_3}{\boldsymbol{v}_2\cdot\boldsymbol{v}_2}\boldsymbol{v}_2 = \begin{bmatrix}0\\0\\1\\1\end{bmatrix} - \frac{2}{4}\begin{bmatrix}1\\1\\1\\1\end{bmatrix} - \frac{1/2}{12/16}\begin{bmatrix}-3/4\\1/4\\1/4\\1/4\end{bmatrix}$$

$$= \begin{bmatrix}0\\0\\1\\1\end{bmatrix} - \begin{bmatrix}1/2\\1/2\\1/2\\1/2\end{bmatrix} - \begin{bmatrix}-3/6\\1/6\\1/6\\1/6\end{bmatrix} = \begin{bmatrix}0\\-2/3\\1/3\\1/3\end{bmatrix}$$

Note that

$$\boldsymbol{v}_1\cdot\boldsymbol{v}_2 = -3/4+1/4+1/4+1/4 = 0, \quad \boldsymbol{v}_1\cdot\boldsymbol{v}_3 = 0-2/3+1/3+1/3 = 0, \quad \boldsymbol{v}_2\cdot\boldsymbol{v}_3 = 0-2/3+1/3+1/3 = 0.$$

Therefore, $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$ is an orthogonal basis for W. If we normalize:

$$\boldsymbol{u}_1 = \frac{1}{\sqrt{4}}\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = \begin{bmatrix}1/2\\1/2\\1/2\\1/2\end{bmatrix}, \quad \boldsymbol{u}_2 = \sqrt{\frac{4}{3}}\begin{bmatrix}-3/4\\1/4\\1/4\\1/4\end{bmatrix} = \begin{bmatrix}-3/\sqrt{12}\\1/\sqrt{12}\\1/\sqrt{12}\\1/\sqrt{12}\end{bmatrix}, \quad \boldsymbol{u}_3 = \sqrt{\frac{9}{6}}\begin{bmatrix}0\\-2/3\\1/3\\1/3\end{bmatrix} = \begin{bmatrix}0\\-2/\sqrt{6}\\1/\sqrt{6}\\1/\sqrt{6}\end{bmatrix},$$

then $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$ is an orthonormal basis of W. ∎

3.6.2 Isometries

Definition 3.6.3. A real square matrix U is called **orthogonal** if $U^\top U = I$, that is, $U^T = U^{-1}$.

Theorem 3.6.4. *A square matrix U is orthogonal if and only if its column vectors form an orthonormal set.*

Since $U^\top = U^{-1}$ and $UU^\top = UU^{-1} = I$, it also follows that the row vectors of an orthogonal matrix U must also form an orthonormal set.

Rotation and reflection matrices are examples of orthogonal matrices.

Example 3.6.5. The matrix

$$U = \begin{bmatrix}3/\sqrt{11} & -1/\sqrt{6} & -1/\sqrt{66}\\ 1/\sqrt{11} & 2/\sqrt{6} & -4/\sqrt{66}\\ 1/\sqrt{11} & 1/\sqrt{6} & 7/\sqrt{66}\end{bmatrix}$$

is an orthogonal matrix. Note that

$$U^\top U = \begin{bmatrix}3/\sqrt{11} & 1/\sqrt{11} & 1/\sqrt{11}\\ -1/\sqrt{6} & 2/\sqrt{6} & 1/\sqrt{6}\\ -1/\sqrt{66} & -4/\sqrt{66} & 7/\sqrt{66}\end{bmatrix}\begin{bmatrix}3/\sqrt{11} & -1/\sqrt{6} & -1/\sqrt{66}\\ 1/\sqrt{11} & 2/\sqrt{6} & -4/\sqrt{66}\\ 1/\sqrt{11} & 1/\sqrt{6} & 7/\sqrt{66}\end{bmatrix}$$

$$= \begin{bmatrix} (9+1+1)/11 & (-3+2+1)/\sqrt{66} & (-3-4+7)/\sqrt{726} \\ (-3+2+1)/\sqrt{66} & (1+4+1)/6 & (1-8+7)/\sqrt{396} \\ (-3-4+7)/\sqrt{726} & (1-8+7)/\sqrt{396} & (1+16+49)/66 \end{bmatrix} = I_3 \qquad \blacksquare$$

Theorem 3.6.6. *Let U be an orthogonal matrix, and let $\boldsymbol{x}, \boldsymbol{y} \in F^n$. Then*

$$(U\boldsymbol{x}) \cdot (U\boldsymbol{y}) = \boldsymbol{x} \cdot \boldsymbol{y},$$

that is, the matrix transformation $\boldsymbol{x} \mapsto \boldsymbol{U}\boldsymbol{x}$ preserves inner products.

Proof.

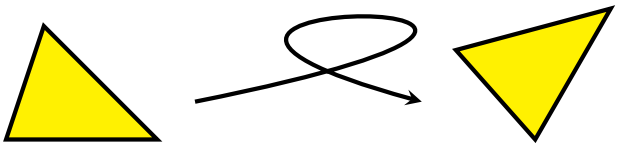
$$(U\boldsymbol{x}) \cdot (U\boldsymbol{y}) = (U\boldsymbol{x})^\top (U\boldsymbol{y}) = (\boldsymbol{x}^\top U^\top)(U\boldsymbol{y}) = \boldsymbol{x}^\top (U^\top U)\boldsymbol{y} = \boldsymbol{x}^\top \boldsymbol{y} = \boldsymbol{x} \cdot \boldsymbol{y}. \qquad \square$$

Since multiplication by an orthogonal matrix U preserves inner products, as a consequence, it also preserves lengths, distances, angles, and orthogonality of vectors. For example, $\|U\boldsymbol{x}\| = \|\boldsymbol{x}\|$ and $\operatorname{dist}(U\boldsymbol{x}, U\boldsymbol{y}) = \operatorname{dist}(\boldsymbol{x}, \boldsymbol{y})$ for any vectors $\boldsymbol{x}, \boldsymbol{y}$ and any orthogonal matrix U.

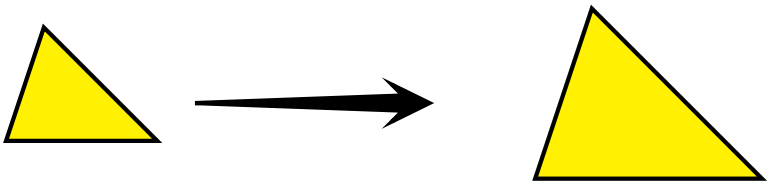
Definition 3.6.7. An **isometry** (or a **rigid motion**) is a function $R : F^n \to F^n$ such that $\operatorname{dist}(T(\boldsymbol{x}), T(\boldsymbol{y})) = \operatorname{dist}(\boldsymbol{x}, \boldsymbol{y})$ for all $\boldsymbol{x}, \boldsymbol{y} \in F^n$.

Isometry, which translates as "same-measure" from Greek, are those maps that preserve distances and lengths. Using trigonometry, necessarily angles are preserved too. Therefore, an isometry is exactly a map which "moves" objects is the vector space without "distorting" them, that is, the image of a shape will be congruent to the original shape. Linear isometries are exactly multiplication by an orthogonal matrix.

Example 3.6.8. The following graphic illustrates an example of an isometry. Note that the triangle on the left is the original shape and the triangle on the right is its transformation. In this example, the original triangle is rotated and translated, but the original shape is not changed. Note that the distances between the three vertices is the same in the before and after images. Likewise, the angles are congruent between the three angles in the before and after images. This illustrates an isometry.



Conversely, the second graphic illustrates a transformation which is not isometric. Again, the triangle on the left is the original shape and the triangle on the right is its transformation. Note that the distance between the three vertices is greater in the transformation (50% to be precise). Thus, the distance between these three points increased, which makes the transformation not an isometry.



On the other hand, the measure of the three angles is congruent to the original. This shows that a distance-breaking transformation can still preserve angles. On the other hand, any distance preserving transformation must also preserve angle meaures. $\blacksquare$

3.6.3 The QR Factorization

Theorem 3.6.9 (The QR Factorization)**.** *If A is an $m \times n$ matrix with linearly independent columns, then A can be factored as $A = QR$, where Q is an $m \times n$ matrix with orthonormal columns and $\operatorname{Col} A = \operatorname{Col} Q$ and R is an $n \times n$ upper triangular matrix with positive diagonal entries. In particular, R is necessarily invertible.*

The matrix Q is constructed by using an orthonormal basis provided by the Gram-Schmidt process, although some columns might need to be scaled by -1 to guarantee that the diagonal entries of R are positive. If $A = QR$, then it can be shown that $R = \begin{bmatrix} \langle \boldsymbol{q}_i, \ \boldsymbol{a}_j \rangle \end{bmatrix}$, where $\boldsymbol{q}_i$ and $\boldsymbol{a}_j$ denote the columns of Q and A, respectfully. With respect to the dot product, this gives that $Q^\top A = R$.

Example 3.6.10. Let $A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$. By Example 3.6.2, $\{\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3\}$ is an orthonormal basis for $\operatorname{Col} A$ where

$$Q = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \boldsymbol{u}_3 \end{bmatrix} = \begin{bmatrix} 1/2 & -3/\sqrt{12} & 0 \\ 1/2 & 1/\sqrt{12} & -2/\sqrt{6} \\ 1/2 & 1/\sqrt{12} & 1/\sqrt{6} \\ 1/2 & 1/\sqrt{12} & 1/\sqrt{6} \end{bmatrix}.$$

Let

$$R = Q^\top A = \begin{bmatrix} 1/2 & 1/2 & 1/2 & 1/2 \\ -3/\sqrt{12} & 1/\sqrt{12} & 1/\sqrt{12} & 1/\sqrt{12} \\ 0 & -2/\sqrt{6} & 1/\sqrt{6} & 1/\sqrt{6} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 3/2 & 1 \\ 0 & 3/\sqrt{12} & 1/\sqrt{3} \\ 0 & 0 & 2/\sqrt{6} \end{bmatrix}.$$

Therefore, $A = QR$. ∎

Suppose that A is nonsingular. Then it has a QR factorization, say $A = QR$. Then

$$A^{-1} = (QR)^{-1} = R^{-1}Q^{-1} = R^{-1}Q^\top.$$

Thus, finding the inverse of A comes to finding the inverse of a triangular matrix. While this is better, we are not quite there yet.

Let us return to the solving of linear systems, namely solve $A\boldsymbol{x} = \boldsymbol{b}$. Again, using the QR factorization, we have

$$\begin{aligned} A\boldsymbol{x} &= \boldsymbol{b} \\ (QR)\boldsymbol{x} &= \boldsymbol{b} \\ Q^\top(QR)\boldsymbol{x} &= Q^\top\boldsymbol{b} \\ (Q^\top Q)R\boldsymbol{x} &= Q^\top\boldsymbol{b} \\ R\boldsymbol{x} &= Q^\top\boldsymbol{b} \end{aligned}$$

So, since we can find the inverse of Q so easily, we need only to solve a linear system associated to an upper triangular matrix, that is, a system in echelon form which we know is very simple by back-substitution.

Combining these two algorithms, the QR factorization and back-substitution, we have now an algorithm which is computationally faster than the Gauss-Jordan elimination (compute the RREF) which we saw earlier. While challenging to do by hand, the QR factorization is utilized in many numerical linear algebra applications.

Note that

$$A^\dagger A = [(A^\top A)^{-1}A^\top]A = (A^\top A)^{-1}(A^\top A) = I.$$

Hence, $A^\dagger$ is a left inverse of A in this case. In the case that A is a square matrix (and hence nonsingular), $A^\dagger = A^{-1}$. If $A = QR$ is a QR factorization of A, then

$$A^\dagger = (A^\top A)^{-1}A^\top = ((QR)^\top QR)^{-1}(QR)^\top = (R^\top Q^\top QR)^{-1}R^\top Q^\top = (R\top R)^{-1}R^\top Q^\top = R^{-1}(R^\top)^{-1}R^\top Q^\top = R^{-1}Q^\top.$$

This last identity holds even when A is not a square matrix.

Python Programming

The `NumPy` command `np.linalg.qr(A)` computes the QR factorization of the $m \times n$ matrix A. It returns two matrices: Q and R. But, for the sake of computational simplicity, the diagaonals of R are not necessarily positive. To guarantee positive diagaonl entries,[i], we can manually inspect the diagonal entries of R (`np.diag(R)`) using command `np.sign(<array>)` which returns an array, of the same length, of ± 1 depending on whether the corresponding entries are positive or negative.

Example 3.6.11. The following code:

```
import numpy as np

Q,R = np.linalg.qr([[0,-1],[1,3]])
print(Q)
print(R,'\n')


sign = np.sign(np.diag(R)) #Extract the signs of the diagonal entries of R
Q, R = Q*sign, R*sign      #Change the negative signs in Q,R
print(Q)
print(R,'\n')
```

outputs:

```
[[ 0.  -1.]
[-1.  0.]]
[[-1.  -3.]
[ 0.  1.]]

[[-0.  -1.]
[ 1.  0.]]
[[ 1.  -3.]
[-0.  1.]]
```

■

[i] If the matrix A is rank-deficient, you may want to include the line `ph[ph == 0] = 1` to handle any zeros on the diagonal.

Exercises

(Go to Solutions)

For Exercises 1-5, compute the QR-factorizations of the matrix.

♠ 1. $\begin{bmatrix} 0 & -1 \\ 1 & 3 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 1 & -1 \\ 1 & 3 \end{bmatrix}$

♠ 3. $\begin{bmatrix} 4 & 2 \\ -4 & 4 \\ 2 & -5 \end{bmatrix}$

♠ 4. $\begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 3 & 4 \\ 1 & 2 \end{bmatrix}$

♠ 5. $\begin{bmatrix} 1 & 1 & 1 \\ -2 & 0 & 1 \\ 2 & 1 & 2 \end{bmatrix}$

6. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/gram.ipynb

"We can have exciting experiences as we learn about our vibrant, dynamic ancestors. They were very real, living people with problems, hopes, and dreams like we have today." – James E. Faust

3.7 Linear Dynamical Systems

We are often interested in predicting the future, and our only tool to do so is to look at the present (or maybe also the past). Humans are really good as a species at recognizing patterns. We understand that when a pattern is established its projection into the future can be an accurate prediction model. In mathematical modeling, this pattern recognition and future projection is known as dynamical systems.

3.7.1 Dynamical Systems

Definition 3.7.1. A **dynamical system** (or **Markov chain**) is a sequence of n-vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \ldots$ and a sequence of functions $f_t : \mathbb{R}^n \to \mathbb{R}^n$ for $t = 1, 2, 3, \ldots$ such that

$$\boldsymbol{x}_{t+1} = f_t(\boldsymbol{x}_t).$$

Each vector in the sequence is called a **state**. If $\boldsymbol{x}_t$ is the *current state*, then $\boldsymbol{x}_{t-1}$ would be the *previous state* and $\boldsymbol{x}_{t+1}$ would be the *next state*. The sequence of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \ldots$ is called the **trajectory**. The functions $f_t : \mathbb{R}^n \to \mathbb{R}^n$ are called the **transition functions** for the dynamical system.

If we have a dynamical system and know the current state $\boldsymbol{x}_t$, as well as all the future transition functions $f_t, f_{t+1}, f_{t+2}, \ldots$, then we can compute the future state $\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+2}, \boldsymbol{x}_{t+3}, \ldots$ by iterating the above equation, namely

$$\boldsymbol{x}_{t+1} = f_t(\boldsymbol{x}_t), \quad \boldsymbol{x}_{t+2} = f_{t+1}(\boldsymbol{x}_{t+1}), \quad \boldsymbol{x}_{t+3} = f_{t+2}(\boldsymbol{x}_{t+2}), \ldots$$

Thus, we can compute the future trajectory using the current state and the transition future. This is called a **simulation**. Of course, the accuracy of this simulation comes from the measurement of the current state $\boldsymbol{x}_t$ (this can generally be found reliable so long as qualitative data collection methods were used) and the future transition functions f_t, f_{t+1}, f_{t+2} (generally, if a prediction is bad is comes from poor predictions on the future transitions, as trends can change over time it can be very difficult to predict these functions far into the future).

Example 3.7.2. As some examples of a dynamical system, let $\boldsymbol{x}_t$ represent the current prices of several stocks listed in a portfolio, which change regularly through any given trading day. The dynamical system could measure the quarterly activities of companies, organizations, or sectors in an economy. Or $\boldsymbol{x}_t$ could denote the positions (or velocities, accelerations, etc.) of objects in some mechanical system. The system could measure the distribution of certain demographics in a population that change over time. The concept of a Markov chain is extremely broad and hence has wide applications. ∎

3.7.2 Linear Dynamical Systems

Definition 3.7.3. A **linear dynamical system** is a dynamical system where each of the transition functions $f_t : \mathbb{R}^n \to \mathbb{R}^n$ is itself a linear transformation. As such, the transition from one state to another can be completed by matrix multiplication. If A_t is the $n \times n$ matrix representation of f_t, then

$$\boldsymbol{x}_{t+1} = A_t \boldsymbol{x}_t,$$

which is called the **dynamic equation**. The matrices A_t are called the **transition matrices**. A linear dynamical system is called **time-invariant** if $A_t = A$ for all t, that is, the current transition matrix does not depend on the current time t.

Often in practice, a dynamical system does vary its transition matrices over time, but because it is difficult (or impossible) to accurately predict the transitions far in the future it is common to simulate a dynamical system as if it were time-invariant for a short time until new measurements can be made to update the transition matrix.

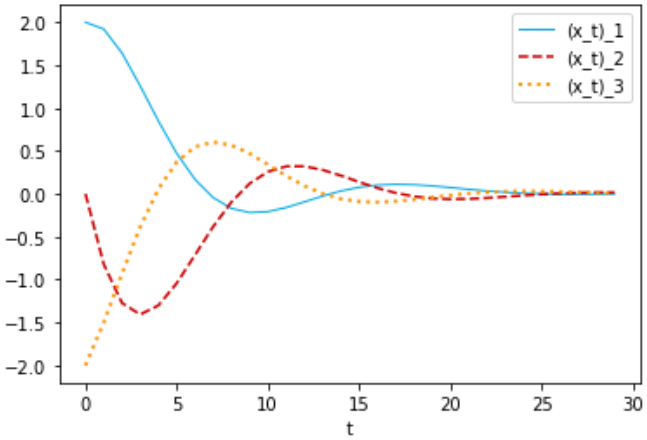
Example 3.7.4. Let $A = \begin{bmatrix} 0.87 & 0.2 & -0.09 \\ -0.1 & 0.75 & 0.31 \\ 0.06 & -0.21 & 0.81 \end{bmatrix}$ be the transition matrix for a time-invariant dynamical system. Suppose that $\boldsymbol{x}_1 = (2, 0, -2)^\top$. Find $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$.

Analyzing the transition matrix A, we could possible interpret it as possible percentage increases and decreases based upon the current state. For example, the first entry in the next state is 87% of the first entry, 20% of the second entry, and -9% of the last entry from the current state.

$$\boldsymbol{x}_2 = A\boldsymbol{x}_1 = \begin{bmatrix} 0.87 & 0.2 & -0.09 \\ -0.1 & 0.75 & 0.31 \\ 0.06 & -0.21 & 0.81 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ -2 \end{bmatrix} = \begin{bmatrix} 1.74 - 0.18 \\ -0.2 - 0.62 \\ 0.12 - 1.62 \end{bmatrix} = \begin{bmatrix} 1.92 \\ -0.82 \\ -1.5 \end{bmatrix},$$

$$\boldsymbol{x}_3 = A\boldsymbol{x}_2 = \begin{bmatrix} 0.87 & 0.2 & -0.09 \\ -0.1 & 0.75 & 0.31 \\ 0.06 & -0.21 & 0.81 \end{bmatrix} \begin{bmatrix} 1.92 \\ -0.82 \\ -1.5 \end{bmatrix} = \begin{bmatrix} 1.674 - 0.164 + 0.135 \\ -0.192 - 0.615 - 0.465 \\ 0.1152 + 0.1722 - 1.215 \end{bmatrix} = \begin{bmatrix} 1.6414 \\ -1.272 \\ -0.9276 \end{bmatrix}.$$

The following graphic illustrates how the three entries $\boldsymbol{x}_t$, namely $x_1(t)$, $x_2(t)$, $x_3(t)$, vary over time as $0 \leq t \leq 50$.



The following code is used to create the graphic above:

```
import numpy as np

x_1 = np.array([2,0,-2])            # initial vector of dynamic system
n = len(x_1)                        # this will be 3
T = 30                              # total time (displayed on the x-axis)
A = np.array([[0.87, 0.2, -0.09],   # transition matrix
              [-0.1, 0.75, 0.31],
              [0.06,-0.21, 0.81]])
x = np.zeros((T,len(x_1)))          # initialize dynamical system, starts with all zeros
x[0] = x_1                          # set the first vector of dynamical system

for t in range(T-1):
    x[t+1] = A @ x[t]               # the dynamic equation x_{t+1} = Ax_t

import matplotlib.pyplot as plt     # See Appendix for explanation of plt
for i in range(len(x[0])):
    plt.plot(np.arange(T),x[:,i],styles[i],c=colors[i],linewidth=lnwth[i])
plt.legend(['(x_t)_1','(x_t)_2','(x_t)_3'])
plt.xlabel('t')
```

■

One can naturally generalize the dynamical equation $\boldsymbol{x}_{t+1} = Ax\boldsymbol{x}_t$. First, the next state may depend on multiple previous states, for example, (called a *K-Markov chain*)

$$\boldsymbol{x}_{t+1} = A_1\boldsymbol{x}_t + A_2\boldsymbol{x}_{t-1} + \ldots + A_K\boldsymbol{x}_{t-K+1}.$$

It turns out that this situation can be made into a simple dynamical system by using block vectors. Let $\boldsymbol{z}_t = \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{x}_{t-1} \\ \boldsymbol{x}_{t-2} \\ \boldsymbol{x}_{t-3} \\ \vdots \\ \boldsymbol{x}_{t-K+1} \end{bmatrix}$ be a block Kn-vector and let $A = \begin{bmatrix} A_1 & A_2 & A_3 & \cdots & A_{K-1} & A_K \\ I_n & 0 & 0 & \cdots & 0 & 0 \\ 0 & I_n & 0 & \cdots & 0 & 0 \\ 0 & 0 & I_n & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & I_n & 0 \end{bmatrix}$ be a block $Kn \times Kn$ matrix. Then

$$A\boldsymbol{z}_t = \begin{bmatrix} A_1\boldsymbol{x}_t + A_2\boldsymbol{x}_{t-1} + \ldots + A_K\boldsymbol{x}_{t-K+1} \\ \boldsymbol{x}_t \\ \boldsymbol{x}_{t-1} \\ \vdots \\ \boldsymbol{x}_{t-K+2} \end{bmatrix}.$$

3.7.3 Control

It could be there are factors $\boldsymbol{u}_t$ that affect the dynamics of $\boldsymbol{x}_t$ that are not reported in the vector $\boldsymbol{x}_t$ itself. In this case, we can use the more generalized dynamical equation

$$\boldsymbol{x}_{t+1} = A_t\boldsymbol{x}_t + B\boldsymbol{u}_t + \boldsymbol{c}_t,$$

where $\boldsymbol{u}_t$ is some *input m-vector*, B_t is $m \times n$ *input matrix*, and $\boldsymbol{c}_t$ is the *offset n-vector*. This can also be turned into a simple dynamical system (assuming there are $n \times m$, $m \times m$, and $n \times n$ matrices C_t, D_t, E_t) as

$$\begin{bmatrix} A_t & B_t & I_n \\ C_t & D_t & E_t \\ 0 & 0 & I_n \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{u}_t \\ \boldsymbol{c}_t \end{bmatrix}.$$

Example 3.7.5. Consider the linear dynamical system

$$\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t + B\boldsymbol{u}_t.$$

As we discussed, the input m-vector $\boldsymbol{u}_t$ influences the dynamics of $\boldsymbol{x}_{t+1}$, but what if $\boldsymbol{u}_t$ is something that we can *control*. In many linear dynamical systems, the input vector $\boldsymbol{u}_t$ is a linear output of the state n-vector $\boldsymbol{x}_t$, meaning there exists a $m \times n$-matrix K such that

$$\boldsymbol{u}_t = K\boldsymbol{x}_t.$$

This situation is called **state feedback control**, and the matrix K is called the **state-feedback gain matrix**. Therefore,

$$\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t + B\boldsymbol{u}_t = A\boldsymbol{x}_t + B(K\boldsymbol{x}_t) = (A + BK)\boldsymbol{x}_t.$$ ∎

Example 3.7.6. Consider a time-invariant linear dynamical system with equation $\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t$. Then

$$\boldsymbol{x}_t = I_n\boldsymbol{x}_t = A^0\boldsymbol{x}_t, \qquad \boldsymbol{x}_{t+1} = A\boldsymbol{x}_t = A^1\boldsymbol{x}_t, \qquad \boldsymbol{x}_{t+2} = A\boldsymbol{x}_{t+1} = A(A\boldsymbol{x}_t) = A^2\boldsymbol{x}_t,$$
$$A\boldsymbol{x}_{t+3} = A(A\boldsymbol{x}_{t+2}) = A^2\boldsymbol{x}_{t+1} = A^2(A\boldsymbol{x}_t) = A^3\boldsymbol{x}_t, \ldots, \boldsymbol{x}_{t+k} = A\boldsymbol{x}_{t+k-1} = A^2\boldsymbol{x}_{t+k-2} = \ldots = A^k\boldsymbol{x}_t.$$

Therefore, $\boldsymbol{x}_{t+k} = A^k \boldsymbol{x}_t$.

Imagine now the dynamical system has some input, say $\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t + B\boldsymbol{u}_t$. Then

$$\begin{aligned} \boldsymbol{x}_{t+2} &= A\boldsymbol{x}_{t+1} + B\boldsymbol{u}_{t+1} = A(A\boldsymbol{x}_t + B\boldsymbol{u}_t) + B\boldsymbol{u}_{t+1} = A^2\boldsymbol{x}_t + AB\boldsymbol{u}_t + B\boldsymbol{u}_{t+1}, \\ &\vdots \\ \boldsymbol{x}_{t+k} &= A^k\boldsymbol{x}_t + A^{k-1}B\boldsymbol{u}_t + A^{k-2}B\boldsymbol{u}_{t+1} + \ldots + AB\boldsymbol{u}_{t+k-2} + B\boldsymbol{u}_{t+k-1} \end{aligned}$$

If we have state feedback control on $\boldsymbol{u}_t$, that is, $\boldsymbol{u}_t = K\boldsymbol{x}_t$, then $\boldsymbol{x}_{t+1} = (A + BK)\boldsymbol{x}_t$ and

$$\boldsymbol{x}_{t+k} = (A + BK)^k \boldsymbol{x}_t.$$

This illustrates the power of having control over the input vector $\boldsymbol{u}_t$. ∎

Exercises

(Go to Solutions)

For Exercises 1-6, compute the state vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$ given the transition matrix A, listed first, and the initial state $\boldsymbol{x}_0$, listed second.

1. $\begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}, \begin{bmatrix} 2 \\ 3 \end{bmatrix}$

♠ 3. $\begin{bmatrix} 4 & 2 & 6 \\ 3 & -5 & 4 \\ -2 & 7 & 5 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

♠ 4. $\begin{bmatrix} 1 & 2 & -3 & -4 \\ 6 & -1 & 3 & 2 \\ -4 & -2 & -1 & 5 \\ 2 & 2 & -5 & 3 \end{bmatrix}, \begin{bmatrix} 4 \\ -2 \\ 6 \\ 9 \end{bmatrix}$

5. $\begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 6 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$

6. $\begin{bmatrix} 4 & 2 & 3 & 1 & 5 \\ 3 & -2 & 3 & 2 & -3 \\ -1 & 2 & -3 & 4 & -5 \\ 5 & -4 & 3 & -2 & -1 \\ -2 & 1 & -2 & 3 & 4 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$

7. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/markov.ipynb

"Be dynamic, just like life. Innovate. Create. Explore. Never become stagnant." – Akin Olokun

3.8 Applications of Linear Dynamical Systems

Understanding the dynamics of growth is crucial in various scientific fields, including ecology, epidemiology, sociology, and economics, to name a few. Linear dynamical systems provide a powerful framework for modeling and analyzing such dynamic processes. In this section, we will explore a few examples where a linear dynamical system is employed to model the growth or decline of a "population." We will see how linear dynamical systems can be used for predictive modeling, allowing us to make informed predictions about future population trends based on the current conditions.

3.8.1 Population Growth

Example 3.8.1. Let us consider a simple scenario involving population dynamics in two cities. City A has a population of 800,000, and City B has a population of 500,000. The initial population vector would then be $\boldsymbol{x}_0 = \begin{bmatrix} 800000 \\ 500000 \end{bmatrix}$, where City A's population is listed first, followed by City B's population. Suppose further that every year, 20% of City A's population moves to City B, and 10% of City B's population moves to City A. If these migration trends remain constant for 10 years, then the population changes can be modelled using a linear dynamical system with population transition matrix given as

$$A = \begin{bmatrix} 1-0.2 & 0.2 \\ 0.1 & 1-0.1 \end{bmatrix} = \begin{bmatrix} 0.8 & 0.2 \\ 0.1 & 0.9 \end{bmatrix}.$$

The following code is used to simulate this linear dynamical system for ten years:

```
import numpy as np

x_0 = np.array([800000,500000]) # Initial state vector [Population_A,Population_B]
A = np.array([[0.8,0.1],        # Define the state transition matrix A
              [0.2,0.9]])

# Simulate the population dynamics for T years
T = 10                          #number of years
x = np.zeros((T+1,len(x_0))) #initialize linear dynamical system
x[0] = x_0
print(f"Year 0: Population A = {x[0,0]:.0f}, Population B = {x[0,1]:.0f}")
for t in range(T):
    x[t+1] = A @ x[t]        #the dynamic equation
    print(f"Year {t + 1}: Population A = {x[t+1,0]:.0f}, Population B = {x[t+1,1]:.0f}")
```

which produces the output

```
Year 0:  Population A = 800000, Population B = 500000
Year 1:  Population A = 690000, Population B = 610000
Year 2:  Population A = 613000, Population B = 687000
Year 3:  Population A = 559100, Population B = 740900
Year 4:  Population A = 521370, Population B = 778630
Year 5:  Population A = 494959, Population B = 805041
Year 6:  Population A = 476471, Population B = 823529
Year 7:  Population A = 463530, Population B = 836470
Year 8:  Population A = 454471, Population B = 845529
Year 9:  Population A = 448130, Population B = 851870
Year 10:  Population A = 443691, Population B = 856309
```

■

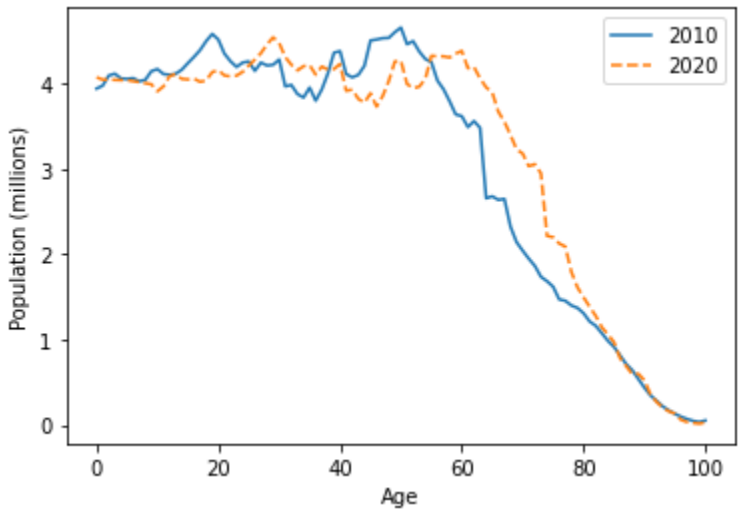
Example 3.8.2. Consider a dynamical system where $\boldsymbol{x}_t$ is a 101-vector, where $x_{t,i}$ denotes the number of i-year-old people in a certain population, where $\boldsymbol{x}_t = (x_{t,0}, x_{t,1}, x_{t,2}, \ldots, x_{t,99}, x_{t,100})^\top$ (note that $x_{t,100}$ encapsulates not just those people who are 100-years old but also all those over 100). Let t then denote one

year of time. Thus, as t transitions to $t+1$, everyone who is i-years-old will then move into the $(i+1)$-years-old category. But, unfortunately, there might be some in this group who passed away in the previous year. Therefore, some percentage is removed as it moves into the higher group. Also, some percentage of the population will give birth to new members of the population, which will all be added into the newborn group, that is, $i = 0$.[i] With these considerations, the population dynamic would then be a linear dynamical system with transition matrix

$$A = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 & \dots & b_{98} & b_{99} & b_{100} \\ 1-d_0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1-d_1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1-d_2 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-d_3 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1-d_{98} & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 1-d_{99} & 1-d_{100} \end{bmatrix}$$

where b_i and d_i are the birth and death rates, respectively, for i-year-olds in year t.

Let us consider the age distribution of the USA in 2010 using data from the US Census Bureau and the US Center of Disease Control.[ii] For simplicity, we will assume that the birth and death rates do not change much year to year, which allows use to treat this as a time-invariant dynamical system. Therefore, $\boldsymbol{x}_0$ denotes the US population in 2010. Thus, $\boldsymbol{x}_1$ is the US population in 2011, $\boldsymbol{x}_2$ is the US population in 2012, etc. Using this dynamical system, we can estimate the US population for 2020 by computing $\boldsymbol{x}_{10}$. Both the 2010 population and the predicted 2020 population are illustrated below.



To find the total US population for t years after 2010, we compute $\boldsymbol{x}_t = A^t\boldsymbol{x}_0$. To find the total population, no longer stratified for age, we compute $\mathbf{1}\cdot(A^k\boldsymbol{x}_0) = (\mathbf{1}^\top A^k)\boldsymbol{x}_0$. Note that the US population in 2020 was 309 million and our projected 2020 population was 322 million. According to the US 2020 Census, the 2020 population is 330 million.[iii]

The following code is used to solve this linear dynamical system:

```
import numpy as np
A= np.vstack((b,     # Creates transition matrix A using the birth/death rates for each age
              np.block([np.diag(1-d[:99]),np.zeros((99,2))]),
              (1-d[99])*np.eye(1,101,99) + (1-d[100])*np.eye(1,101,100)))
```

[i] Another major contributor to the population dynamics that we choose to ignore in this model would be migration, that is, some number of people moved into the population and some number moved out. This would be recorded in the $\boldsymbol{u}_t$ part of the dynamical equation as the rate of migration is not necessarily related to the current population vector $\boldsymbol{x}_t$.

[ii] See https://www.cdc.gov/nchs/data/nvsr/nvsr61/nvsr61_01.pdf.

[iii] The discrepancy mostly is due to our lack of consideration of immigration into the US.

```
year=2020                          # The year on which to terminate the simulation

import matplotlib.pyplot as plt # See Appendix for explanation of plt
plt.plot(x)                        # Plots the values for 2010 (initial values)
for i in range(year-2010):         # Using 2020, this loop will run 10 times.
    x = A @ x
plt.plot(x,"--")
plt.xlabel('Age')
plt.ylabel('Population (millions)')
plt.legend(['2010',str(year)])
plt.show()

print("2010 Population:",round(sum(P),1),"million")
print("Predicted",year,"Population:",round(sum(x),1),"million")
print("Actual 2020 Population: 329.5 million")
```

3.8.2 Infectious Spread

Example 3.8.3. The spread of an infestious disease can be modeled using a linear dynamical system. Suppose that a disease is introduced into a population. After each period of time (maybe days or weeks), we count the proportion of the total population that is infected, but we break it up into four infection states:

- *Susceptible* - These individuals are not currently infected but are capable of becoming infected by the disease in the next state if they are exposed to the disease.
- *Infected* - These individuals are currently infected with the disease.
- *Recovered* - These individuals have had the disease, survived it, and now have immunity to the disease and cannot infected again. This group also includes those who have been vaccinated against the disease and thus also have immunity.
- *Deceased* - These individuals had the disease but did not survive it. They are removed from the population count because they passed away.

We will represent the current state $\boldsymbol{x}_t$ as a 4-vector of percentages of the population in these four groups, for example, $\boldsymbol{x}_t = (0.75, 0.1, 0.1, 0.05)^\top$ means that 75% of the population has not been infected but could catch the disease, 10% currently have the disease, 10% have recovered and/or are immune to the disease, and 5% have died from the disease.

We need to next consider how an individual migrated among these groups, which will lead to our transition matrix.[iv]

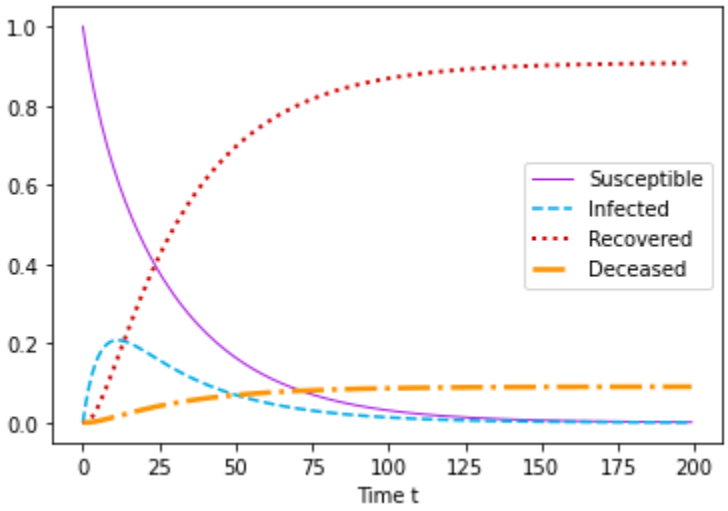
- Assume 5% of the susceptible population acquires the disease each state.
- Assume 1% of the current infected populatoin dies from the disease by the next state. Likewise, 10% recover and become immune, 4% recover but are still susceptible (that is, they did not acquire immunity).

[iv] In practice, a nonlinear dynamical system would be more accurate since it would be able to consider the infection rate based upon the current percentage of infected individuals compared to the current percentage of susceptible individuals. Our model also does not consider how the disease spreads and instead assumes that the susceptible will eventually become infected at a constant rate.

Under these assumptions, our linear dynamical system is time-invariant with the following transition matrix

$$A = \begin{bmatrix} 0.95 & 0.04 & 0 & 0 \\ 0.05 & 0.85 & 0 & 0 \\ 0 & 0.10 & 1 & 0 \\ 0 & 0.01 & 0 & 1 \end{bmatrix}.$$

Using the dynamic equation, $\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t$ and the initial value $\boldsymbol{x}_0 = (1, 0, 0, 0)^\top$, the diagram illustrates the simulation for 200 days after the initial infection. We note that by day 100 the epidemic has essentially stabilized toward 10% deceased and 90% immune.



The following code is used to solve this linear dynamical application:

```
import numpy as np

T = 200
A = np.array([[0.95,0.04,0,0],
              [0.05,0.85,0,0],
              [0,0.1,1,0],
              [0,0.01,0,1]])
x = np.zeros((T,4))
x[0] = np.array([1,0,0,0])

for t in range(T-1):
    x[t+1] = A @ x[t]

import matplotlib.pyplot as plt
```

```
for i in range(len(x[0])):
    plt.plot(np.arange(T),x[:,i],styles[i],c=colors[i],linewidth=lnwth[i])
plt.xlabel('Time t')
plt.legend(['Susceptible','Infected','Recovered','Deceased'])
plt.show()
```

■

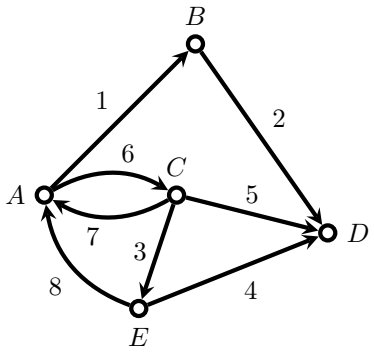
3.8.3 Directed Graphs and Logisitics

Let Γ be directed graph representing a network of stores selling a commodity. The vertices of Γ are stores $i = 1, 2, 3, \ldots, n$. Let the edges of Γ representing shipping lines between two stores (the direction of such an edge is only referential, as in it does not matter which direction the arrow points). Let $\boldsymbol{x}_t$ be the n-vector recording the surplus each store has of the commodities, that is, $x_{t,i}$ is the number of extra commodities store i has at time t. If a store needs some commodities, then $x_{t,i} < 0$. At each store, customers purchase $s_{t,i}$ amount of the commodity at time t, and $\boldsymbol{s}_t$ denotes the sales vector for time t. Likewise, the amount

store i purchases more commodities from the wholesaler at time t is $p_{t,i}$. Hence, $\boldsymbol{p}_t$ denotes the purchase vector for time t. We allow $p_{t,i}$ or $s_{t,i}$ to possibily be negative to account for possible returns. Certainly, as $\boldsymbol{x}_t$ transitions into $\boldsymbol{x}_{t+1}$, we add $\boldsymbol{p}_t - \boldsymbol{s}_t$ to $\boldsymbol{x}_t$.

Definition 3.8.4. For a directed graph Γ with m verices and n edges, the **incidence matrix**[v] A is the $m \times n$ matrix such that $a_{ij} = 1$ when edge j points *toward* vertex i, $a_{ij} = -1$ when edge j points *from* vertex i, and $a_{ij} = 0$ otherwise.

Example 3.8.5. Consider the graph Γ, illustrated below. Its incidence matrix is displayed below right.



$$A = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & -1 & 1 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 1 & -1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & -1 \end{bmatrix}$$

■

Returning to the supply chain dynamic, let $\boldsymbol{f}_t$ be the flow vector when $f_{t,j}$ denotes the number of commodities flowing along edge j. If $f_{t,j} > 0$, then this indicates the flow is in the same direction as edge j. If $f_{t,j} < 0$, then this indicates the flow is in the opposite direction as edge j. If A denotes the incidence matrix of the store network Γ, then

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + A\boldsymbol{f}_t + (\boldsymbol{p}_t - \boldsymbol{s}_t).$$

As stores have direct control on their purchases from the suppliers and shipping between the stores and have no control on the amount the customers buys at time t, we actually treat $-\boldsymbol{s}_t$ as the offset vector and the block $(m+n)$-vector $(\boldsymbol{f}_t, \boldsymbol{p}_t)^\top$ as the input vector. Then the dynamic equation becomes

$$\boldsymbol{x}_{t+1} = I_n \boldsymbol{x}_t + \begin{bmatrix} A & I_n \end{bmatrix} \begin{bmatrix} \boldsymbol{f}_t \\ \boldsymbol{p}_t \end{bmatrix} - \boldsymbol{s}_t.$$

[v]The *incidence matrix* introduced here should not be confused with the *adjacency matrix* introduced in Section **??**.

Exercises

(Go to Solutions)

For Exercises 1-1, compute the transition matrix A and initial state vector $\boldsymbol{x}_0$ for the given linear dynamical system.

♠ 1. The spread of a zombie infestation in the USA over a 12 month period, given that all humans in the USA are at risk with an attrition rate of 10% each month. Assume everyone that dies, turns into a zombie. Zombies do not ever become human again[vi] and humans manage to kill about 5% of the zombies each month. The initial population of the USA is 330 million and there are initially 10 zombies.

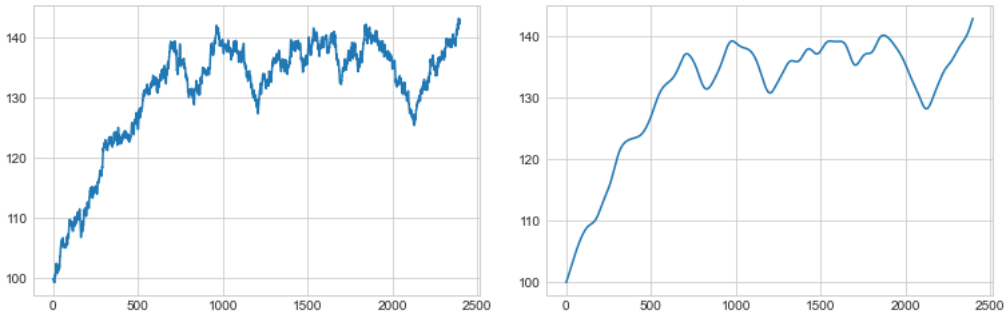
2. In the decade 2010-2020, the US gained on average 1,063,000 immigrants per year. Assuming the distribution of ages for immigrants entering the US was the same as those residents already living in the US, estimate the 2020 US population using the same data from Example 3.8.2. How many 20 year-olds would the US have in 2020?

3. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/markovapps.ipynb

[vi] No warm bodies love story here.

Python Projects

1. The data set below is a 2400-vector representing the stock value (in USD) of the company *Open Linear* as a time series for every minute of trading from last week. The stock started last week (Monday at 9:00 am) at \$100. A line-graph illustrating the stock's price drops and gains is given below left. It frankly looks very rough. Use convolution to smooth this time series so that the number of turning points on the graph is less than 30, such as explained in Section 3.2.3. Also, convolution typically extends the number of entries in a vector, and applying convolution toward smoothing graphs often creates bizarre behavior on the boundary. Your final vector must still have 2400 entries but not have bizarre behavior on the edges. An example of such is illustrated below right. Python code is started for you at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/convolution.py



2. Cryptography is the science of encrypting and decrypting messages for secure communications. One cryptosystem involving matrices is called the *Hill cipher*.[vii] Imagine Alice has a secret message, $\boldsymbol{x}$, which is an n-vector she wants to share with Bob. This secret message is called the *plaintext*. She is afraid that Eve will eavesdrop on her communication with Bob and discover the plaintext. Therefore, Alice and Bob had previously decided upon an $n \times n$ matrix A and an n-vector $\boldsymbol{b}$ to use as their secret key. The Hill cipher works in the following way. Alice calculates the affine image of the plaintext with respect to A and $\boldsymbol{b}$, that is, Alice calculates $\boldsymbol{y} = A\boldsymbol{x} + \boldsymbol{b}$. This n-vector is called the *ciphertext*. Alice then shares the ciphertext with Bob. Because Bob knows A and $\boldsymbol{b}$, he can compute the plaintext, namely $\boldsymbol{x} = A^{-1}(\boldsymbol{y} - \boldsymbol{b})$. On the other hand, because Eve does not have the key she would have to guess $(A, \boldsymbol{b})$ to try to decrypt $\boldsymbol{y}$, which would take forever. Therefore, Alice and Bob can communicate securely, so long as they keep their key secret from Eve.

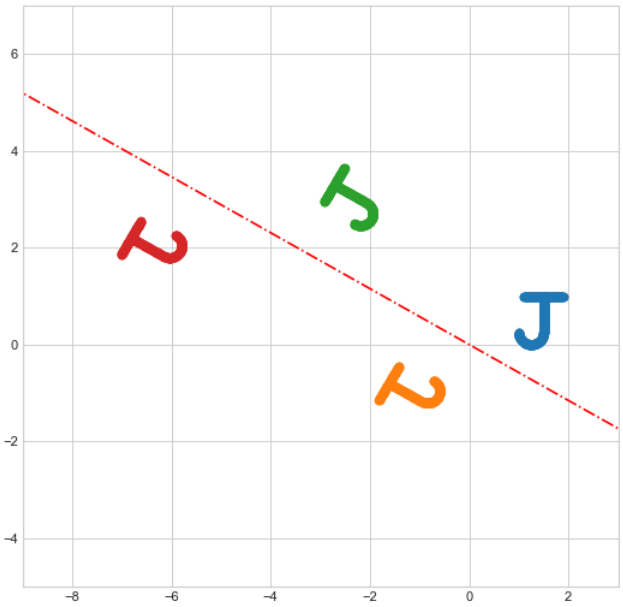
   This strategy also works when the plaintext is a list of n-vectors by iteration. For this project, I will play the role of Alice and you will play the role of Bob. In code below, you will find the 10×10 matrix A and the 10-vector $\boldsymbol{b}$ that we will use for our encrypted communication. The procedure `PadMessage` is a helper to convert plain English into a list of vector. Likewise, `DepadMessage` is a helper to convert a list of vectors into plain English. You will be able to ignore these procedures. The procedure `HillEncrypt` encrypts the plaintext message (written in plain English) using the secret key $(A, \boldsymbol{b})$. It will then output a list of 10-vectors (the length of this list depends on the length of the plaintext), which is the ciphertext. For example, using

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} 5 \\ 6 \end{bmatrix},$$

[vii] Note that the Hill cipher was devised to use modular arithmetic, but this implementation has been adapted to use floating points.

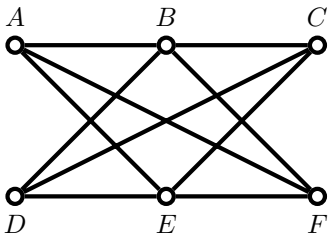
one can encrypt the message `"Linear Algebra is Awesome!"` as $\begin{bmatrix} 753882 & 1979583 \\ 946448 & 2516513 \\ 636791 & 1544640 \\ 928063 & 2219336 \\ 1473 & 4408 \end{bmatrix}$. Decrypt the linked ciphertext using our private key from above. The problem is that the code to `HillDecrypt` is incomplete. Repair the missing part of this code. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/Hillcipher.py

3. In the image to the right, you will see four J's in the plane. Using Python, reproduce this image using affine transformations, such as explained in Section 3.5. Python code is started for you below with the first J, but you will need to graph the other three :
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/moveJ.py



4. Recall that a walk along a graph is a sequence of adjacent vertices on a graph, such as explained in Section 3.2.1 Related to this notion is that of a random walk, that is, a walk where the next vertex in the walk is selected randomly among the adjacent vertices in accordance to some probability function. This project will help us explore a random walk along a graph. Python code will be helpful to simulate the associated linear dynamical system. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/randomwalk.py

   (a) The undirected graph Γ is illustrated to the right. There are six vertices and 10 edges. Let $A(\Gamma)$ denote the adjacency matrix for this graph. Find $A(\Gamma)$.



   (b) Consider a random walk along this graph Γ. Suppose that the choice among each available edge is equally likely, that is, if at vertex A then there is a $\frac{1}{3}$ chance of moving to B, a $\frac{1}{3}$ chance of moving to E, and a $\frac{1}{3}$ chance of moving to F in the next state of the walk. Similarly, if you are at vertex B, then there is a $\frac{1}{4}$ chance of moving to A, a $\frac{1}{4}$ chance of moving to C, a $\frac{1}{4}$ chance of moving to D, and a $\frac{1}{4}$ chance of moving to F.

   Let $\boldsymbol{x}_t$ be a 6-vector that records in entry x_i the probability of our person being at vertex i at state t in the random walk. For example, consider the random walk where the person starts at vertex B. Then

$$\boldsymbol{x}_0 = (0, 1, 0, 0, 0, 0),$$

since, with certainty, the person is at B for $t = 0$. At $t = 1$,

$$\boldsymbol{x}_1 = (0.25, 0, 0.25, 0.25, 0, 0.25),$$

since there is a 25% chance that the person walks to the vertices A, C, D, or F. There is a 0% chance that the person is at B or E after one step. Next, at $t = 2$,

$$\boldsymbol{x}_2 = (0.08333, 0.33333, 0.08333, 0.08333, 0.33333, 0.08333),$$

etc. Create a linear dynamical system $\boldsymbol{x}_{t+1} = T\boldsymbol{x}_t$ such that $\boldsymbol{x}_t$ is the 6-vector of probabilities that a person following a random walk will be at vertex i after t steps. The transition matrix T will be time-invariant and can be considered a modification of the adjacency matrix for Γ. What is the relationship between these two matrices? Starting at vertex A, what is $\boldsymbol{x}_3$ (of course, $\boldsymbol{x}_0 = (1, 0, 0, 0, 0, 0)$ in this case)? What is $\boldsymbol{x}_{10}$? What is $\boldsymbol{x}_{100}$? What pattern do you see emerging?

(c) Repeat the previous part for a random walk starting at vertex B. What pattern do you see emerging here? What happens if another vertex is the starting location? Make a conjecture about $\boldsymbol{x}_t$ as $t \to \infty$. How could this information had been gleaned from the original graph in retrospect?

5. In a large forest area, there lives a population of approximately 100 thousand rabbits. The game warden who oversees this forest, has noticed that the rabbit population has been falling over the last few years. In order to bolster up the rabbit population, the warden wants to recommend that the number of hunting tags for the rabbits' primary predator, foxes, be increased to lower the fox population, currently around 12 thousand, which should allow the rabbit population to climb. The warden's recommendation it to hunt approximating 100 foxes each year. An ecologist who studies this forest, including the rabbits and foxes, counters that both populations of rabbits and foxes are healthy and hunting more foxes who do more damage, as the forest only has a carrying capacity of approximately 200 thousand rabbits. If $\boldsymbol{x}_t = (x_{t,1}, x_{t,2})^\top$, where $x_{t,1}$ is the number of rabbits and $x_{t,2}$ is the number of foxes at year t, use the transition matrix $T = \begin{bmatrix} 1.9 & -8 \\ 0.1 & 0.1 \end{bmatrix}$ to model both the ecologist's and warden's predictions. Who is right based upon these models? Defend your answer. This project will utilize a linear dynamical system, such as explained in Section 3.8. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/rabbitfox.py

6. In this project, you continue the programs of Chapter **??** and write Python functions to compute a row-reduced echelon form and compute the inverse of matrix, such as explained in Sections 2.3.1 and 3.3. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/RREF.py

(a) The Backward Phase Algorithm transforms an echelon matrix into row reduced echelon form. This is done recursively using three procedures. The first is to locate the next pivot. The second is, once the pivot position is located, to scale the leading entry into a 1. Lastly, the third procedure is to place a zero into every entry above the pivot. Write a Python function `backward_phase(coefficient_matrix, constant_matrix, pivots)` which takes input parameters:[vii] `coefficient_matrix` is A and `constant_matrix` is $\boldsymbol{b}$ from a linear system $[A \mid \boldsymbol{b}]$, and `pivots` will be the set of pivot positions, listed as ordered pairs (i, j), of the leading entries in `coefficient_matrix`. The matrix A must be in echelon form. The function will return two objects: `reduced_coefficient_matrix` and `reduced_costant_matrix`. `reduced_coefficient_matrix` and `reduced_constant_matrix` will be row-reduced versions of `coefficient_matrix` and `constant_matrix`, respectively, accomplished by repeated usage of `zero_above` for each pivot column. Note that `reduced_coefficient_matrix` must be in row-reduced echelon form. Verify your code on the example:

$$\left[\begin{array}{cccc|c} \boxed{1} & 1 & 2 & 3 & 4 \\ 0 & 0 & \boxed{-4} & -8 & -12 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \sim \left[\begin{array}{ccccc} \boxed{1} & 1 & 0 & -1 & -2 \\ 0 & 0 & \boxed{1} & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right].$$

(b) Combining the Forward Phase and Backward Phase algorithms, we can compute the row-reduced echelon form of a matrix. Write a Python function `rref(coefficient_matrix, constant_matrix, pivots)` which takes input parameters:[vii] `coefficient_matrix` is A and `constant_matrix` is $\boldsymbol{b}$ from a linear system $[A \mid \boldsymbol{b}]$. The function will return two objects: `reduced_coefficient_matrix` and `reduced_costant_matrix`. `reduced_coefficient_matrix` and `reduced_constant_matrix` will be row-reduced versions of `coefficient_matrix` and `constant_matrix`, respectively, accomplished by using the `forward_phase` and `backward_phase` functions written previously. Verify your code on Example 2.3.2.

(c) The Inversion Algorithm is a method to compute matrix inverses using row reduction, in particular, if A is a nonsingular matrix, then $[A \mid I_n] \sim [I_n \mid A^{-1}]$. Using your `rref(coefficient_matrix, constant_matrix)` function to implement this into NumPy. Most specifically, write a Python function `matrix_inverse(matrix)` which takes input parameter: `matrix` is a nonsingular matrix A. The function outputs the inverse of A, namely A^{-1}, using the `rref` function written previously. This function must use your `rref` function. Verify your code on Example 3.3.6.

7. Recall that the null space of an $m \times n$ matrix A, denoted $\text{Nul}(A)$, is the set of all vectors $\boldsymbol{x} \in \mathbb{R}^n$ such that $A\boldsymbol{x} = \boldsymbol{0}$. As the name suggests, the null space is a subspace of $\mathbb{R}^n$. Thus, we can express each element of $\text{Nul}(A)$ as a unique linear combination of elements from its basis, such as explained in Section 2.8.1. In Section 2.8.1, we previously learned an algorithm to compute a basis for $\text{Nul}(A)$ using the row reduced echelon form of A. Write a Python function `Null(matrix)` which takes input parameters: `matrix` is A. The function will return an array of vectors (hence a 2d-np.array): `null`, which is a basis for $\text{Nul}(A)$ calculated from the row-reduced echelon form of A, using the `rref` function written previously. Verify your code on Example 2.8.4. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/null_space.py

Chapter 4

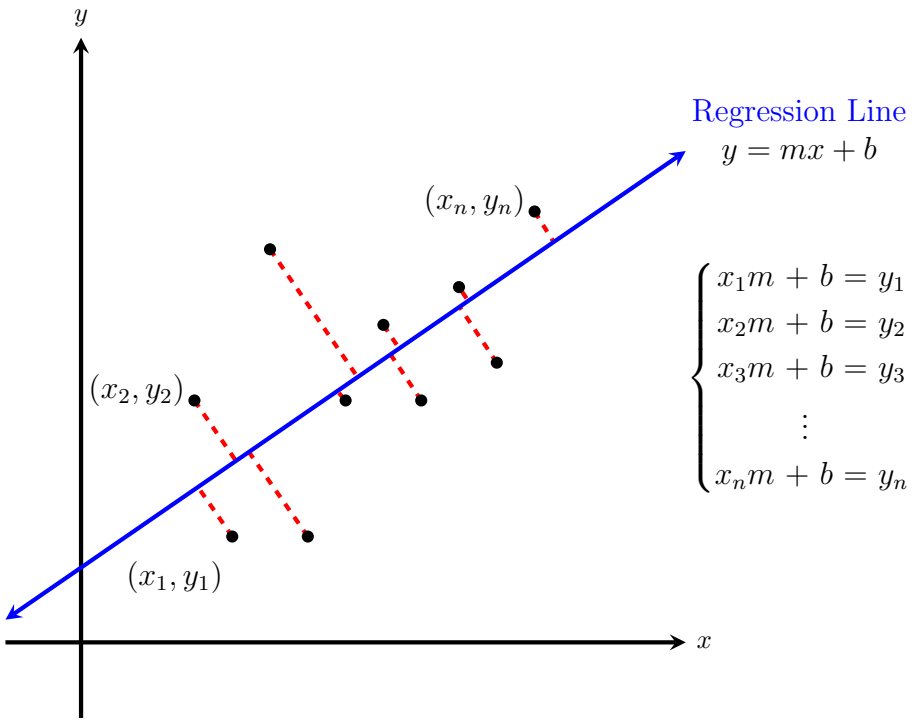
Least Squares

"Hypotheses are what we lack the least." – Henri Poincare

4.1 The Least Squares Problem

4.1.1 Normal Equations

In the past, we have considered the matrix equation $A\boldsymbol{x} = \boldsymbol{b}$. We have learned how to determine if the equation has a solution and how to calculate the solution set when it is consistent. But what about when $A\boldsymbol{x} = \boldsymbol{b}$ is inconsistent? An inconsistent system has an empty solution set, but what if we really needed it to have a solution? No amount of begging will change the fact that the matrix equation does not have a solution. For example, what if we want to find a line that is incident to all of the below points. As these points are not collinear, no such line exists. But is there a line that best fits the points, a so-called *regression line*?



Instead of asking, "Is there a vector $\boldsymbol{x}$ such that $A\boldsymbol{x} = \boldsymbol{b}$," we ask, "Is there a vector $\boldsymbol{x}$ such that $A\boldsymbol{x} \approx \boldsymbol{b}$?" By approximation, we mean to find a vector $\boldsymbol{x}$ such that $\|A\boldsymbol{x} - \boldsymbol{b}\|$ is sufficiently small. This leads to the **least-squares problem**.

Definition 4.1.1. If A is an $m \times n$ matrix and $\boldsymbol{b} \in \mathbb{R}^m$, a **least-squares solution** of $A\boldsymbol{x} = \boldsymbol{b}$ is a vector $\widehat{\boldsymbol{x}} \in \mathbb{R}^n$ such that

$$\|\boldsymbol{b} - A\widehat{\boldsymbol{x}}\| \leq \|\boldsymbol{b} - A\boldsymbol{x}\|$$

for all $\boldsymbol{x} \in \mathbb{R}^n$

Note that if $A\boldsymbol{x} = \boldsymbol{b}$ is consistent, then there exists a vector $\widehat{\boldsymbol{x}}$ such that $A\widehat{\boldsymbol{x}} = \boldsymbol{b}$ and $\|\boldsymbol{b} - A\widehat{\boldsymbol{x}}\| = \|\boldsymbol{0}\| = 0$. Thus, any solution is a least-square solution. On the other hand, if $A\boldsymbol{x} = \boldsymbol{b}$ is inconsistent, then the matrix equation has no solution but must certainly still have a least-square solution. To find this least-squares solution, let us change our focus. Why is the linear system inconsistent and who should be blamed for it? Well, the vector $\boldsymbol{b}$, sort of.

Let A be an $m \times n$ matrix and $\boldsymbol{b} \in \mathbb{R}^m$. Let $\boldsymbol{x} \in \mathbb{R}^n$ be a generic vector (treat $\boldsymbol{x}$ as a variable vector). Then $A\boldsymbol{x}$ is a generic vector of the subspace $\operatorname{Col} A$. Let $\widehat{\boldsymbol{b}} = \operatorname{proj}_{\operatorname{Col} A} \boldsymbol{b} = \boldsymbol{b}_{\operatorname{Col}(A)}$. Then

$$\|\boldsymbol{b} - \widehat{\boldsymbol{b}}\| \leq \|\boldsymbol{b} - A\boldsymbol{x}\|$$

by the Best Approximation Theorem. Since $\widehat{\boldsymbol{b}} \in \operatorname{Col} A$, there exists some vector $\widehat{\boldsymbol{x}} \in \mathbb{R}^n$ such that $A\widehat{\boldsymbol{x}} = \widehat{\boldsymbol{b}}$. Therefore, every linear system has a least-squares solution $\widehat{\boldsymbol{x}}$, which is a solution to the linear system $A\boldsymbol{x} = \widehat{\boldsymbol{b}}$, since for all $\boldsymbol{x}$ we have

$$\|\boldsymbol{b} - A\widehat{\boldsymbol{x}}\| \leq \|\boldsymbol{b} - A\boldsymbol{x}\|.$$

Of course, we also have that the linear system $A\boldsymbol{x} = \widehat{\boldsymbol{b}}$ is ALWAYS consistent.

Given any $m \times n$ matrix A, let $\operatorname{proj}_{\operatorname{Col} A} : \mathbb{R}^m \to \operatorname{Col} A$ be the *orthogonal projection*[i] onto the column space of A, where $\widehat{\boldsymbol{b}} := \operatorname{proj}_{\operatorname{Col} A}(\boldsymbol{b})$ for $\boldsymbol{b}$. The orthogonal projection gets its name since $\widehat{A\boldsymbol{x}} = A\boldsymbol{x}$ and $(\boldsymbol{b} - \widehat{\boldsymbol{b}}) \cdot A\boldsymbol{x} = 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{b} \in \mathbb{R}^m$. In other words, the orthogonal projection acts like the identity to those vectors in the column space and the vector difference $\boldsymbol{b} - \widehat{\boldsymbol{b}}$ is orthogonal to all vectors in $\operatorname{Col} A$. This implies that $A^\top(\boldsymbol{b} - \widehat{\boldsymbol{b}}) \cdot \boldsymbol{x} = 0$ for all $\boldsymbol{x}$. As $\boldsymbol{x}$ is arbitrary, this can only happen if $A^\top(\boldsymbol{b} - \widehat{\boldsymbol{b}}) = \boldsymbol{0}$. This further implies that $A^\top\boldsymbol{b} - A^\top\widehat{\boldsymbol{b}} = \boldsymbol{0}$ or $A^\top\widehat{\boldsymbol{b}} = A^\top\boldsymbol{b}$. Therefore, $A^\top A\widehat{\boldsymbol{x}} = A^\top\boldsymbol{b}$. In particular, every least-squares solution to $A\boldsymbol{x} = \boldsymbol{b}$ is a solution to the system of **normal equations**

$$A^\top A\widehat{\boldsymbol{x}} = A^\top\boldsymbol{b}. \tag{4.1.1}$$

Theorem 4.1.2. *The set of least-squares solutions of $A\boldsymbol{x} = \boldsymbol{b}$ coincides with the solution set of the normal equations $A^\top A\widehat{\boldsymbol{x}} = A^\top\boldsymbol{b}$.*

Example 4.1.3. Find a least-squares solution of the inconsistent system $A\boldsymbol{x} = \boldsymbol{b}$ with

$$A = \begin{bmatrix} 4 & 0 \\ 0 & 2 \\ 1 & 1 \end{bmatrix} \text{ and } \boldsymbol{b} = \begin{bmatrix} 2 \\ 0 \\ 11 \end{bmatrix}.$$

We begin by constructing the normal equations:

$$A^\top A = \begin{bmatrix} 4 & 0 & 1 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 \\ 0 & 2 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 17 & 1 \\ 1 & 5 \end{bmatrix}; \qquad A^\top\boldsymbol{b} = \begin{bmatrix} 4 & 0 & 1 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 11 \end{bmatrix} = \begin{bmatrix} 19 \\ 11 \end{bmatrix}.$$

Thus, $A^\top A\widehat{\boldsymbol{x}} = A^\top\boldsymbol{b}$ becomes

$$\begin{bmatrix} 17 & 1 \\ 1 & 5 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 19 \\ 11 \end{bmatrix}.$$

Since $A^\top A$ is invertible, we have

$$\widehat{\boldsymbol{x}} = (A^\top A)^{-1}A^\top\boldsymbol{b} = \frac{1}{84}\begin{bmatrix} 5 & -1 \\ -1 & 17 \end{bmatrix}\begin{bmatrix} 19 \\ 11 \end{bmatrix} = \frac{1}{84}\begin{bmatrix} 84 \\ 168 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \qquad \blacksquare$$

Example 4.1.4. Find a least-squares solution for

[i] See Appendix C for more details on orthogonality.

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \text{ and } \boldsymbol{b} = \begin{bmatrix} -3 \\ -1 \\ 0 \\ 2 \\ 5 \\ 1 \end{bmatrix}.$$

$$A^\top A = \begin{bmatrix} 6 & 2 & 2 & 2 \\ 2 & 2 & 0 & 0 \\ 2 & 0 & 2 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix}; \qquad A^\top \boldsymbol{b} = \begin{bmatrix} 4 \\ -4 \\ 2 \\ 6 \end{bmatrix}; \qquad \left[\begin{array}{cccc|c} 6 & 2 & 2 & 2 & 4 \\ 2 & 2 & 0 & 0 & -4 \\ 2 & 0 & 2 & 0 & 2 \\ 2 & 0 & 0 & 2 & 6 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 3 \\ 0 & 1 & 0 & -1 & -5 \\ 0 & 0 & 1 & -1 & -2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right].$$

Therefore,

$$\widehat{\boldsymbol{x}} = \begin{bmatrix} 3 \\ -5 \\ -2 \\ 0 \end{bmatrix} + x_4 \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}. \qquad \blacksquare$$

As illustrated in the last example, a least-squares solution need not be unique.

Theorem 4.1.5. *Let A be an $m \times n$ matrix. Then the following are equivalent:*

(i) The equation $A\boldsymbol{x} = \boldsymbol{b}$ has a unique least-squares solution for each $\boldsymbol{b} \in \mathbb{R}^m$.

(ii) The columns of A are linearly independent.

(iii) The matrix $A^\top A$ is invertible.

In this situation, the unique least-squares solution has the form

$$\widehat{\boldsymbol{x}} = (A^\top A)^{-1} A^\top \boldsymbol{b}.$$

Corollary 4.1.6. *Let A be an $m \times n$ matrix with linearly independent columns. Let $A = QR$ be a QR factorization. Let $\boldsymbol{b} \in \mathbb{R}^m$. Then the unique least-squares solution $\widehat{\boldsymbol{x}} \in \mathbb{R}^n$ has the form*

$$\widehat{\boldsymbol{x}} = R^{-1} Q^\top \boldsymbol{b}.$$

4.1.2 Applications of Least-Squares

Let us consider some examples of least squares problems and their solutions. To analyze how good a least squares solution is for $A\widehat{\boldsymbol{x}} = \boldsymbol{b}$, we consider the **residual vector** $\boldsymbol{r} = \boldsymbol{b} - A\widehat{\boldsymbol{x}}$. Specifically, we want $\text{rms}(\boldsymbol{r}) = \text{rms}(\boldsymbol{b} - A\widehat{\boldsymbol{x}})$ to be small.

Example 4.1.7. Suppose a company is analyzing its advertisement strategy. Suppose the company advertises to m demographic groups or audiences. The company has a desired amount of views or impressions that it wants members of those groups to experience. Say that vector $\boldsymbol{t} = (t_1, t_2, \ldots, t_m)$ is the *target vector*

with regret to these impressions, where t_i is the amount of views the company wants group i to see. To reach these audiences, the company purchases advertisements in n different channels or media (maybe radio ads, TV ads, web ads, print ads, etc). Let $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ be the cost vector for advertising in these different channels, where c_j is the amount the company pays to channel j for advertisement. Let V_{ij} denote the average number of views produces for group i in channel j, measured in views per dollar spent. These values V_{ij} have been calculated by the company's market research division. Let $V = \begin{bmatrix} V_{ij} \end{bmatrix}$ be the associated $m \times n$ *viewing matrix*. For the company to acquire the targeted number of views $\boldsymbol{t}$, they must solve the least squares problem $V\boldsymbol{c} = \boldsymbol{t}$. This is a least squares problem because this linear system may be inconsistent as there may be no cost vector that perfectly produces the targeted average number of views in all audiences. Of course, a least squares solution gives the company the best marketing strategy to meet its target.

For example, suppose the company wants to advertise to $m = 5$ audiences via $n = 4$ advertising channels. Suppose that company want each of the audiences to have 600,000 views in the next advertising cycle, that is, $\boldsymbol{t} = 600\mathbf{1}$. Finally, suppose

$$V = \begin{bmatrix} 2.0 & 1.1 & 0.1 & 3.2 \\ 1.2 & 2.2 & 1.3 & 0.2 \\ 0.5 & 1.0 & 3.1 & 2.7 \\ 3.1 & 0.4 & 2.4 & 1.2 \\ 1.1 & 1.1 & 2.2 & 2.3 \end{bmatrix},$$

where V_{ij} is measured in 1000 views per \$1, that is, the company expects to garner 2000 views in audience 1 for each \$1 spent for advertisting in channel 1. Solving the least squares problem using `c = np.linalg.lstsq(V, t, rcond = None)[0]`. Then $\boldsymbol{c} = (\$95.02, \$176.77, \$65.20, \$65.69)$. With this cost vector, the views will be $V\boldsymbol{c} = (601, 601, 604, 601, 594)$, which gives a relative RMS error of 0.57% of the target impressions. ∎

Python Programming

Solving the least-squares problem $A\boldsymbol{x} = \boldsymbol{b}$ is similar to solving linear systems, in particular, we must solve the normal equations $A^\top A\boldsymbol{x} = A^\top \boldsymbol{b}$. This would be easy to implement in Python, that is, `np.linalg.solve(A.T@A, A.T@b)`, but easy than that is the command `c = np.linalg.lstsq(A, b, rcond = None)[0]`. There are multiple output to this function, but only the first output, the least-squares solution, is needed here.

Example 4.1.8. The code:

```
import numpy as np
V = np.array([[2.0,1.1,0.1,3.2],
              [1.2,2.2,1.3,0.2],
              [0.5,1.0,3.1,2.7],
              [3.1,0.4,2.4,1.2],
              [1.1,1.1,2.2,2.3]])
t = 600*np.ones(5) #target impressions
B = 375 #Budget

print("Target Impressions in each Demographic Group =\n", t, "\n")

c = np.linalg.lstsq(V,t, rcond=None)[0]
print("Expected Impressions in each Demographic Group =\n", [int(round(impress, 0)) for
    impress in V@c], "\n")
print("Relative RMS Error = " + str(round(100*(np.linalg.norm(V@c - t)/np.linalg.norm(t)),2)
    ) + "%\n\n")

c = lstsq_constraint(V, t, np.ones(4), B)
print("Fixed Budget: Cost = $" + str(B) +"\n")
print("Spending in each Channel = $", [round(cost,2) for cost in c], "\n")
print("Expected Impressions in each Demographic Group =\n", [int(round(impress, 0)) for
    impress in V@c], "\n")
```

solves the least-squares problem of Example **??** and produces the output:

```
Target Impressions in each Demographic Group =
 [600.  600.  600.  600.  600.]

No Budget:  Cost = $402.68

Spending in each Channel = $ [95.02, 176.77, 65.2, 65.69]

Expected Impressions in each Demographic Group =
 [601, 601, 604, 601, 594]

Relative RMS Error = 0.57%
```

■

Exercises

(Go to Solutions)

For Exercises 1-10, compute the least square solutions for linear system.

1. $\left[\begin{array}{cc|c} 3 & 0 & 4 \\ 4 & 1 & -1 \\ 8 & 3 & -2 \end{array}\right]$

♠ 2. $\left[\begin{array}{cc|c} 1 & 0 & 3 \\ 0 & 1 & 4 \\ 1 & 1 & 1 \end{array}\right]$

3. $\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 2 & 1 & 6 \\ 4 & 1 & 1 \end{array}\right]$

4. $\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 1 & 2 & 2 \\ 1 & 3 & 4 \end{array}\right]$

5. $\left[\begin{array}{cc|c} 2 & 1 & 5 \\ 1 & -1 & 0 \\ 3 & 2 & 7 \end{array}\right]$

♠ 6. $\left[\begin{array}{cc|c} 0 & 1 & 3 \\ 1 & 0 & -2 \\ 2 & -1 & -7 \\ 1 & 0 & -2 \end{array}\right]$

♠ 7. $\left[\begin{array}{ccc|c} -2 & 0 & 2 & -2 \\ 1 & -2 & 0 & 1 \\ 2 & 0 & 1 & 0 \\ -1 & 1 & 2 & -2 \\ 2 & 1 & 2 & -1 \end{array}\right]$

8. $\left[\begin{array}{ccc|c} 1 & -2 & -2 & 6 \\ 0 & 2 & -8 & -8 \\ -4 & 6 & 2 & 10 \end{array}\right]$

9. $\left[\begin{array}{cc|c} 1 & 0 & 1 \\ 1 & 1 & 2 \\ 1 & 2 & 2 \\ 1 & 3 & 5 \end{array}\right]$

10. $\left[\begin{array}{cc|c} 1 & 1 & 6 \\ 1 & 2 & 5 \\ 1 & 3 & 7 \\ 1 & 4 & 10 \end{array}\right]$

11. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/least.ipynb

"Celebrate the idea that you don't fit in. Find your own fit. Stay unique." – Betsey Johnson

4.2 Data Fitting

Recall the problem of **data fitting**[i] (or **interpolation**), that is, given a mathematical model $f(x) = \sum_{i=1}^{n} c_i f_i(x)$ where $c_i \in \mathbb{R}$ are scalars and f_i are real-valued functions and given a set of m points $(x_1, y_1), \ldots, (x_m, y_m)$, can we find the missing coefficients $c_1, \ldots, c_n$ so that the data fits into the model g. Substituting the points in the formula of $f(x)$ gives a linear equation in terms of the coefficients for each point (x_i, y_i) into the model g:

$$\begin{cases} c_1 f_1(x_1) + c_2 f_2(x_1) + \ldots + c_n f_n(x_1) = y_1 \\ c_1 f_1(x_2) + c_2 f_2(x_2) + \ldots + c_n f_n(x_2) = y_2 \\ \vdots \quad\quad \vdots \quad\quad \ddots \quad\quad \vdots \quad\quad \vdots \\ c_1 f_1(x_m) + c_2 f_2(x_m) + \ldots + c_n f_n(x_m) = y_m \end{cases}$$

We previously considered the case where $m = n$. Of course, if $m < n$, then no unique function can be determined, but often the case is that $m > n$, that is, we have more points than independent functions in the model. If we imagine the set of points in a scatter plot, then there might not actually be a function that perfectly fits the data. Of course, the least squares solution to this often inconsistent linear system offers the **best fit model**.

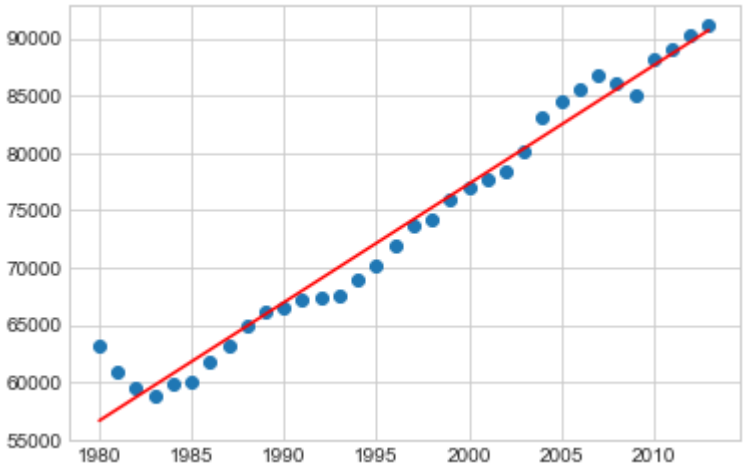
In the case of a polynomial function, the general template would be $f(x) = c_n x^n + \ldots + c_2 x^2 + c_1 x + c_0$. The coefficient matrix for a polynomial interpolation takes on a special form, called a **Vandermonde matrix**:

$$V(\boldsymbol{x}) = \begin{bmatrix} x_1^n & \cdots & x_1^3 & x_1^2 & x_1 & 1 \\ x_2^n & \cdots & x_2^3 & x_2^2 & x_2 & 1 \\ x_3^n & \cdots & x_3^3 & x_3^2 & x_3 & 1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n^n & \cdots & x_n^3 & x_n^2 & x_n & 1 \end{bmatrix}$$

Example 4.2.1. Consider the following time series of the world's annual petroleum consumption from 1980 to 2013 measured in thousands of barrels per day, as illustrated. While there is definitely some fluctuation illustrated here, the time series very much resembles a line. Setting t to denote the number of years since 1980, let p_t denote the average thousands of barrels per day for year t. Using the 34×2 Vandermonde matrix

$$V = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \\ \vdots & \vdots \\ 33 & 1 \end{bmatrix},$$

we solve the least squares problem $V\boldsymbol{c} = \boldsymbol{p}$, giving $\boldsymbol{c} = (1033, 56638)$. This means that the line of best fit to this data would be $y = 1033x + 56638$, which is also illustrated in the diagram.



[i] See Section 4.2.

The following Python code can be used to solve this example:

```
import numpy as np

#A 34-vector of the world annual petroleum consumption between
#1980 and 2013, in thousand barrels/day.
petroleum_data = np.array([
    63122, 60953, 59551, 58785, 59795, 60083, 61819, 63107, 64978, 66090,
    66541, 67186, 67396, 67619, 69006, 70258, 71880, 73597, 74274, 75975,
    76928, 77732, 78457, 80089, 83063, 84558, 85566, 86724, 86046, 84972,
    88157, 89105, 90340, 91195 ])

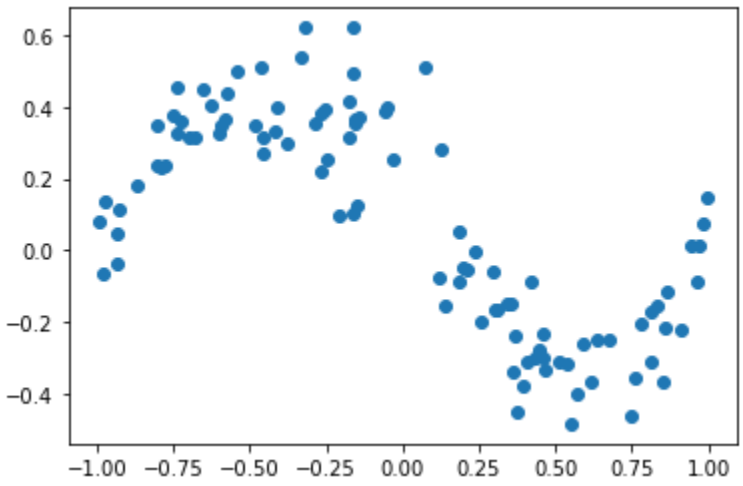
n = len(petroleum_data)
A = np.array([[k,1] for k in range(n)]) #np.vander(range(n),2)

x = np.linalg.lstsq(A,petroleum_data, rcond=None)[0]

import matplotlib.pyplot as plt
plt.scatter(np.arange(1980,2014), petroleum_data)
plt.plot(np.arange(1980,2014), A @ x, 'r')
plt.show()
```

■

Example 4.2.2. Consider the scatter plot of 100 points illustrated below.



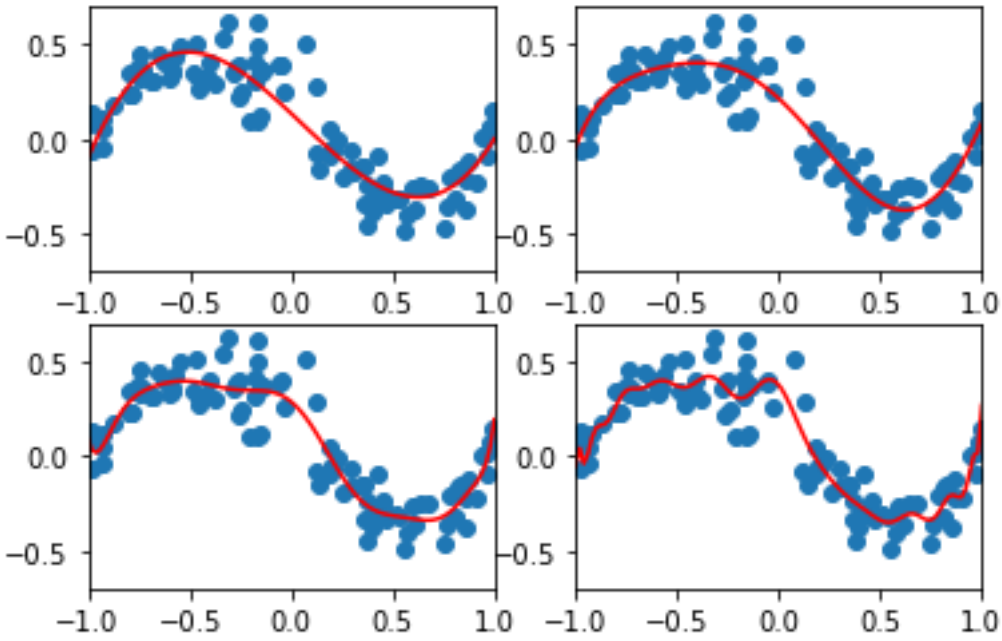
No line would fit the data very well, but a cubic polynomial, $f(x) = c_3x^3 + c_2x^2 + c_1x + c_0$, might. To pursue this, let $\boldsymbol{x}$ and $\boldsymbol{y}$ the set of x- and y-coordinates of this scatter plot, respectively. Let A be the 100×4[ii] Vandermonde matrix associated to $\boldsymbol{x}$. To find a cubic polynomial $f(x)$ to fit this data, we solve the least squares problem $V\boldsymbol{c} = \boldsymbol{y}$, where $\boldsymbol{c}$ is the coefficient vector $\boldsymbol{c} = (c_3, c_2, c_1, c_0)$. Doing so, we get the polynomial

$$f(x) = 1.05812279x^3 - 0.18469187x^2 - 1.00704633x + 0.13590464.$$

The graph of $y = f(x)$ is displayed below in the top left.

We can also use higher degree polynomials to attempt to fit the data better. After all, the higher the degree of the polynomial the more turning points it can have. In the diagram, we compute the polynomial interpolations using degrees 7, 13, and 23, respectively. In Python, the data fitting of a polynomial can be streamlined using the command `np.polyfit(x, y, n)`, where n is the degree of the polynomial to be interpolated. This command will solve the associated least squares problem with coefficient matrix being the associated Vandermonde matrix. Of course, the error is improved for higher polynomials, but not necessarily by much. The RMS errors for each of these polynomial interpolations are, respectively, `0.1199, 0.1097, 0.1049, 0.1016`, respectively.

[ii] Recall that the *degree* of a polynomial $c_nx^n + \ldots + c_2x^2 + c_1x + c_0$ is the highest exponent of a term with nonzero coefficient. When modeling polynomials of degree n, there are $n + 1$ coefficients to find, that is, $c_n, \ldots, c_2, c_1, c_0$. As such the number of columns in the Vandemonde matrix will be $n + 1$.



The following Python code can be used to solve this example:

```
import numpy as np

points = np.array([[ 0.6346537948227042, -0.25053562659185047],
x = points[:,0]
y = points[:,1]

A = np.vander(x,3+1)
f = np.linalg.lstsq(A,y, rcond=None)[0]

#Compute Polynomial Interpolates
degrees = [3,7,13,23]
polys = []
for n in degrees:
    polys.append(np.polyfit(x,y,n))

#Compute RMS error
for g in polys:
    print(np.sqrt(np.mean((y-np.polyval(g,x))**2)))

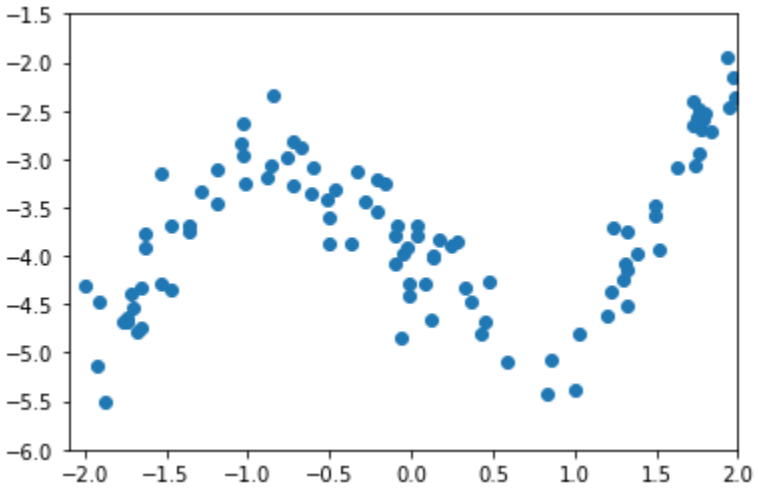
#Graph Scatterplot
import matplotlib.pyplot as plt
plt.scatter(x,y)
plt.axis([-1,1,-0.7,0.7])
plt.show()

#Graph Polynomials
t = np.linspace(-1,1,num = 100)

 #f and polys[0] have the same graph
cubics = [f,polys[0]]
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.axis([-1,1,-0.7,0.7])
    plt.scatter(x,y)
    plt.plot(t, np.poly1d(cubics[i])(t), 'r')
plt.show()

 #Graph polys
for i in range(len(polys)):
    plt.subplot(2,2,i+1)
    plt.axis([-1,1,-0.7,0.7])
    plt.scatter(x,y)
    plt.plot(t, np.poly1d(polys[i])(t), 'r')
plt.show()
```

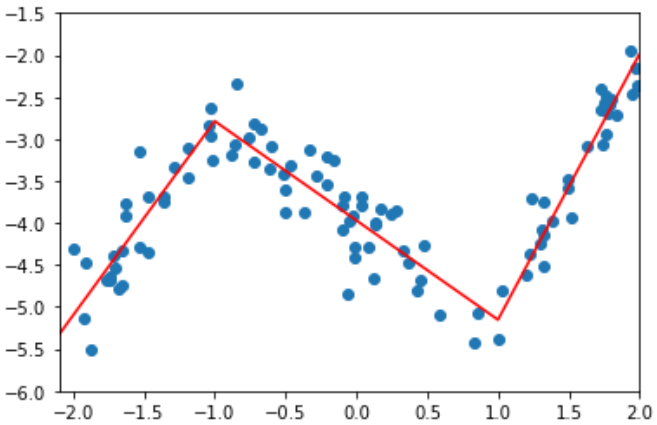
Example 4.2.3. Sometimes the correct model to use is a piecewise linear function.[iii] Consider the scatter plot of 100 points illustrated. There appears to be two turning points on the scatter plot, but otherwise the function model appears to be very straight. This suggests that a piecewise linear model would be a better fit than a polynomial model.

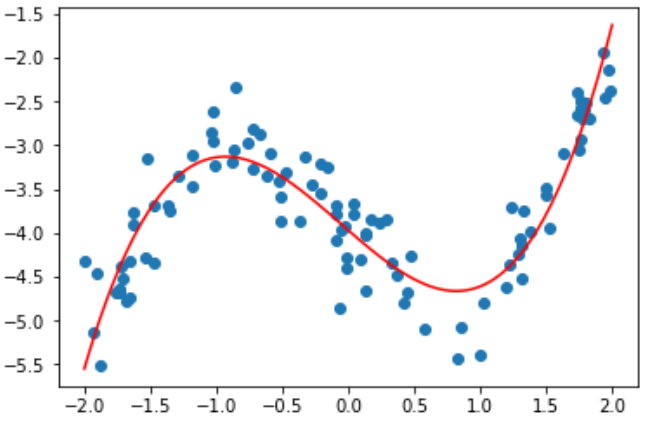


Let $(x)_+ = \max(x, 0)$. Let f be a piecewise linear function with cusps as $x = a_1, a_2, \ldots, a_n$. Then it can be shown that $f(x) \in \text{Span}\{1, x, (x-a_1)_+, (x-a_2)_+, \ldots, (x-a_n)_+\}$. Therefore, if $\boldsymbol{x}$ and $\boldsymbol{y}$ is the x- and y-coordinates of the points in the scatter plot, then the piecewise linear fit of this data would be the least squares solution to the linear system $A\boldsymbol{c} = \boldsymbol{y}$ where

$$A = \begin{bmatrix} 1 & x_1 & (x_1-a_1)_+ & (x_1-a_2)_+ & \ldots & (x_1-a_n)_+ \\ 1 & x_2 & (x_2-a_1)_+ & (x_2-a_2)_+ & \ldots & (x_2-a_n)_+ \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ 1 & x_m & (x_m-a_1)_+ & (x_m-a_2)_+ & \ldots & (x_m-a_n)_+ \end{bmatrix}$$

For this scatter plot, there appears to be two cusps at $a_1 = -1$ and $a_2 = 1$. We display below the piecewise fit and the cubic fit of this scatter plot. The RMS errors of the piecewise and cubic regressions are `0.3213` and `0.3743`, respectively. Thus, we see that the piecewise model was better for this set of data.





The following Python code can be used to solve this example:

```
import numpy as np

points = np.array([[ -1.4683000758467482, -3.6948838397506325],
```

[iii] With our terminology, the more correct term would be piecewise affine.

```
x = points[:,0]
y = points[:,1]

#rows of coefficient matrix associated to piecewise linear fit
piecewise_row = lambda x, cusps : np.concatenate(([1,x], [np.maximum(x-a, 0) for a in cusps
    ]))

#Solve least squares fit
cusps=[-1,1]
A = np.array([piecewise_row(x[i], cusps) for i in range(len(x))])
f = np.linalg.lstsq(A,y,rcond=None)[0]

f3 = np.polyfit(x,y,3)

#RMS Error
print(np.sqrt(np.mean((y-A@f)**2)))
print(np.sqrt(np.mean((y-np.polyval(f3,x))**2)))


#Graph Scatterplot
import matplotlib.pyplot as plt
plt.ion()
plt.axis([-2.1, 2, -6, -1.5])
plt.scatter(x,y)
plt.show()

#Graph piecewise linear and cubic fits
t = np.linspace(-2,2,num = 100)
yhat = np.array([piecewise_row(x,cusps) for x in t])@f

functions = [yhat,np.poly1d(f3)(t)]
for g in functions:
    plt.axis([-2.1, 2, -6, -1.5])
    plt.scatter(x,y)
    plt.plot(t, g, 'r')
    plt.show()
```

Exercises

(Go to Solutions)

For Exercises 1–1, interpolate the function $f(x) = a_1x + a_0$ through the given points.

♠ 1. $(0, 1)$, $(1, 2)$, $(2, 4)$

For Exercises 2–2, interpolate the function $f(x) = a_2x^2 + a_1x + a_0$ through the given points.

♠ 2. $(-1, 1)$, $(0, 0)$, $(1, 2)$, $(2, 5)$

For Exercises 3–3, interpolate the function $f(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ through the given points.

3. $(-2, 231)$, $(-1, 19)$, $(0, 5)$, $(1, 9)$, $(2, 19)$, $(3, -169)$, $(-1.5, 100)$, $(-0.5, 20)$, $(0.5, 12)$, $(2.5, 0)$, $(3.5, -25)$

For Exercises 4–4, interpolate the function $f(x) = \dfrac{a_2x^2 + a_1x + a_0}{b_2x^2 + b_1x + 1}$ through the given points.

4. $\left(-2, \frac{12}{11}\right)$, $\left(-1, \frac{3}{2}\right)$, $(0, 2)$, $(1, 0)$, $(2, 0)$, $\left(-1.5, \frac{10}{3}\right)$, $\left(-0.5, \frac{20}{15}\right)$, $\left(0.5, -\frac{12}{5}\right)$, $\left(2.5, -\frac{11}{3}\right)$, $\left(3.5, -\frac{25}{7}\right)$

5. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/datafit.ipynb

"As long as men are free to ask what they must, free to say what they think, free to think what they will, freedom can never be lost and science can never regress." – Marcel Proust

4.3 Regression

4.3.1 Affine Regression

Of special importance is the affine function of the form $T : \mathbb{R}^n \to \mathbb{R}$. In this case, there exists a vector $\boldsymbol{a} \in \mathbb{R}^n$ and a scalar $b \in \mathbb{R}$ such that $T(\boldsymbol{x}) = \boldsymbol{a} \cdot \boldsymbol{x} + b$, a so-called **affine functional**. Of course, an affine functional is just a linear functional with a possible translation by b. Note that $b = T(\boldsymbol{0})$ and $a_i = T(\boldsymbol{e}_i) - b$, which demonstrates how to construct $\boldsymbol{a}$ and b from T.

In practice, mathematical functions are often used to approximate, estimate, predict, or forecast data that perhaps is not already known. Among other applications, this can inform decision making in the present to help plan for the future or help unravel mysteries of the past. This process of forming a function to approximate data is called *mathematical modeling*. Naturally, linear and affine transformations are quite common in modeling. The **regression model** is exactly that, a model of predicted data using an affine transformation. Suppose that $\boldsymbol{x}$ is an n-vector of known data, called the **regressors**, and y is a quantitative feature of this data which is unknown, called the **outcome**. Let $\widehat{y}$ be our **prediction** (or **estimate**) of y. Then the regression model of $\widehat{y}$ with respect to $\boldsymbol{x}$ would be

$$\widehat{y} = \boldsymbol{a} \cdot \boldsymbol{x} + b = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n + b \tag{4.3.1}$$

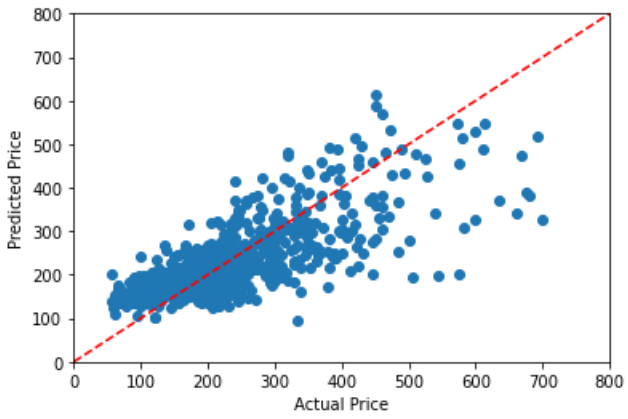
where $\boldsymbol{a}$ represents the **weight vector** (or **coefficient vector**) and b represents the **offset** (or **intercept**). The entries of $\boldsymbol{a}$ then measure how strongly the entries of $\boldsymbol{x}$ affect the outcome y. When $|a_i|$ is large, then x_i has a large affect on y. When $|a_i|$ is small, feature x_i has little affect on the outcome.

Example 4.3.1. The regression model is very interpretable when all of the features are Boolean. As a simple example consider a regression model for the lifespan of a person in some group, with $x_1 = 0$ if the person is female ($x_1 = 1$ if male), $x_2 = 1$ if the person has type II diabetes, and $x_3 = 1$ if the person smokes cigarettes. In this case, b is the regression model estimate for the lifespan of a female non-diabetic nonsmoker; β_1 is the decrease in estimated lifespan if the person is male, β_2 is the decrease in estimated lifespan if the person is diabetic, and β_3 is the increase in estimated lifespan if the person smokes cigarettes. In this example, $\beta_i < 0$. Also, a more accurate model could be created by using more features than just the three listed here. ∎

Example 4.3.2. As another example, let us consider the selling price of a house y, in thousands of dollars (something that seller would really like to predict). The selling price of a home depends on lots of factors, but for this example let us only consider two: x_1 is the area of the home (in hundreds of square feet), and x_2 is the number of bedrooms (number of bathrooms anyone?). A certain realtor, once upon a time, used the following parameters to model the sales of home in Sacramento, CA:

$$\boldsymbol{a} = (147.7251, -18.8534)^\top, \quad b = 54.4017.$$

For example, a 1-bedroom 846 square feet home sold for $y = \$115{,}000$, and the model predicted the sale would be $\widehat{y} = \$161{,}000$. For another sale, a 2-bedroom 1324 square feet home sold for $y = \$234{,}500$, and the model predicted the sale would be $\widehat{y} = \$212{,}000$. The following scatterplot shows the sales of 774 homes in Sacramento, CA and compares them to the prediction from this model. The diagonal line indicates $y = x$, which would be the location of any perfect prediction. Hence, the farther away a point is from the dashed line the worse the prediction.



How does one find this regression model? To find this, we wish to solve the affine equation $\boldsymbol{a} \cdot \boldsymbol{x} + b = y$, which is equivalent to solving the linear equation $(a_1, a_2, b) \cdot (x_1, x_2, 1) = y$. As the data set produces an overdetermined linear equation, we seek the least squares solution. Let $X = \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \mathbf{1} \end{bmatrix}$ be the 774×3 matrix whose first column is the vector of all the areas of the sold homes and $\boldsymbol{x}_2$ is the vector of all the bedroom counts. Thus, we need to solve the least squares problem for $X \begin{bmatrix} a_1 \\ a_2 \\ b \end{bmatrix} = \boldsymbol{y}$, where $\boldsymbol{y}$ is the vector of all the sales prices of these homes. Doing so, we discover the model mentioned above. ∎

Of course, multiple outcomes can be considered simultaneously with regression models. Suppose that outcomes $y_1, y_2, \ldots, y_m$ are to be estimated using the n-vector $\boldsymbol{x}$. Then for each outcome y_i, we seek a prediction $\widehat{y}_i$, which is determined by the regression model

$$\widehat{y}_i = \boldsymbol{a}_i \cdot \boldsymbol{x} + b_i,$$

where $a_i \in \mathbb{R}^n$ and $b_i \in \mathbb{R}$. If we stack these equations on top of each other, we can treat all estimates as a single matrix regression model, namely

$$\widehat{\boldsymbol{y}} = A\boldsymbol{x} + \boldsymbol{b},$$

where $\widehat{\boldsymbol{y}} = (\widehat{y}_1, \widehat{y}_2, \ldots, \widehat{y}_m)^\top$ is the **prediction m-vector**, $A = [\boldsymbol{a}_1\ \boldsymbol{a}_2 \cdots \boldsymbol{a}_m]^\top$ is the $m \times n$ **weight matrix** (or **coefficient matrix**), and $\boldsymbol{b} = (b_1, b_2, \ldots, b_m)^\top$ is the **offset m-vector**. Therefore, the general affine transformation $T(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$ can be viewed as a system of regression models.

Let $\boldsymbol{r} = \boldsymbol{y} - \widehat{\boldsymbol{y}}$ be the **residual vector** (or **error vector**). This vector measures how good of a prediction $\widehat{\boldsymbol{y}}$ was. Now, it is natural to adjust the regression mode $T(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$ by replacing $\boldsymbol{b}$ with $\boldsymbol{b} + \operatorname{avg}(\boldsymbol{r})\mathbf{1}$. This shifts the prediction by a factor of the average error between the prediction and actual data. This will have the affect of de-meaning $\boldsymbol{r}$, and hence is a best practice. When $\operatorname{avg}(\boldsymbol{r}) = 0$, then necessarily $\operatorname{rms}(\boldsymbol{r}) = \operatorname{std}(\boldsymbol{r})$, and hence standard deviation only is sufficient to measure the spread of $\boldsymbol{r}$. Ideally, a better prediction would be when the standard deviation is small. Another useful tool for measuring the efficacy of a regression model is the correlation coefficient. Let

$$\rho = \frac{(\widehat{\boldsymbol{y}} - \operatorname{avg}(\widehat{\boldsymbol{y}})\mathbf{1}) \cdot (\boldsymbol{y} - \operatorname{avg}(\boldsymbol{y})\mathbf{1})}{\|\widehat{\boldsymbol{y}} - \operatorname{avg}(\widehat{\boldsymbol{y}})\mathbf{1}\| \|\boldsymbol{y} - \operatorname{avg}(\boldsymbol{y})\mathbf{1}\|},$$

where of course $\widehat{\boldsymbol{y}} - \operatorname{avg}(\widehat{\boldsymbol{y}})\mathbf{1}$ and $\boldsymbol{y} - \operatorname{avg}(\boldsymbol{y})\mathbf{1}$ are the de-meaned versions of $\widehat{\boldsymbol{y}}$ and $\boldsymbol{y}$, respectively. As good prediction would find $\widehat{\boldsymbol{y}}$ and $\boldsymbol{y}$ highly correlated (ρ close to 1). Note that adjusting the offset $\boldsymbol{b}$ will have no bearing on $\operatorname{std}(\boldsymbol{r})$ nor ρ.

Example 4.3.3. Revisiting Example 4.3.2, we can compute statistics for the residual vector $\boldsymbol{r}$. Namely,

$$\operatorname{avg}(\boldsymbol{r}) = \$0, \qquad \operatorname{rms}(\boldsymbol{r}) = \$74{,}845.72, \qquad \operatorname{std}(\boldsymbol{r}) = \$74{,}845.72.$$

We next compute the Pearson correlation coefficient, namely, $\rho = 0.7480599032761147$, which arguably is an indicator of a good (but not great) regression model.

The following Python code can be used to solve this example:

```
import numpy as np
#initialize data
D = house_sales_data()
y = D["price"]
x = np.column_stack((D["area"], D["beds"], np.ones(len(y))))

#preform least squares on the data
a_hat = np.linalg.lstsq(x,y,rcond=None)[0]
print(a_hat)

# parameters in regression model
a = a_hat[:-1] #np.array([148.7251, -18.8534])
```

```
b = a_hat[-1] #54.4017

y_hat = lambda x: a @ x + b

#Evaluate regression model prediction
x1 = np.array([0.846, 1])
y1 = 115
print("y_hat = $" + str(int(round(y_hat(x1),0))*1000))
print("y = $" + str(int(y1*1000)), "\n")

# do it again, but with another sample house
x2 = np.array([1.324, 2])
y2 = 234.50
print("y_hat = $" + str(int(round(y_hat(x2),0))*1000))
print("y = $" + str(int(y1*1000)), "\n")

predicted = a_hat @ x.T

r = D['price'] - predicted                    # vector of predicted errors
print("average = $" + str(round(np.average(r)*1000,2)))
print("root mean square = $" + str(round(np.sqrt(sum(r**2)/len(r))*1000,2)))
print("standard deviation = $" + str(round(np.std(r)*1000,2)), "\n")

de_predicted = predicted-np.average(predicted)
de_y =  D['price'] - np.average(D['price'])
print("correlation = " + str(np.inner(de_predicted, de_y )
                             /np.linalg.norm(de_predicted)/np.linalg.norm(de_y)), "\n")

# Graphing
import matplotlib.pyplot as plt
plt.scatter(D['price'], predicted, color='b')
plt.plot((0,800),(0,800) ,ls='--', c = 'r')
plt.ylim(0,800)
plt.xlim(0,800)
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.show()
```

4.3.2 Auto-regressive Time Series

Definition 4.3.4. Let $\boldsymbol{t} = (t_1, t_2, \ldots)$ be a time series. We say that $\boldsymbol{t}$ is **auto-regressive**[i] if the next term in the times series is approximated by the ℓ of its predecessors, that is,

$$\widehat{t_{t+1}} = c_1 t_t + c_2 t_{t+1} + \ldots + c_\ell t_{t-\ell+1}.$$

The number of terms ℓ is called the **lag** (or **memory**) of the time series.

An auto-regressive time series with lag ℓ can be computed using the least squares approach. Suppose we have already collected the first T entries of $\boldsymbol{t}$, that is, $t_1, t_2, \ldots, t_T$ is known, for $T > \ell$. Then for $\ell \leq i < T$ we find the equation

$$t_{i+1} = c_1 t_i + c_2 t_{i-1} + \ldots + c_\ell t_{i-\ell+1}.$$

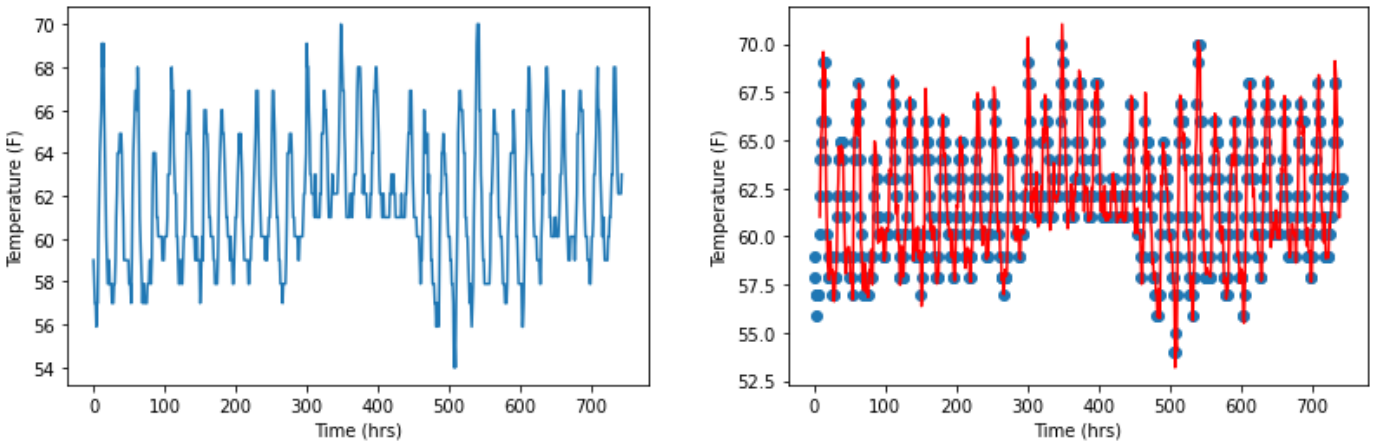
Then

$$\boldsymbol{t}_{[\ell+1,T]} = \begin{bmatrix} t_{\ell+1} \\ t_{\ell+2} \\ t_{\ell+3} \\ \vdots \\ t_i \\ \vdots \\ t_T \end{bmatrix} = \begin{bmatrix} t_\ell & t_{\ell-1} & t_{\ell-2} & \cdots & t_1 \\ t_{\ell+1} & t_\ell & t_{\ell-1} & \cdots & t_2 \\ t_{\ell+2} & t_{\ell+1} & t_\ell & \cdots & t_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_{i-1} & t_{i-2} & t_{i-3} & \cdots & t_{i-\ell} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_{T-1} & t_{T-2} & t_{T-3} & \cdots & t_{T-\ell} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_\ell \end{bmatrix}$$

[i] We note that an auto-regressive time series is a special type of linear dynamical system, where $n = 1$, that is, each vector in the sequence is a scalar, and $A_i = c_i$.

If $T \geq 2\ell$, then we have enough equations to solve the least squares problem and find the coefficient of the auto-regression.

Example 4.3.5. The following graph shows the hourly temperatures of the Los Angeles International Airport on 1-31 May 2016 at 12:53 AM to 11:53PM, illustrated below left



Temperature levels are generally auto-regressive, that is, the next hours temperature is computed using the previous temperatures using a 24-hour lag. For this vector, there are $24(31) = 744$ entries. Thus, $\ell = 24$ and $T = 744 > 48 = 2\ell$, which implies that an auto-regressive model can be constructed using the least squares method mentioned above. Doing so produces the auto-regression illustrated above right.

The following Python code can be used to solve this example:

```
import numpy as np

# Hourly temperature at LAX in May 2016 in a vector of length 31*24 = 744.  Measurements are
    hourly at 12:53 AM to 11:53PM.
t = np.array([
T = len(t)
l = 8 #lag
y = t[l:] #first period removed, values to predict
A = np.array([t[i:i+T-l] for i in reversed(range(l))]).T
print(A)
coeffs = np.linalg.lstsq(A,y, rcond=None)[0]
y_hat = A @ coeffs

#Graph Time Series
import matplotlib.pyplot as plt
plt.plot(range(24*31), t)
plt.xlabel('Time (hrs)')
plt.ylabel('Temperature (F)')
plt.show()

plt.scatter(range(24*31), t)
plt.plot(range(l,24*31), y_hat[:24*31-l], 'r')
plt.xlabel('Time (hrs)')
plt.ylabel('Temperature (F)')
plt.show()

plt.scatter(range(24*7), t[:24*7])
plt.plot(range(l,24*7), y_hat[:24*7-l], 'r')
plt.xlabel('Time (hrs)')
plt.ylabel('Temperature (F)')
plt.show()
```

■

4.3.3 Periodic Regression

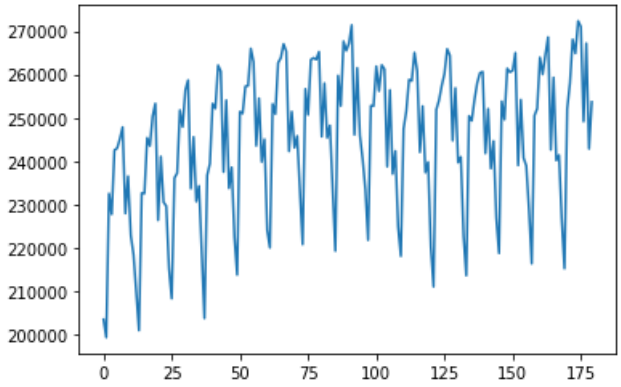
Definition 4.3.6. A function $f : \mathbb{R} \to \mathbb{R}$ is **periodic** if there exists some positive integer P such that $f(x+P) = f(x)$ for all $x \in \mathbb{R}$. The **period** p of f is the smallest positive integer p such that $f(x+p) = f(x)$.

Example 4.3.7. The functions $y = \sin(x)$ and $y = \cos(x)$ are both periodic with period $p = 2\pi$. The function $y = \tan(x)$ has a period of $p = \pi$. ■

Let us now consider a regression model on a time series $\boldsymbol{t}$ that deals with a periodic behavior, that is, it repeats itself every p units of time. Such a time series $\boldsymbol{t}$ is illustrated to the right. If $P = \begin{bmatrix} I_p \\ I_p \\ \vdots \\ I_p \end{bmatrix}$, then $\boldsymbol{t} = P\boldsymbol{t}_{1:p}$. If $\boldsymbol{s}$ is a time series that is approximately periodic, then a least squares solution to $P\boldsymbol{x} = \boldsymbol{s}$ provides an approximation to the periodicity of $\boldsymbol{s}$ and can be used to forecast the time series forward or backward.

$$\boldsymbol{t} = (t_1, t_2, \ldots, t_p, t_1, t_2, \ldots, t_p, \ldots, t_1, t_2, \ldots, t_p) = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_p \end{bmatrix}$$

Example 4.3.8. The following linear graph illustrates the number of miles driven by US drivers per month from 2000-2014.[ii]



One can quickly determine a periodic behavior to the graph, that is, driving habits in summertime are similar year-after-year. Similarly, wintertime driving is also very similar, although the amount of miles is slowing progressing each year certainly due to an increased number of drivers year-after-year. The upward progression suggests to us that there should be some linear component to the regression and the seasonal behavior suggests a periodic times series. We are actually able to combine these two ideas.

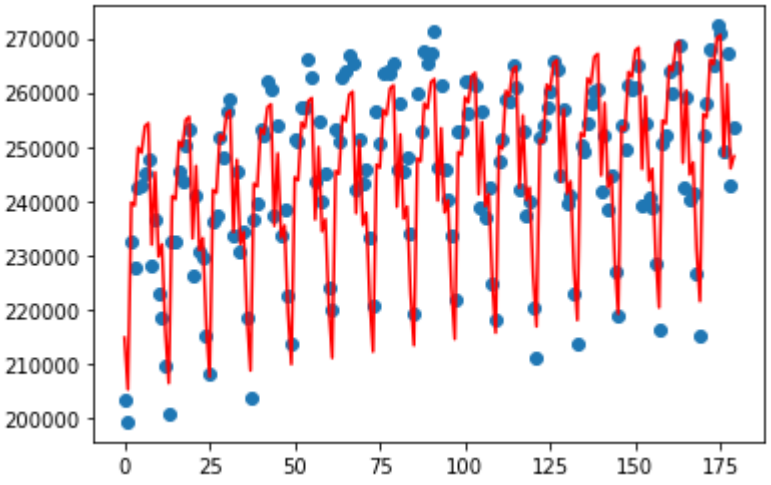
Let $\boldsymbol{t}$ be the time series for the monthly mileage for US drivers from 2000-2014. Then $\widehat{\boldsymbol{t}}$ will be our approximation of $\boldsymbol{t}$. Note that $\widehat{\boldsymbol{t}} = \widehat{\boldsymbol{t}}_{\text{linear}} + \widehat{\boldsymbol{t}}_{\text{periodic}}$. Let $\boldsymbol{c}$ be the coefficient vector associated to this regression model. Then we must find the least squares solution to the system illustrated below right.

[ii] U.S. Department of Transportation, Bureau of Transportation Statistics, www.transtats.bts.gov.

Of course, we can note that the first column of this matrix is the sum of the last p columns and therefore is not necessary for solving the problem (we prefer a matrix with linear independent columns). Instead, we seek to find the least squares solution to the system illustrated below left. Solving this least squares problem, we obtain the following periodic regression to the vehicle mileage, as illustrated below right.

$$\begin{bmatrix} 1 & 1 & 1 & 0 & \dots & 0 \\ 1 & 2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & p & 0 & 0 & \dots & 1 \\ 1 & p+1 & 1 & 0 & \dots & 0 \\ 1 & p+2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2p & 0 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (N-1)p+1 & 1 & 0 & \dots & 0 \\ 1 & (N-1)p+2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & Np & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{p+1} \end{bmatrix} = \boldsymbol{t}.$$

$$\begin{bmatrix} 1 & 1 & 0 & \dots & 0 \\ 2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p & 0 & 0 & \dots & 1 \\ p+1 & 1 & 0 & \dots & 0 \\ p+2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 2p & 0 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (N-1)p+1 & 1 & 0 & \dots & 0 \\ (N-1)p+2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Np & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{p+1} \end{bmatrix} = \boldsymbol{t}.$$



The following Python code can be used to solve this example:

```
import numpy as np

# Vehicle Miles Traveled (VMT) (Millions)
# US Dept of Transportation, Bureau of Transportation Statistics
# www.transtats.bts.gov
# 15 x 12 matrix: monthly value for 15 years 2000-2014
vmt = np.array([

#Solve Regression Model
m = 15*12
A = np.column_stack((range(m), np.vstack([np.identity(12) for i in range(15)])))
b = np.concatenate(vmt)
x = np.linalg.lstsq(A,b, rcond=None)[0]

#Graph Data
import matplotlib.pyplot as plt
plt.plot(range(m), b)
plt.show()

#Graph Periodic Regression
```

```
plt.scatter(range(m), b)
plt.plot(range(m), A @ x, 'r')
plt.show()
```

Exercises

(Go to Solutions)

For Exercises 1–1, use the data set below to construct the given regression model. Find the Pearson coefficient between the predicted and actual data.

♠ 1. $(-1,1)$, $(0,0)$, $(1,2)$, $(2,5)$; $f(x) = a_2x^2 + a_1x + a_0$

For Exercises 2–2, use the data set from Example 4.3.2 to construct a regression model as given. Find the Pearson coefficient between the predicted and actual data. *A calculator or computer is recommended.*

2. $y = a_1x_1^2 + a_2x_2^2 + a_3x_1x_2 + a_4x_1 + a_5x_2 + b$

3. A golf coach is helping her player analyze how clubhead speed at impact affects the distance of the golf ball. The player hits several shots and the coach records the following data (to the right). The coach wants to create a linear model $y = a_1x + a_0$ that can predict the ball's distance y (in yards) using the clubhead speed x (in mph). Compute this regression model and use it to predict the distance of the ball when the clubhead speed is 95 mph.

Clubhead Speed (mph)	Golf Ball Distance (yards)
80	180
90	205
100	230
110	250

4. A teacher collected data on how many hours students studied and their test scores (to the right). The teacher wants to find the best-fit line $y = a_1x + a_0$ to predict scores based on study time. Compute this regression model and use it to predict the test score of a student who studies for 4 hours.

Hours Studied	Test Scores
0.5	52
1	58
1.5	67
2	71
2.5	78

5. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/regression.ipynb

"Before you look for validation in others, try and find it in yourself." – Greg Behrendt

4.4 Validation

In the previous sections, we have seen examples of models being used to fit tightly to data and how these models can be used to predict future data. The major problem we have is that while we can measure how effectively the model fits the data in hand how can we measure how well the model predict new data? After all, if we had the new data already we could measure the error between it and the prediction, but if we had the new data why would we need a prediction? It is somewhat of a problem. The ***generalization*** *of a model is its ability to react properly to new data. When making a model, it is important that it has good generalization. Surprisingly, a model with relatively small RMS error can have worse generalization than another model with a worse RMS error. We will attempt to explain this phenomenon.*

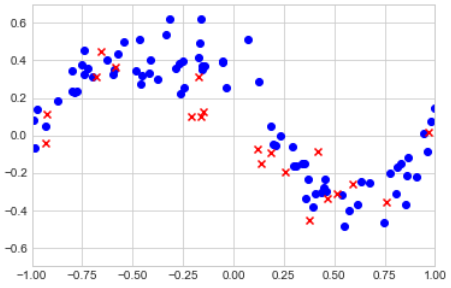
4.4.1 Out-of-Sample Validation

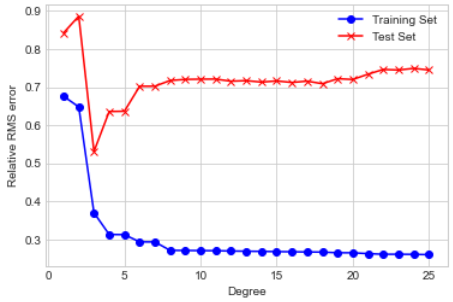
Validation is a strategy of using some of the observed data to build the model, called the **training set**, and using the other data to verify the quality of the model, called the **test set** (or **validation set**). If the prediction error on the test set is comparable to the training set then we can assume that the model has good generalization. If the prediction error is worse on the test set than on the training set, then we can conclude that the model's generalization is poor. If new data is not significantly different from the already observed data, then validation can be effective in predicting the generalization of the model. Of course, predictions that depend of current trends that will evolve into new trends in the future indicate to us that even validation methods have their limitations. Even still, validation can be a valuable tool in data fitting computations.

When analyzing the prediction error of a model $g(x) = \sum_{i=1}^{n} c_i f_i(x)$, this error never worsens when another function $f_{n+1}(x)$ is added to the spanning set of functions $\{f_1, f_2, \ldots, f_n\}$. After all, either the new function f_{n+1} offers no new benefit to modeling the data, which would imply that $c_{n+1} = 0$ and the extended model of g is the same as if only the first n functions were used. More likely though, $c_{n+1} \neq 0$ and to some degree $f_{n+1}(x)$ improves the prediction of data. Therefore, it is tempting to keep on adding more and more functions into the spanning set of g to further improve the prediction error of g on the data. For example, if one is modeling data using polynomial interpolation, then one can continue to increase of the degree of the polynomial to reduce the prediction error and raise the correlation. We want to avoid this temptation because of an anomaly known as *over-fitting*. **Under-fitting** is when too few functions are used to model data causing the prediction error of the model to be very poor, that is, the model is too simple. Conversely, **over-fitting** is when too many functions are used to model data causing generalization of the model to be very poor, that is, the model is too complicated. Finding a *balanced* model requires minimizing prediction error while still maximizing generalization. We describe two validation methods that do exactly this.

Out-of-Sample Validation is a validation method which places 80-90% of the data randomly into the training set and the remaining 10-20% data into the test set. The regression model is made by using only the training set. Then using the test set, which the model has "never seen," we compute the prediction error and compare it to the prediction error of the training set.

Example 4.4.1. Consider the 100-point data set that we considered in Example 4.2.2. This data is illustrated below where now the training and test sets are indicated and were randomly selected. We previously modeled this set using polynomial regression and we observed that the higher the degree of the polynomial the better the fit to the data. Unfortunately, these higher polynomials over-fit the data. The second diagram illustrates both the prediction error of the model on the training and test sets. We see that the prediction error only goes down when higher powers are added to the model. On the other hand, the prediction error on the test set dropped initially until $n = 3$ where it begins to raise. This indicates to us that for this data set a cubic polynomial is probably the most generalizable model, that is, the cubic model has the best chance of predicting new data. Those polynomials of degree less than 3 were under-fit to the data while those polynomials of degree higher than 3 were over-fit. Thus, a cubic polynomial is really the only balanced model, at least indicated by this test set.





The following Python code can be used to solve this example:

```
import numpy as np
import matplotlib.pyplot as plt

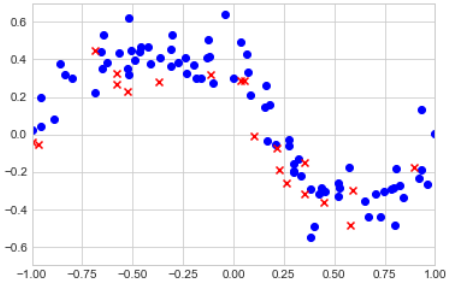
training_set = np.array([[ 0.6346537948227042, -0.25053562659185047],
test_set = np.array([[ 0.3751545125444873, -0.4535170295993656],
x = training_set[:,0]
y = training_set[:,1]
x_test = test_set[:,0]
y_test = test_set[:,1]

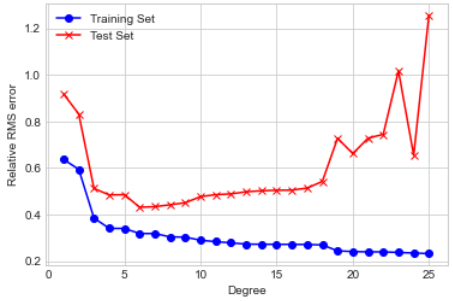
plt.ion()
plt.scatter(x,y, color='b', marker='o')
plt.scatter(x_test,y_test, color='r', marker='x')
plt.axis([-1,1,-0.7,0.7])
plt.show()

d = 25
training_error = np.zeros(d)
test_error = np.zeros(d)
for n in range(1,d+1):
    f = np.polyfit(x,y,n)
    training_error[n-1] = np.linalg.norm(np.poly1d(f)(x) - y)/np.linalg.norm(y)
    test_error[n-1] = np.linalg.norm(np.poly1d(f)(x_test) - y_test)/np.linalg.norm(y_test)

print(test_error[2], "\n")
plt.ion()
plt.plot(range(1,d+1), training_error, 'b-o', label = 'Train')
plt.plot(range(1,d+1), test_error,  'r-x', label = 'Test')
plt.xlabel('Degree')
plt.ylabel('Relative RMS error')
plt.legend(['Training Set', 'Test Set'])
plt.show()
```

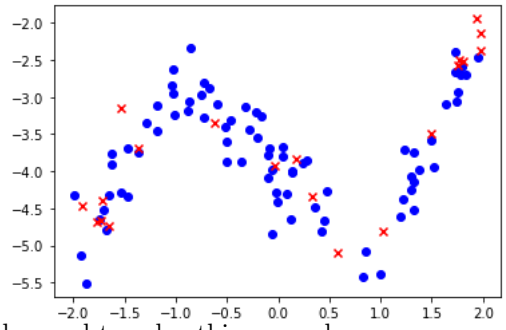
Example 4.4.2. Next, we generated a new set of 100 points that resembles the set from the previous example, illustrated below with the training and test sets indicated. If we attempt to model this data via polynomials like the previous one, we see a similar analysis. Having higher and higher degrees improves the prediction error on the training set, but worsens the ability to generalize. For this data, there appears to be a much larger range of balanced models, but the sextic polynomial ($n = 6$) appears to have the best generalization as it has the lowest prediction error on the test set. We also note that the relative RMS error of the sextic model on this data set is lower than the relative RMS error of the cubic polynomial from the previous data set, approximately 43% versus 53%. Therefore, we have higher confidence that this sextic model will generalize better than the previous cubic model.





The linked Python code can be used to solve this example. Note that it is identical to the previous example except for the data set. ■

Example 4.4.3. Consider the 100-point data set that we considered in Example 4.2.3. This data is illustrated below where now the training and test sets are indicated and where randomly selected. We previously modeled this set using piecewise linear and cubic regressions, and we observed that the piecewise regression appeared to have a better fit. Frankly, both of these models fit very well. The relative RMS error for the piecewise linear regression is 8.5% and 8.2% on the training and test sets, respectively. The relative RMS error for the cubic regression is 9.8% and 9.8% on the training and test sets, respectively. Therefore, both models have great generalization, but the piecewise linear regression did fit the data better.



The following Python code can be used to solve this example:

```
import numpy as np
import matplotlib.pyplot as plt

training_set = np.array([[ -1.4683000758467482, -3.6948838397506325],
test_set = np.array([[ 0.16924755009871761, -3.837131660800567],
x = training_set[:,0]
y = training_set[:,1]
x_test = test_set[:,0]
y_test = test_set[:,1]

plt.ion()
plt.scatter(x,y, color='b', marker='o')
plt.scatter(x_test,y_test, color='r', marker='x')
plt.show()

#least square fitting
piecewise = lambda x, cusps : np.concatenate(([1,x], [np.maximum(x-a, 0) for a in cusps]))
cusps=[-1,1]
A = np.array([piecewise(x[i], cusps) for i in range(len(x))])
f = np.linalg.lstsq(A,y,rcond=None)[0]
print(f)

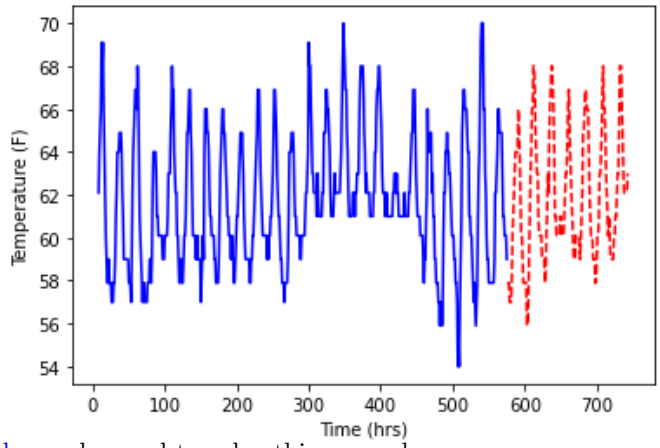
print("relative training error (piecewise) =", np.linalg.norm(f[0] + f[1]*x + f[2]*np.
    maximum(x+1,0) + f[3]*np.maximum(x-1,0)-y)/np.linalg.norm(y))
print("relative test error (piecewise) =", np.linalg.norm(f[0] + f[1]*x_test + f[2]*np.
    maximum(x_test+1,0) + f[3]*np.maximum(x_test-1,0)-y_test)/np.linalg.norm(y_test), "\n")

f3 = np.polyfit(x,y,3)
print("relative training error (cubic) =", np.linalg.norm(np.poly1d(f3)(x)-y)/np.linalg.norm
    (y))
print("relative test error (cubic) =", np.linalg.norm(np.poly1d(f3)(x_test)-y_test)/np.
    linalg.norm(y_test), "\n")
```
■

While it is generally best to select a random subset of the data to form the test set, this can lead to bias in the case of time series. Unlike an arbitrary data set, the elements of a time series are sequential and must be considered in that correct order. If a random subset is selected from the time series to form the training set, then likely elements will be chosen from all across the time period, including points near the end of time series. New data added to the time series will necessarily be added at the end of the time series (or potentially at the beginning of the time series). Therefore, building a model using data across the whole series will give the model some sense of the future and improve its ability to predict values near the end of the

series but artificially inflate any perception that it is generalizable. As such, when running an out-of-sample validation on a time series model, it is best to select the end of the time series to be the test set. Removing the end-of-series data will prevent the training set from "knowing" this simulated future. In this way, any good prediction on the test set would suggest that the model generalizes well to actual, unknown data.

Example 4.4.4. Consider again the hourly temperature (in Fahrenheit) measured as the Los Angeles International Airport during May 2016, seen in Example 4.3.5. We originally used the least squares method to compute an auto-regression of the time series with a lag of $\ell = 8$ hours. Using the out-of-sample validation method, we select the end portion of the time series for our test set. As the data covers a whole month, we will select the last week of May as our test set, that is, 25-31 May. This leaves 1-24 May for the test set. The auto-regression is computed the same as before but now only using the first three weeks, which obtains the coefficients of the auto-regression. Using these coefficients, the last week of temperatures can be computed recursively. The relative RMS error for the auto-regression on the training set is 1.66%, and the relative RMS error on the test set is 1.57%. Therefore, a hourly temperature time series is effectively modeled using an auto-regression.



The following Python code can be used to solve this example:

```
import numpy as np
import matplotlib.pyplot as plt

# Hourly temperature at LAX in May 2016 in a vector of length 31*24 = 744.  Measurements are hourly at 12:53 AM to 11:53PM.
t = np.array([
l = 8 #lag
end = 576 #= 24 days x 24 hours

y = t[l:end]
y_test = t[end:]
A = np.column_stack([t[i:i-l+end] for i in reversed(range(l))])
coeffs = np.linalg.lstsq(A,y, rcond=None)[0]
A_test = np.column_stack([t[end-l+i:i-l+len(t)] for i in reversed(range(l))])

print("relative training error (time series) =", np.linalg.norm(A @ coeffs -y)/np.linalg.norm(y))
print("relative test error (time series) =", np.linalg.norm(A_test @ coeffs-y_test)/np.linalg.norm(y_test), "\n")

plt.ion()
plt.plot(range(l,end), y, color='b')
plt.plot(range(end, len(t)), y_test, color='r', linestyle ='--')
plt.xlabel('Time (hrs)')
plt.ylabel('Temperature (F)')
plt.show()
```

■

4.4.2 Cross-Validation

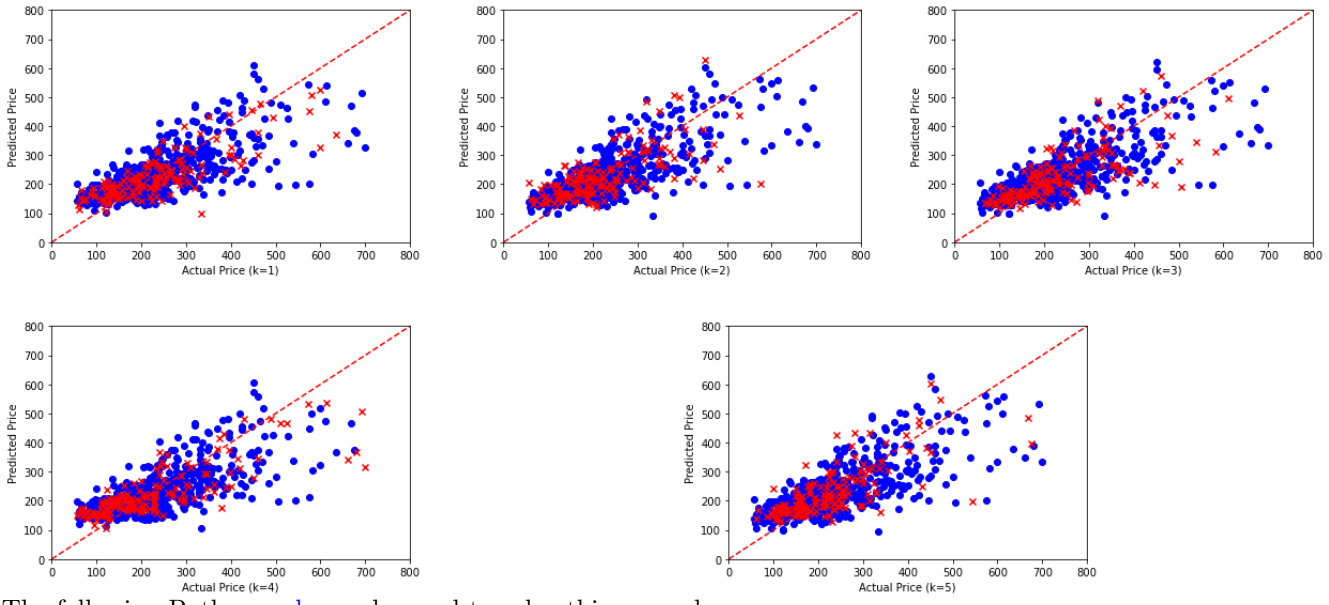
The main drawback of out-of-sample validation is that with only one random sample we might get the false assumption of generalization (or lack of it). **Cross-validation** is a stronger validation method that employs

multiple randomization. First, a random partition is assigned to the data X with N different groups, called **folds**, of approximately the same size. Typically, $N = 5$ or 10. For each fold F, we run an out-of-sample validation where F is the test set and $X \setminus F$ is the training set. Thus, there will be N models with N prediction errors at the end of the process. Good generalization will occur when all the models created are approximately the same and each of the prediction errors are likewise approximately the same. If the model is determined accurate and generalizable, then one of the similar models is selected (likely the best performer) or an average of all the models is used instead.

Example 4.4.5. Consider the regression model for home prices in San Francisco. We will run cross-validation against the data set to verify the quality of our regression model. The data is randomly put into 5 folds of approximately 154 points each. The following table shows the coefficients of the model $\widehat{y} = a_1x_1 + a_2x_2 + b$ constructed using fold k as the test set and its associated prediction errors.

k	a_1	a_2	b	Training Error	Test Error
1	145.18603792	-18.92044941	59.19140685	0.2995637	0.26981602
2	150.83141088	-15.98823261	43.02502977	0.28923148	0.31347337
3	149.11722641	-15.80113893	42.75642007	0.29043393	0.30586853
4	145.37314336	-24.62709541	78.3829534	0.29109469	0.30672883
5	152.40344709	-18.75549071	49.22418284	0.2948938	0.28810665

We see that each model is quite comparable to the first model we construct for this data. Likewise, the prediction errors are each from 29%. The consistency we see from this cross-validation supports our assumption that this model will remain with about 29% prediction error on any new housing data.



The following Python code can be used to solve this example:

```
import numpy as np
import matplotlib.pyplot as plt

#cross validate the linear system Xc = y
def cross_validate(X,y,folds=5,display=False,label="Data",lower_x=0,upper_x=10,lower_y=0,
    upper_y=10):
    plt.ion()
    m,n = X.shape
    partition = np.array_split(np.random.permutation(m),folds)  #randomly splits the data
    into (roughly) equal number of folds and represents this partition as a set of set of
    indices. The default number of folds is 5.
```

```
    c = np.zeros((folds,n)) #initialize the regression models. c[k] is the coefficient vector for the model generated when partition[k] acts as test set
    training_error = np.zeros(folds) #initialize the training errors. training_error[k] is the training error for the model c[k].
    test_error = np.zeros(folds) #initialize the test errors. test_error[k] is the test error for the model c[k].
    for k in range(folds):
        test_index = partition[k]                    #set of indices for current test set
        training_index = np.concatenate(partition[:k]+partition[k+1:]) #set of indices for current training set
        a = np.linalg.lstsq(X[training_index],y[training_index],rcond=None)[0]
        c[k] = a                                    #create model c[k] using least squares
        training_error[k] = np.linalg.norm(X[training_index] @ a - y[training_index])/np.linalg.norm(y[training_index]) #compute training error of c[k]
        test_error[k] = np.linalg.norm(X[test_index] @ a - y[test_index])/np.linalg.norm(y[test_index]) #compute test error of c[k]
        if display:              #display visualization of cross-validation if requested
            plt.scatter(y[training_index], X[training_index] @ a, color='b', marker='o')
            plt.scatter(y[test_index], X[test_index] @ a, color='r', marker='x')
            plt.plot((lower_x,upper_x),(lower_y,upper_y) ,ls='--', c = 'r')
            plt.ylim(lower_y,upper_y)
            plt.xlim(lower_x,upper_x)
            plt.xlabel(f"Actual {label} (k= {str(k+1)})")
            plt.ylabel(f"Predicted {label}")
            plt.show()
    return c,training_error,test_error
D = house_sales_data()
y = D['price']
X = np.column_stack([D['area'],D['beds'],np.ones(len(y))])

C, train, test = cross_validate(X,y,display=True,label="Price",upper_x=800,upper_y=800)
print(C, "\n\n", np.column_stack((train, test)))
```

Exercises

(Go to Solutions)

For Exercises 1–1, use the data set from Example 4.3.2 to cross-validate a regression model as given. Find the average prediction error with 5 folds. *A calculator or computer is recommended.*

1. $y = a_1x_1^2 + a_2x_2^2 + a_3x_1x_2 + a_4x_1 + a_5x_2 + b$

2. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/valid.ipynb

"One new feature or fresh take can change everything." – Neil Young

4.5 Feature Engineering

4.5.1 Linear Features

Data is simply recorded observations, but not all the data recorded from an observation is necessarily meaningful though. For example, facial recognition software will look at photos and these pictures may have lots of attributes such as furniture, animals, floral patterns, just to name a few possibilities. But the only portion of a picture that is of any relevence to the software is a human face. This meaningful, relevant attribute in the data is called a **feature**. For facial recognition software to be effective, it has to be able to identify this feature among all the attributes of the observation. The only pixels that matter, in this case, are those pixels belonging to faces.

Features become meaningful to a mathematical model $g(x_1, \ldots, x_n) = \sum_{j=1}^{n} f_j(x_1, \ldots, x_n)$ based upon the functions f_i in play. For a fixed list of attributes $x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$, if $f_j(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)$ remaining contants (or approximately constant) as x_i varies, then the attribute x_i is not a feature of f_j. Likewise, if x_i is negligible to all the functions f_j, then x_i will negligible to g too. Thus, the functions f_j we choose to build the model g from are of the utmost importance. Adjusting the functions f_j in the model g to capture specific attributes from the data set is known as **feature engineering**.

The term "engineering" here suggests that the practice is as much an art form as it is a science (if not more), as good engineering occurs when creativity and inspiration are coupled with sound scientific, logical, and quantitative reasoning. Thus, there is no algorithm available to tell someone how to effectively engineer features, but we can still list in this section some common best practices for feature engineering while also alluding to some more advanced techniques.

Keep your model simple. Adding new functions to the model can lead to over-fitting. Often, fewer features may be even better. If fewer features are adequate for the prediction, a simpler model is preferred.

Make your model be at least affine. It is a good practice to include in your list of functions $f(x_1, \ldots, x_n) = x_i$, that is, the function which reports attribute x_i. Likewise, the constant function $f(x_1, \ldots, x_n) = 1$ is wise to include. If we do not include any other base functions in the model, then g will be exactly the affine regression model we introduced before. This allows for each attribute x_i to have some potential to affect the prediction $\widehat{y}$. Of course, in the least squares solution the coefficient of x_i may be zero (or approximately zero), which indicates that x_i is a poor predictor of y. It should probably be omitted from the basis of g in that case.

It is a good practice to compute the Pearson correlation coefficient ρ_i between the dependent variable y and the potential predictor x_i. If ρ_i is far from 0, then x_i likely should be a feature of the model. If ρ_i is close to 0, then x_i is likely an atrribute to be omitted. Be careful though. The Pearson coefficient measures how closely there is a linear relationship between y and x_i. If there is a nonlinear relationship between y and x_i, then ρ_i may not measure this well. Of course, we need to consider the possibility of nonlinear features too, but we will address this below.

Example 4.5.1. In Example 4.3.2, we considered the housing data from Sacramento of 774 home sales. The original model used only the features x_1 measuring area and x_2 measuring the number of bedrooms. The data set has other attributes too, such as x_3 which is the number of bathrooms, x_4 which is a boolean characterizing if the home is a condo or not, and x_5 which indicates the location of the home by ZIP code. We calculate the Pearson correlation coefficient ρ_i for each of these attributes, as illustrated below:

$$\rho_1 = 0.7418072776132951, \rho_2 = 0.45812908393973756, \rho_3 = 0.5213758700245009,$$
$$\rho_4 = -0.18135729526643038, \rho_5 = 0.1532960368118083$$

We see that x_1 and y have a very strong correlation at 74.2%. Certainly, square footage contributes to the price of a sale and x_1 should be part of the model. The number of bedrooms and the number of bathrooms appear to have a decent correlation on the sales prices, at 45.8% and 52.1%, respectively, although the correlation is not as strong as x_1. Lastly, x_4 and x_5 have very little correlation with y, at -18.1% and 15.3%, respectively. As such, we will omit x_4 and x_5 from out model.

We will update our model for predicting y to be

$$\widehat{y} = a_1x_1 + a_2x_2 + a_3x_3 + b = \boldsymbol{a} \cdot \boldsymbol{x} + b.$$

Running cross-validation on this updated model gives the following analysis:

k	a_1	a_2	a_3	b	Training Error	Test Error
1	149.14976263	-20.06167042	3.52500147	49.98032155	0.29429803	0.29069453
2	146.79534495	-18.17443456	-3.75956933	61.25857165	0.28358244	0.32682492
3	148.13534402	-18.58382705	0.28979069	54.84480212	0.30257374	0.25704957
4	150.25562178	-20.08621302	-1.09045356	57.94051249	0.29134644	0.30221601
5	151.04077182	-16.78872085	-2.37554322	49.97373785	0.29437691	0.28978581

We note that the coefficients a_1 and b made hardly no change, and the prediction error likewise has no significant improvement. The coefficient of x_3 is relatively small and appears to have been siphoned off from a_2. Why was there no improvement by adding an attribute with 54% correlation?

It actually begs to reason that the number of bedrooms and bathrooms are correlated. After all, the more people living in the same home will benefit from an extra bathroom. A quick check shows that $\rho = 64.2\%$. So it appears that the contribution of x_3 on y was already felt from x_2. Thus, for the sake of simplicity, we elect to not incorporate x_3 as a feature of our model.

Of course, it also seems quite reasonable that the number of bedrooms is also correlated to square footage. After all, an extra room adds more area to the home. The correlation coefficient here is $\rho = 70.9\%$. Thus, it seems reasonable that the feature x_1 is already accounting for the contribution of x_2 toward y (which might account for the change negative coefficent attached to x_2). Cross-validation gives the following analysis:

k	a_1	b	Training Error	Test Error
1	131.27174687	22.04230353	0.29705149	0.29453812
2	133.51133127	16.37652548	0.28217656	0.34462785
3	128.72205099	23.03196748	0.30274665	0.27459174
4	128.89742073	23.57078658	0.29973978	0.28409358
5	134.94442009	18.10311033	0.29958309	0.28302047

Thus, we see that the attribute x_1 alone can maintain the same level of accuracy and generalization as including x_2 and x_3. Therefore, x_1 is the only feature to predicting y found in our data set. ∎

So far, we have only considered the raw data. Sometimes it is important to **transform the data** to discover features. This could involving standarding the data, using the function $f(x_1, \ldots, x_n) = z_i = \dfrac{x_i - \text{avg}(\boldsymbol{x}_i)\mathbf{1}}{\text{std}(\boldsymbol{x}_i)}$, where x_i is the variable attribute and $\boldsymbol{x}_i$ is the vector listing all the ith attributes. Standardizing data into z-scores is often important to eliminate bias between two attributes that may exist in very different ranges. Since $z_i \in \text{Span}\{1, x_i\}$, standardization is not needed for a purely affine model. When

nonlinear features are used however, it might be necessary.

Another type of data transformation that may be appropriate is clipping or so-called *winsorizing*[i]. This could involve restricting the data set down to a smaller, more feasible window or the removal of outliers and other irregular points.

Lastly, even with a standardized and pruned data set, it might be appropriate to change the scale of the data. For example, the magnitude of a single earthquake can easily be million of times larger than another earthquake. Thus, the Ritcher scale employs logarithms to shrink the range of possible earthquake intensities so that they differ by a more palatable amount. Decibels and pH work on a simlar *logarithmic scale*, which are useful in measuring quantities that range by factors. On our typical scale, called a *linear scalar*, the amount between two markers is a common difference[ii]. The amount between two markers on a logarithmic scale is instead a constant ratio[iii]. By default, an attribute lies on a linear scale, but these should be modified to a logarithmic scale or some other scale when appropriate.

4.5.2 Boolean Features

An attribute is *quantitative* if it is acquired by measuring or counting something in the observation, such as length or mass. Conversely, an attribute is *qualitative* if it is not quantitative but instead represents some category, such as color, location, or gender. Now qualitative data can sometimes be represented using numbers, such as a ZIP code, home address, or identification numbers, but these are still categorical labels with no arithmetical interpretations (it does not mean anything to add or multiply together ZIP codes). A Boolean it a great example of a qualitative variable (`True` or `False`) often expressed numerically (`1` or `0`). Quantitative variables work well with mathematical modeling because arithmetic to them is meaningful. Numerically represented qualitatives do not have this same characteristic, or at least not without some modification. A Boolean in a model can work well to add a fixed amount (if an affine model is used) when the property is present but otherwise add nothing when the property is absent. By nature, qualitative variables have to do with grouping. To treat general qualitative variables, it is useful to seperate the categories into a sequence of Booleans, where the sequence is one shorter than the number of categories, where a `True` for one term in the sequence means the assignment belongs to that group and no other. Thus, only one `True` can appear for that qualitative variable. A full sequence of `False`'s means that the assignment is none of the listed groups but instead the one group not listed (the so-called *default group*). This is called **one-hot encoding** (or **Boolean encoding**).

There are also variables which can be both quantitative and qualitative. For example, the age of a patient is a quantitative variable as one's age does measure the passage of time and has (limited) arithmetic meaning, but it is also qualitative as people of similar ages often have similar life styles and health problems. If age is treated as a quantitative variable in an affine model, then increasing by one year from 10 to 11 would add the same health risk as someone else ageing from 90 to 91, which seems a little absurd.

Example 4.5.2. Let us engineer the features of our housing model one more time. The attributes x_4 and x_5 are qualitative and need to be treated as such. The fact that they are qualitative variables is probably why their correlation to y appeared weak. Attribute x_4, about condo status, is already Boolean and needs no further adjustment. Attribute x_5 measures location using ZIP code and should not be treated as a quantitative variable, as we did in Example 4.5.1. We will put the homes into four geographic regions based upon their location in Sacramento. As such, we will introduce three Boolean features f_1, f_2, and f_3. Now, $f_1(x_5)$ measures if ZIP code x_5 belongs to Region 2 or not, $f_2(x_5)$ measures if x_5 belongs to Region 3, and $f_3(x_5)$ measures if x_5 belongs ot Region 4. If all $f_i(x_5)$ are zero, then x_5 belongs to Region 1.

Let us retry this problem using now the model

$$\widehat{y} = b + a_1x_1 + a_4x_4 + c_1f_1(x_5) + c_2f_2(x_5) + c_3f_3(x_5).$$

The table below shows that cross-validation:

[i] Named after the statistician Charles P. Winsor.

[ii] The markers on a linear scale increase according to an *arithmetic progression* $a_n = a_{n-1} + d = a_0 + nd$.

[iii] The markers on a logarithmic scale increase according to a *geometric progression* $a_n = a_{n-1}r = a_0r^n$.

k	b	a_1	a_4	c_1	c_2	c_3	Training Error	Test Error
1	120.05635254	132.45611038	-17.82572442	-107.55011792	-112.16156981	-20.38412996	0.26684814	0.29135667
2	158.97942651	129.94245959	-11.15840699	-140.48560006	-154.90440744	-66.34690236	0.26610289	0.29774069
3	123.41872796	127.94969393	-8.84184287	-102.76951829	-110.26003395	-27.67786271	0.28002255	0.23084926
4	119.54238656	124.47041402	-13.45212348	-95.41936397	-103.28114617	-16.44893687	0.27157587	0.26999142
5	122.13039391	122.76230931	-11.44654728	-96.78950007	-102.9321835	-12.16266847	0.26713546	0.28697731

Thus, this new model improves the prediction error from about 29.3% to 27%. ∎

4.5.3 Nonlinear Features

Next, we mention that often we have to **create new features**. This generally means enlisting nonlinear functions on the attributes $x_1, \ldots, x_n$, but this can also include other creations, such as replacing a qualitative feature of n groups into $n-1$ Boolean features, as discussed above. While we do not have space to exhaustively discuss all the reasons one might include certain nonlinear features, we can at least offer some intuition. In the case of data fitting, the functions we used were often supported by the "shape" of the data. This is easy to visualize for univariate data as it is 2-dimensional and can actually be seen. Higher dimensions are much more difficult to visualize, but some analogous notion of shape can also be wrangled in these contexts, although these geometric notions take us beyond linear algebra. But two notions we can briefly discuss here. The graph of a linear (or affine) map is, by virtue, flat and has no curvature. Higher exponents, such as x_1^2 or x_2^5, do have curvature. A piecewise linear functions, such as $(x_3 - a)_+$, while still flat locally, may have turning points, which allows it to fit data better than just a straight linear model. Thus, some allowance of curvature may be better at prediction. For example, if a feature x_i seems to affect the prediction differently beyond a certain amount a, then it makes sense to add the nonlinear feature $(x_i - a)_+$ to account for this benchmark.

Nonlinear features can also possibly capture the influence of interactions, for example the feature $x_i x_j$. If x_i and x_j are both Boolean variables, then $x_i x_j$ is only 1 when both attributes are present. Their combination then has an oppurtunity to influence the prediction.

Example 4.5.3. It is conjectured that the area of a home above 1500 has a different affect toward the price than those below 1500 square feet. Thus, we add the feature $(x_1 - 1.5)_+$ to our model and run cross-validation, which produces a relative prediction error still around 27%. We see no benefit to the piecewise function's inclusions, and thus remove it. After experimenting with differen $(x - a)_+$ for various a values, we still see no benefit from the piecewise function in this model. ∎

One piece to feature engineering is **stratifying your data**. This is related to the qualitative variables discussed above. Sometimes certain subsets behave very differently than the main group and must be treated seperately. For example, for a medical diagnostic it may make sense to separate male and female patients as male versus female biology may involve very distinct implications with regards to medicine. When a stratification is necessary, it might be appropriate to use completely different models for the different strata, e.g, a different diagnostic for male than for females. Clustering, such as the k-means algorithm, may be very useful for this type of feature engineering.

Lastly, no one can expect any model to be perfect. After all, many quantities we wish to predict may be subject to some element of randomness. Random variables can also be woven into the model to try to incorporate this, but that is a subject that takes us beyond linear algebra.

Python Programming

Example 4.5.4. The following code:

```
import numpy as np
from scipy.stats import pearsonr as pearson
import matplotlib.pyplot as plt

#cross validate the linear system Xc = y
def cross_validate(X,y,folds=5,display=False,label="Data",lower_x=0,upper_x=10,lower_y=0,upper_y=10):
    plt.ion()
    m,n = X.shape
    partition = np.array_split(np.random.permutation(m),folds)
    c = np.zeros((folds,n))
    training_error = np.zeros(folds)
    test_error = np.zeros(folds)
    for k in range(folds):
        test_index = partition[k]
        training_index = np.concatenate(partition[:k]+partition[k+1:])
        a = np.linalg.lstsq(X[training_index],y[training_index],rcond=None)[0]
        c[k,:] = a
        training_error[k] = np.linalg.norm(X[training_index] @ a - y[training_index])/np.linalg.norm(y[training_index])
        test_error[k] = np.linalg.norm(X[test_index] @ a - y[test_index])/np.linalg.norm(y[test_index])
        if display:
            plt.scatter(y[training_index], X[training_index] @ a, color='b', marker='o')
            plt.scatter(y[test_index], X[test_index] @ a, color='r', marker='x')
            plt.plot((lower_x,upper_x),(lower_y,upper_y) ,ls='--', c = 'r')
            plt.ylim(lower_y,upper_y)
            plt.xlim(lower_x,upper_x)
            plt.xlabel(f"Actual {label} (k= {str(k+1)})")
            plt.ylabel(f"Predicted {label}")
            plt.show()
    return c,training_error,test_error

###################### Example ######################
D = house_sales_data()
y = D['price']
m = len(y)
x = np.zeros((5,m))
x[0] = D['area']
x[1] = D['beds']
x[2] = D['baths']
x[3] = D['condo']
x[4] = D['location']

#Pearson Correlation Coefficients of Attributes
for i in range(len(x)):
    print(f"rho_{i} = {pearson(x[i],y)[0]}")
print(f"\ncorrelation between x_2 and x_3 = {pearson(x[1],x[2])[0]} \n")
print(f"correlation between x_1 and x_2 = {pearson(x[0],x[1])[0]} \n")

#cross-validate affine models
ns = [2,3,1]
for n in ns:
    X = np.concatenate([x[:n],[np.ones(m)]]).T
    coeff, train_error, test_error = cross_validate(X, y)
    model = "+".join([str(round(np.mean(coeff[:,i]),4)) + "x_" + str(i+1) for i in range(n+1)])
    print("y = " + model[:-3])
    print(f"train: {np.round(100*np.mean(train_error),2)}%")
    print(f"test: {np.round(100*np.mean(test_error),2)}%\n")

#cross-validate with Booleans
X = np.column_stack((np.ones(m),x[0],x[3],
                     [int(x5 == 2) for x5 in x[4]],
                     [int(x5 == 3) for x5 in x[4]],
                     [int(x5 == 4) for x5 in x[4]]))
coeff, train_error, test_error = cross_validate(X, y)
c = [round(np.mean(coeff[:,i]),4) for i in range(len(coeff[0]))]
```

```
model = f"{c[0]}+{c[1]}x_1+{c[2]}x_3\n"
model += "+".join([f"{c[i]}f_{i-2}(x_5)" for i in range(3,6)])
print("y = " + model)
print(f"train: {np.round(100*np.mean(train_error),2)}%")
print(f"test: {np.round(100*np.mean(test_error),2)}%\n")

#cross-validate with (x_1-a)_+
a = 1.5
X = np.column_stack((np.ones(m),x[0],x[3],
                     [int(x5 == 2) for x5 in x[4]],
                     [int(x5 == 3) for x5 in x[4]],
                     [int(x5 == 4) for x5 in x[4]],
                     [np.maximum(0, x1-a) for x1 in x[0]]))
coeff, train_error, test_error = cross_validate(X, y)
c = [round(np.mean(coeff[:,i]),4) for i in range(len(coeff[0]))]
model = f"{c[0]}+{c[1]}x_1+{c[2]}x_3\n"
model += "+".join([f"{c[i]}f_{i-2}(x_5)" for i in range(3,6)])
model += f"\n+{c[6]}(x_1-{a})_+"
print("y = " + model)
print(f"train: {np.round(100*np.mean(train_error),2)}%")
print(f"test: {np.round(100*np.mean(test_error),2)}%\n")
```

solves the examples in Section 4.5. ■

Exercises

(Go to Solutions)

For Exercises 1–1, use the data set from Example 4.3.2 to cross-validate a regression model as given. Find the average prediction error with 5 folds. How does it compare to Example 4.5.2? Interpret what this model is measuring differently from Example 4.5.2. *A calculator or computer is recommended.*

1. $y = b + a_1x_1 + c_1x_4f_1(x_5) + c_2x_4f_2(x_5) + c_3x_4f_3(x_5)$

2. Using the data set from Example 4.5.1, consider the model

$$y = 67.01x_1x_4 - 12.34x_2x_4 + 149.35x_1 - 22.81x_2 - 75.52x_4 + 67.86.$$

   Using this model, calculate the correlation coefficient for the regression. Interpret the coefficient on area. Does it make sense? Interpret the model in conext. Your interpretation should (generally) explain what the regression model tells us about house prices and how trustworthy the model is.

3. Complete all of the Python-based exercises found at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/feature.ipynb

"The small part of ignorance that we arrange and classify we give the name of knowledge."
– Ambrose Bierce

4.6 Classification

4.6.1 Classifiers

The regression models we have discussed so far provide us ways to predict quantitative variables using other quantitatives and, with the use of feature engineering, qualitative variables. Can a model be created to predict a qualitative variable? This question frame the famous *Classification Problem*. Let us start with the simplest possibility, a boolean variable of `True` or `False`, naturally called *binary classification* (or *boolean classification*). This type of classification serves to provide a diagnostic if a patient is sick or healthy, if an email is spam or not spam, or if a song will be liked or not liked by the listener. We can quantify a boolean using 1 and 0, respectively, although in this conservation about classification we will elect to use 1 and -1 instead. The set $\{1, -1\}$ is typically denoted as $\mathbb{Z}_2$.

Definition 4.6.1. A (**binary**) **classifier** is a function f of the form $f : \mathbb{R}^n \to \mathbb{Z}_2$.

As we saw with regression models, (relative) prediction error is an appropriate metric to measure how good a classifier is. As there are only two possible output of a classifier, there are only two possible prediction errors that can occur. Let us say we want to predict the boolean variable y with prediction $\widehat{y}$. If $y = \widehat{y} = 1$, then it is a **true positive**, and no prediction error occurred. Likewise, if $y = \widehat{y} = 0$, then it is called a **true negative**, and again no error occured. A **false positive** is when $y = 0$ but $\widehat{y} = 1$. A **false negative** occurs when $y = 1$ but $\widehat{y} = 0$.

Definition 4.6.2. Let $\boldsymbol{y}$ be a boolean n-vector predicted by $\widehat{\boldsymbol{y}}$. With regard to $\widehat{\boldsymbol{y}}$ predicting $\boldsymbol{y}$, let $\boldsymbol{y}_{tn}$ denote the number of true negatives, $\boldsymbol{y}_{fp}$ denote the number of false positives, $\boldsymbol{y}_{fn}$ denote the number of false negatives, and $\boldsymbol{y}_{tp}$ denote the number of true positiives. Then the **confusion matrix** of $\boldsymbol{y}$ is the 2×2 matrix $C_{\boldsymbol{y}}$ such that

$$C = \begin{bmatrix} \boldsymbol{y}_{tn} & \boldsymbol{y}_{fp} \\ \boldsymbol{y}_{fn} & \boldsymbol{y}_{tp} \end{bmatrix}.$$

Note that the sum of the first row of C, $\boldsymbol{y}_n := \boldsymbol{y}_{tn} + \boldsymbol{y}_{fp}$, is the number of actual negatives in $\boldsymbol{y}$, the sum of the second row of C, $\boldsymbol{y}_p := \boldsymbol{y}_{fn} + \boldsymbol{y}_{tp}$, is the number of actual positives in $\boldsymbol{y}$, the sum of the first column of C, $\widehat{\boldsymbol{y}}_n := \boldsymbol{y}_{tn} + \boldsymbol{y}_{fn}$, is the number of predicted negatives in $\widehat{\boldsymbol{y}}$, and the sum of the first column of C, $\widehat{\boldsymbol{y}}_p := \boldsymbol{y}_{fp} + \boldsymbol{y}_{tp}$, is the number of predicted positives in $\widehat{\boldsymbol{y}}$. Of course, the sum of all the entries in C $\boldsymbol{y}_{tn} + \boldsymbol{y}_{fp} + \boldsymbol{y}_{fn} + \boldsymbol{y}_{tp} = n$.

The **error rate** of a prediction $\widehat{\boldsymbol{y}}$ on $\boldsymbol{y}$ is $\dfrac{\boldsymbol{y}_{fp} + \boldsymbol{y}_{fn}}{n}$. Conversely, the **accuracy** is $\dfrac{\boldsymbol{y}_{tn} + \boldsymbol{y}_{tp}}{n}$. The **true negative rate** (or **specificity**) is $\dfrac{\boldsymbol{y}_{tn}}{\boldsymbol{y}_n}$ and tells us the proportion of negative cases correctly predicted. The **false positive rate** is $\dfrac{\boldsymbol{y}_{fp}}{\boldsymbol{y}_n}$ and tells us the proportion of negative cases incorrectly predicted as positives. The **true positive rate** (or **recall** or **sensitivity**) is $\dfrac{\boldsymbol{y}_{tp}}{\boldsymbol{y}_p}$ and is the proportion of positive cases correctly predicted. The **false negative rate** is $\dfrac{\boldsymbol{y}_{fn}}{\boldsymbol{y}_p}$ and tells us the proportion of positive cases incorrectly predicted as negatives. Finally, **precision** is $\dfrac{\boldsymbol{y}_{tp}}{\widehat{\boldsymbol{y}}_p}$ and tells us the proportion of positive predictions that are correct.

Clearly, a good classifier will have accuracy, specificity, sensitivity, and precision all close to 100%. Conversely, their complements should be close ot 0%. Why are there so many measurements on the efficacy of a classifer? Well, they measure slightly different things and these nuances are important in certain circumstances.

For example, a false positive is not always the same level of a mistake as a false negative. Consider a medical patient who might have cancer. If the diagnotic test tells the patient that she has cancer but really does not (a false positive), she will seek further testing and begin treatment. Eventually, the doctors will discover that she does not have cancer and the unnecessary treatements will likely have no long term consequences. On the other hand, if the test says she does not have cancer but she really does (a false negative), then the patient will forego potentially life-saving early treatments. This would be disastrous. Consider a classifying algorithm to detect spam emails. If a spam email is not detected (a false negative), then it lands in the user's inbox only to the annoyance of the user because he is wise enough not to fall for even the most eloborate phishing scams. On the other hand, if an important email is labeled as spam (a false positive), the user may not ever read it and could miss out on an important business oppurtunity. Therefore, as the severity of a false positive or false negative may be unbalanced, different importance may be placed upon these metrics of efficacy.

4.6.2 Least Squares Classifiers

In this section, we discuss the method of **least squares classification**. We first motivate it using some geometry. A hyperplane in $\mathbb{R}^n$ is given by an equation $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$, or $\boldsymbol{a} \cdot \boldsymbol{x} = b$. Every hyperplane separates the vector space into two regions: *above the hyperplane* characterized by those vectors $\boldsymbol{x}$ such that $\boldsymbol{a} \cdot \boldsymbol{x} > b$ and *below the hyperplane* characterized by those vectors $\boldsymbol{x}$ such that $\boldsymbol{a} \cdot \boldsymbol{x} < b$. For example, imagine how a line seperates the plane and how a plane separates 3-space into two regions. Let $T : \mathbb{R}^n \to \mathbb{R}$ be the affine transformation $T(\boldsymbol{x}) = \boldsymbol{a} \cdot \boldsymbol{x} - b$. Then $\boldsymbol{x}$ is above the hyperplane if $T(\boldsymbol{x}) > 0$ and below the hyperplane when $T(\boldsymbol{x}) < 0$. A classifier $f : \mathbb{R}^n \to \mathbb{Z}_2$ is then obtained when $f(\boldsymbol{x})$ is equal to the sign of $T(\boldsymbol{x})$, denoted $f(\boldsymbol{x}) = \operatorname{sgn}(T(\boldsymbol{x}))$, that is, $f(\boldsymbol{x}) = +1$ when $T(\boldsymbol{x}) > 0$ and $\boldsymbol{x}$ is above the hyperplane and $f(\boldsymbol{x}) = -1$ when $T(\boldsymbol{x}) < 0$ and $\boldsymbol{x}$ is below the hyperplane. If a set of points is given in the form $(\boldsymbol{x}, 1)$ or $(\boldsymbol{x}, -1)$, then T can be computed by the least squares method to fit the data. Then the associated signage function f on T gives us the **least squares classifier**.

In our above discussion, T was an affine function to model the separating hyperplane. There are of course non-linear functions that can be used to separate $\mathbb{R}^n$. For example, a circle separates the plane into two regions: *inside* and *outside* the circle. Thus, a non-linear model $T : \mathbb{R}^n \to \mathbb{R}$ can be used when the boundary between `True` and `False` is not well modeled by a flat hyperplane.

Example 4.6.3. A famous classification problem due to the statistician Ronald Fisher is the classification of iris flowers into three types: *Setosa*, *Versicolour*, and *Virginica*. We will build a least squares classifier to predict whether an iris is virginica. The classification will utilize four attributes: x_1 is the sepal length in cm, x_2 is the sepal width in cm, x_3 is the petal length in cm, and x_4 is the petal width in cm. Our classifier will have then the form

$$f(\boldsymbol{x}) = \operatorname{sgn}(a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + b).$$

Fisher's data set contains 150 observations, where the three types are labeled 0, 1, and 2, respectively. Thus our data points will have the form $(\boldsymbol{x}, 1)$ if the iris is virginica and $(\boldsymbol{x}, -1)$ otherwise. Solving the least squares problem to fit a hyperplane to this data set, we get the classifier

$$f(x) = \operatorname{sgn}(-0.09342534x_1 + 0.40854914x_2 + 0.0057993x_3 + 1.10845342x_4 - 2.38876778).$$

Using cross-validation, this model has an average accuracy of 93.00% and 91.33%, an average specificity of 92.98% and 92.82%, an average sensitivity of 93.01% and 87.63%, and an average precision of 87.08% and 88.14% on the training set and test set, respectively, across a 5-fold cross-validaiton. Therefore, this model is highly effective in predicting whether an iris is virginica or not.

The following code can be used to solve this example:

```
from sklearn.datasets import load_iris
iris_types = load_iris()['target']
iris_data = np.array(load_iris()['data'])

n = len(iris_types)
X = np.column_stack([iris_data, np.ones(n)])
```

```
28
29 IsVirginica = np.array([boolean(iris == 2) for iris in iris_types])
30
31 coeff, training_matrix, test_matrix = cross_validate_boolean(X,IsVirginica)
32
33 model = "+".join([str(round(np.mean(coeff[:,i]),4)) + "x_" + str(i+1) for i in range(5)])
34 print("y = " + model[:-3] + "\n")
35
36 print("Training Confusion Matrix\n", training_matrix)
37 print("\nTest Confusion Matrix\n", test_matrix)
38
39 accuracy =  np.array([[(training_matrix[j,0,0]+training_matrix[j,1,1])/sum(sum(
       training_matrix[j])),(test_matrix[j,0,0]+test_matrix[j,1,1])/sum(sum(test_matrix[j]))]
       for j in range(5)])
40 print("\nAccuracy = ( ( true pos + true neg ) / total )  [training, test]\n",
41       accuracy, "\n\n", sum(accuracy)/5, "\n\n")
42
43 specificity = np.array([[training_matrix[j,0,0]/(training_matrix[j,0,0]+training_matrix[j
       ,0,1]),test_matrix[j,0,0]/(test_matrix[j,0,0]+test_matrix[j,0,1])] for j in range(5)])
44 print("\nSpecificity = ( true neg / actual neg )  [training, test]\n",
45       specificity, "\n\n", sum(specificity)/5, "\n\n")
46
47 recall = np.array([[training_matrix[j,1,1]/(training_matrix[j,1,0]+training_matrix[j,1,1]),
       test_matrix[j,1,1]/(test_matrix[j,1,0]+test_matrix[j,1,1])] for j in range(5)])
48 print("\nSensitivity/Recall = ( true pos / actual pos )  [training, test]\n",
49       recall, "\n\n", sum(recall)/5, "\n\n")
50
51 precision = np.array([[training_matrix[j,1,1]/(training_matrix[j,0,1]+training_matrix[j
       ,1,1]),test_matrix[j,1,1]/(test_matrix[j,0,1]+test_matrix[j,1,1])] for j in range(5)])
52 print("\nPrecision = ( true pos / predict pos )  [training, test]\n",
53       precision, "\n\n", sum(precision)/5, "\n\n")
```

4.6.3 Multi-class Classifiers

Now that we have discussed the boolean case, what about a general qualiative variable with k possible bins. For example, the Likert scale has surveyors rank statements with the labels *Strongly Disagree*, *Disagree*, *Neutral*, *Agree*, or *Strongly Agree*. These labels are encoded using -2, -1, 0, 1, and 2. In this case, the classfier would need to assign one of these five values to the input vector $\boldsymbol{x}$. Let $\mathbb{Z}_k = \{0, 1, 2, \ldots, k-1\}$. Then we must now consider the problem of assigning a vector to one of the elements of $\mathbb{Z}_k$.

Definition 4.6.4. A **k-class classifier** is a function of the form $f : \mathbb{R}^n \to \mathbb{Z}_k$. When k is unspecified, f is called a **multi-class classifer**.

Let $\boldsymbol{y}$ be a qualitative n-vector with k categories predicted by $\widehat{\boldsymbol{y}}$. Let $\boldsymbol{y}_{ij}$ denote the number of entries of $\boldsymbol{y}$ where $y = i$ and $\widehat{y} = j$. Then the **confusion matrix** of $\boldsymbol{y}$ is the $k \times k$ matrix $C_{\boldsymbol{y}}$ such that

$$C = \left[\boldsymbol{y}_{ij} \right].$$

Please note that this generalizes the 2×2 confusion matrix defined above.

The diagonal entries of $C_{\boldsymbol{y}}$ are those predictions which are correct, and the off-diagonal entries are errors. Therefore, a multi-class classifier is accurate when its confusion matrix is approximately diagonal. In particular, the **accuracy** of a multi-classifier is $\dfrac{\operatorname{tr}(C_{\boldsymbol{y}})}{n}$.

Much like how we encoded qualitative variables as a sequence of boolean attributes for feature engineering (hot-button encoding), we can encode a k-class classifier as a sequence of k binary classifiers. For each of the categories $i \in \mathbb{Z}_k$, define $T_i : \mathbb{R}^n \to \mathbb{R}$ which measures how strongly $\boldsymbol{x}$ belongs to bin i based upon $T_i(\boldsymbol{x})$. The larger $T_i(\boldsymbol{x})$ is the more likely $\boldsymbol{x}$ belongs to bin i. The smaller $T_i(\boldsymbol{x})$ is the less likely $\boldsymbol{x}$ belong to bin i. Then define $f(\boldsymbol{x})$ to be the index j of the maximum among $T_1(\boldsymbol{x})$, $T_2(\boldsymbol{x}), \ldots, T_k(\boldsymbol{x})$, that is,

$$f(\boldsymbol{x}) = \operatorname{argmax}(T_1(\boldsymbol{x}), T_2(\boldsymbol{x}), \ldots, T_k(\boldsymbol{x})).$$

This is known as the *one-versus-all* multi-class classification method.

Example 4.6.5. Let us revisit the iris classification problem from Example 4.6.3, but this time let us try to classify the flowers using all three types: *Setosa*, *Versicolour*, and *Virginica*. We use least squares to construct three affine regressions with respect to these three types, listed below, and our classifier is then given by the formula:

$$f(\boldsymbol{x}) = \operatorname{argmax} \begin{cases} T_0(\boldsymbol{x}) &= 0.13386706x_1 + 0.48665644x_2 - 0.44882137x_3 - 0.11857828x_4 - 0.77450658 \\ T_1(\boldsymbol{x}) &= -0.04557434x_1 - 0.89097497x_2 + 0.44399217x_3 - 0.98747422x_4 + 2.17328698 \\ T_2(\boldsymbol{x}) &= -0.08829271x_1 + 0.40431853x_2 + 0.00482919x_3 + 1.1060525x_4 - 2.3987804 \end{cases}$$

Using cross-validation, this model has an average accuracy of 85% and 84% on the training set and test set, respectively, across a 5-fold cross-validation. Therefore, this model is reasonably effective in classifying an iris into one of these three types.

The following code can be used to solve this example:

```
from sklearn.datasets import load_iris
iris_names = load_iris()['target_names']
iris_types = load_iris()['target']
iris_data = np.array(load_iris()['data'])

print(iris_data)

n = len(iris_types)
X = np.column_stack([iris_data, np.ones(n)])
coeff, train_matrix, test_matrix = cross_validate_multiclass(X, iris_types, len(iris_names))
for j in range(len(iris_names)):
    print(iris_names[j])
    print(coeff[j], "\n")
    model = "+".join([str(round(np.mean(coeff[j,:,i]),4)) + "x_" + str(i+1) for i in range
    (5)])
    print("y = " + model[:-3] + "\n")

print("Training Confusion Matrix\n", train_matrix, "\n")
print("Test Confusion Matrix\n", test_matrix)

accuracy =  np.array([[np.trace(train_matrix[j])/sum(sum(train_matrix[j])),np.trace(
    test_matrix[j])/sum(sum(test_matrix[j]))] for j in range(5)])
print("\nAccuracy = [training, test]\n",accuracy,"\n\n", sum(accuracy)/5, "\n\n")

v = np.array([4,3,2,1])
print(np.argmax([coeff[j] @ v for j in range(len(iris_names))]))
```

■

Exercises

(Go to Solutions)

For Exercises 1–1, construct a binary classifier for the data given in the matrix X and the types given in $\boldsymbol{y}$. The regression model should be affine, ie, $\widehat{y} = a_1x_1 + \ldots + a_nx_n + b$ where $\boldsymbol{x}_i$ is the ith column of X. Use the classifier to predict whether the given vectors $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ have the quality or not. You do not need to cross-validate your classifier. *A calculator or computer is recommended.*

1. $X = \begin{bmatrix} 1.0 & 2.0 \\ 1.5 & 1.8 \\ 2.0 & 1.0 \\ 2.5 & 2.2 \\ 3.0 & 3.0 \\ 3.5 & 2.8 \\ 4.0 & 3.5 \\ 4.5, 3.0 & \end{bmatrix}$, $\boldsymbol{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$, $\boldsymbol{u} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $\boldsymbol{v} = \begin{bmatrix} 10 \\ 2 \end{bmatrix}$, $\boldsymbol{w} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$

For Exercises 2–3, use the data set from Example 4.6.5 to construct a multi-class classifier for the three types of iris flowers. Use it to predict the type for the given vector. *A calculator or computer is recommended.*

2. $[4, 3, 2, 1]$

3. $[1, 1, 1, 1]$

4. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/classify.ipynb

"We all live with the objective of being happy; our lives are all different and yet the same." – Anne Frank

4.7 Multi-Objective Least Squares

4.7.1 Multi-Objectives

Recall that the beauty of the least squares method is that it is a simple linear approach to solving an optimization problem. Given a linear equation $A\boldsymbol{x} = \boldsymbol{b}$, then a least squares solutions $\widehat{\boldsymbol{x}}$ optimizes the objective $\|A\boldsymbol{x} - \boldsymbol{b}\|$, that is,

$$\|A\widehat{\boldsymbol{x}} - \boldsymbol{b}\| \leq \|A\boldsymbol{x} - \boldsymbol{b}\|$$

for all $\boldsymbol{x} \in \mathbb{R}^n$. For simplicity's sake, this optimization problem is equivalent to minimizing the objective $\|A\boldsymbol{x} - \boldsymbol{b}\|^2$. In many applications, it is important to consider multiple objectives simulatenously. Let $J_1, J_2, \ldots, J_k$ be a list of objectives to minimize. We can then consider these multiple objectives together as a single objective, called the **multi-objective**, by considering them as a linear combination, namely

$$J = \lambda_1 J_1 + \lambda_2 J_2 + \ldots + \lambda_k J_k,$$

for $\lambda_i \geq 0$. Increasing the magnitude of a coefficient λ_i increasing the need to minimize the objective J_i in order to minimize the multi-objective J, thus increasing the significance of objective J_i. Conversely, decreasing the magnitude of λ_i, say lowering close to 0, reduces the need to minimize J_i individually and thus reduces its overall importance in the multi-objective J. Typically, there is one objective that is considered more important than all the rest, called the **main objective**, and is typically denoted as J_1. As minimizing the multi-objective aJ, for some $a > 0$, is equivalent to minimizing J, we may, without the loss of generality, set $\lambda_1 = 1$. Thus,

$$J = J_1 + \lambda_2 J_2 + \ldots + \lambda_k J_k.$$

If $\lambda_i < 1$, then it will have less impact on the minimization on J than J_1 does. Conversely, if $\lambda_i > 1$, then J_i will have greater impact on the optimization of J than J_1 (while this second case is very unlikely because, after all, J_1 is the main objective, we do allow this as an option for the sake of feature engineering).

Consider now the situation where each objective has the form $J_i = \|A_i\boldsymbol{x} - \boldsymbol{b}_i\|^2$ for some $m_i \times n$ matrix A_i and some m_i-vector $\boldsymbol{b}_i$. Individually, we could optimize J_i by finding a least squares solution $\widehat{\boldsymbol{x}}_i$ to $A_i\boldsymbol{x} = \boldsymbol{b}_i$. To optimize J though, we need to find a vector $\widehat{\boldsymbol{x}}$ which simultaneously makes $\|A_i\boldsymbol{x} - \boldsymbol{b}_i\|^2$ small (relative to its weight λ_i). Of course, if $\widehat{\boldsymbol{x}}$ was a simulatenous least squares solution to each of the equation $A_i\boldsymbol{x} = \boldsymbol{b}_i$, then this would suffice. Of course, the existence of such a simulatenous least squares solution is unlikely in practice. But this was the motivation behind the least squares problem in the first place, when no solution is present use the best approximation. We can do this for multi-objectives too.

Let $m = m_1 + m_2 + \ldots + m_k$, let A be an $m \times n$ matrix, and let $\boldsymbol{b}$ be an m-vector such that

$$A = \begin{bmatrix} A_1 \\ \sqrt{\lambda_2}A_2 \\ \vdots \\ \sqrt{\lambda_k}A_k \end{bmatrix}, \qquad \boldsymbol{b} = \begin{bmatrix} \boldsymbol{b}_1 \\ \sqrt{\lambda_2}\boldsymbol{b}_2 \\ \vdots \\ \sqrt{\lambda_k}\boldsymbol{b}_k \end{bmatrix}.$$

Let $\widehat{\boldsymbol{x}}$ then be a least squares solution to the linear equation $A\boldsymbol{x} = \boldsymbol{b}$. Then $\widehat{\boldsymbol{x}}$ minimizes the objective $J = \|A\boldsymbol{x} - \boldsymbol{b}\|^2 = \|A_1\boldsymbol{x} - \boldsymbol{b}_1\|^2 + \lambda_2\|A_2\boldsymbol{x} - \boldsymbol{b}_2\|^2 + \ldots + \lambda_k\|A_k\boldsymbol{x} - \boldsymbol{b}_k\|^2$. Thus, for fixed weights λ_i, $\widehat{\boldsymbol{x}}$, the least squares solution found by stacking the objectives, is the best solution to the objective J. More specificly, there does not exists a vector $\boldsymbol{z} \in \mathbb{R}^n$ such that

$$\|A_i\boldsymbol{z} - \boldsymbol{b}_i\|^2 \leq \|A_i\widehat{\boldsymbol{x}} - \boldsymbol{b}_i\|^2,$$

for all i with at least one inequality being strict (note that in the presence of a district least squares solution, equality would occur in all positions). If not, then $\boldsymbol{z}$ would minimize J better than $\widehat{\boldsymbol{x}}$, which would contradict $\widehat{\boldsymbol{x}}$ be a least squares solution to $A\boldsymbol{x} = \boldsymbol{b}$. Thus, $\widehat{\boldsymbol{x}}$ is not just an optimal solution to the multi-objective J,

it is *Pareto optimal*, meaning that the improvement in any dimension could only happen s the worsening of another dimension.

But "best" here is relative to the weights. As any coefficient λ_i is adjusted, it will consequentially change the possible vector assignments to $\widehat{\boldsymbol{x}}$. Thus, we can view $\widehat{\boldsymbol{x}}$ as the image of a function on $k-1$ variables $\lambda_2, \ldots, \lambda_k$. Because of Pareto optimality, any adjustment to a λ_i is a trade-off, an optimal trade-off mind you, but still a trade-off. Raising λ_i will raise the significance of J_i thus leading to an improved J_i objective but at the worsening of at least one other objective. Lowering λ_i will worsen J_i but improve at least one other objective. Therefore, choosing the right weights is determined by feature engineering. This much we may offer here. Even a small change of λ_i can cause a huge improvement or deteriment to J_i, so experimentation on these weights is very much worth the effort.

4.7.2 Applications of Multi-Objective Least Squares

Example 4.7.1. Consider two scalar quantities t_i, s_i such that $f : \mathbb{R} \to \mathbb{R}$ is an affine function such that $f(t_i) = at_i + b = s_i$. Let $\boldsymbol{t}$ and $\boldsymbol{s}$ then be time series which record these dynamic scalars. If we have a desired target $\boldsymbol{s}$, then clearly the main objective $J_1 = \|f(\boldsymbol{t}) - \boldsymbol{s}\|^2$ is present, where $f(\boldsymbol{t}) = (f(t_1), f(t_2), \ldots, f(t_n)) = a\boldsymbol{t} + b\boldsymbol{1}$. As the vector $\boldsymbol{t}$ is itself a time series, the transition from t_i to t_{i+1} may come at some kind of cost. Therefore, it may be desirable to have $x_{i+1} \approx x_i$, that is, we want the difference $|x_{i+1} - x_i|$ to be small for each transition. Such a requirement would be a *smooth transition*. Then the second objective here would be $J_2 = \|D\boldsymbol{t}\|^2$ where D is the $(n-1) \times n$ difference matrix

$$D = \begin{bmatrix} -1 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & -1 & 1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \vdots & -1 & 1 & 0 \\ 0 & 0 & 0 & \vdots & 0 & -1 & 1 \end{bmatrix}.$$

Therefore, we would seek a least squares solution $\widehat{\boldsymbol{t}}$ to the linear system $\begin{bmatrix} aI_n \\ \sqrt{\lambda}D \end{bmatrix} \boldsymbol{t} = \begin{bmatrix} \boldsymbol{s} - b\boldsymbol{1} \\ \boldsymbol{0} \end{bmatrix}$. ∎

Example 4.7.2. Continuing with the previous example, image that $\boldsymbol{x}$ is a smooth periodic time series with period P. This means that $\boldsymbol{t} = (\boldsymbol{t}_{1:P}, \boldsymbol{t}_{1:P}, \ldots, \boldsymbol{t}_{1:P})$, that is, the entries of $\boldsymbol{t}$ repeat themselves every P units of time, but also that $|t_{i+1} - t_i|$ is small, that is, the transition from t_i to t_{i+1} is relatively small. Now, suppose that $\boldsymbol{s} = \boldsymbol{t} + \boldsymbol{\varepsilon}$ where $\boldsymbol{\varepsilon}$ is some *noise* vector. What we mean by this is that ε_i represents some amount of chaos in the system and distorts $\boldsymbol{s}$ from the perfect periodicity of $\boldsymbol{t}$. For example, the hourly temperature over several days in a specific location is approximately periodic on a daily cyclic ($P = 24$). The temperature at noon on Monday gives a great predictor of the temperature at noon on Tuesday, but there are other meteorological affects that might change this, such as clouds or a cold front. Likewise, if we look at the daily high temperature of this same region over the course of several years, we will see that daily high temperatures are approximately periodic on an annual cycle ($P = 365$). The high temperature on 19 November 2021 is a strong prediction of the high temperature on 19 November 2022. But various weather patterns can through noise into the time series. The shopping or dining habits of customers typically can be modeled weekly. The number of customers buying pizza on Tuesday is comparable to any other Tuesday and its typically the lowest shopping day of the week, especially compared to the high sales of Friday or Saturday (hence why so many businesses offer specials exclusive to Tuesday).

Suppose then that $\boldsymbol{s}$ is an observed time series that is approximately periodic on P units. Then our main objective will be to find a P-vector $\boldsymbol{x}$ so that $\boldsymbol{t} = (\boldsymbol{x}, \boldsymbol{x}, \ldots, \boldsymbol{x}) \approx \boldsymbol{s}$. This implies that $J_1 = \|A\boldsymbol{x} - \boldsymbol{s}\|^2$ where

$A = \begin{bmatrix} I_P \\ \vdots \\ I_P \end{bmatrix}$. But the time series $\boldsymbol{t}$ (and hence $\boldsymbol{x}$) should also be smooth. As $t_{P+1} = x_1$ and $t_P = x_P$, we will this time use the $P \times P$ *cyclic difference matrix* matrix D given as

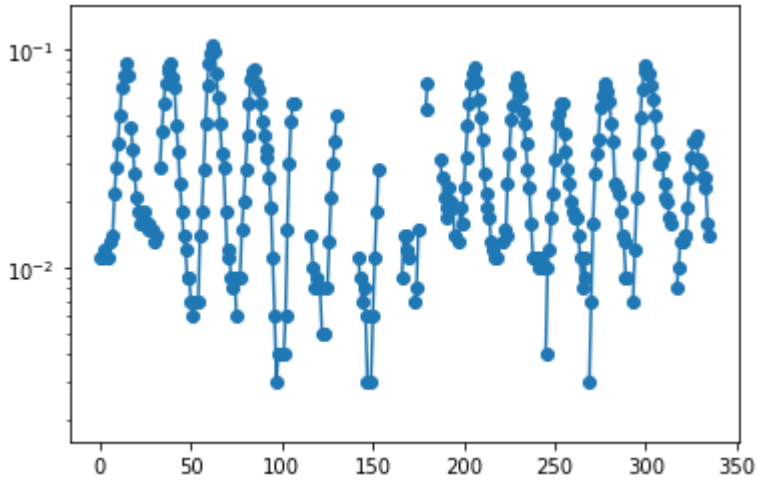
$$D = \begin{bmatrix} -1 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & -1 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \vdots & -1 & 1 & 0 \\ 0 & 0 & 0 & \vdots & 0 & -1 & 1 \\ 1 & 0 & 0 & \vdots & 0 & 0 & -1 \end{bmatrix}.$$

Note that this difference matrix is identical to the difference matrix from Example 4.7.1 but with one extra row which insures that the transition from the end of a cycle to the start of a new cycle is likewise smooth. Thus, $J_2 = \lambda\|D\boldsymbol{x}\|^2$. Therefore, we would seek a least squares solution $\widehat{\boldsymbol{x}}$ to the linear system $\begin{bmatrix} A \\ \sqrt{\lambda}D \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{0} \end{bmatrix}$.

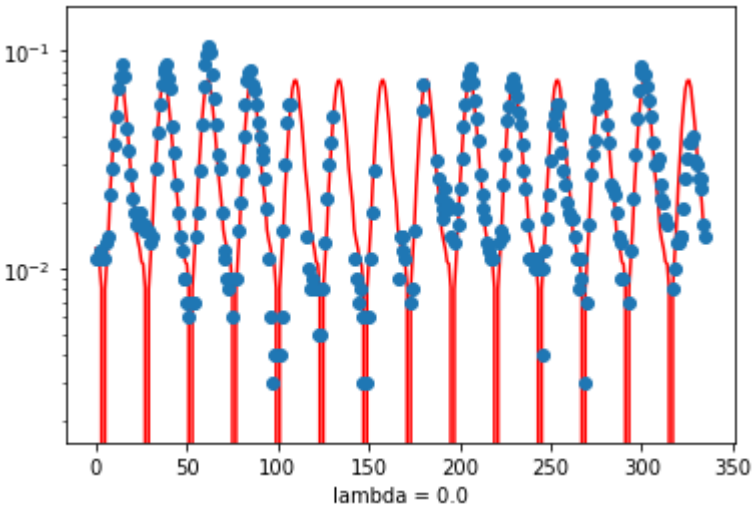
The time series $A\widehat{\boldsymbol{x}}$ is called the **extracted seasonal component**, and $\boldsymbol{s} - A\widehat{\boldsymbol{x}}$ is called the **seasonally adjusted** time series. ■

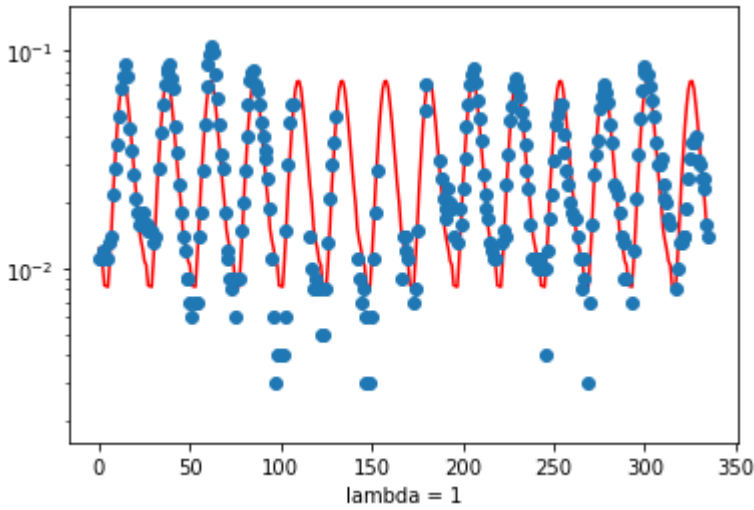
Example 4.7.3. As a specific example, consider the hourly ozone level (ppm) at Azusa, California, during the 14 days of July 2014 (California Environmental Protection Agency, Air Resources Board, www.arb.ca.gov). Measurements start at 12AM on July 1st, and end at 11PM on July 14. Note the large number of missing measurements. In particular, all 4AM measurements are missing. We desire to construct a periodic time series to model the ozone level logarithmically transformed measurements, like we did in Example 4.3.8. Given a period of $p = 24$ hours, let $\boldsymbol{x} = (t_1, \dots, t_{24})$ and let $\boldsymbol{t} = (\boldsymbol{x}, \boldsymbol{x}, \dots, \boldsymbol{x})$, where $\boldsymbol{x}$ is repeated in $\boldsymbol{t}$ 14 times for the 14 days we are studying. Then our primary objective is $J_1 = \|A\boldsymbol{x} - \boldsymbol{t}\|^2$, that is, we want to solve the linear system $A\boldsymbol{x} = \boldsymbol{t}$:

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_p \end{bmatrix} = \boldsymbol{t}$$



via least squares. Note that the 24×24 identity matrix will be stacked in A 14 times. If we solve the least squares problem $A\boldsymbol{x} = \boldsymbol{t}$, we discover some wild behavior at the bottom of the regression curve, as illustrated below left. To better control this behavior, we can require smooth transitions on the time series, via a 24×24 cyclic difference matrix D. More specificially, we attach to our main objective the secondary objective of smoothness $J_2 = \lambda \|D\boldsymbol{x}\|^2$. Upon experimentation, even a small value of λ removes the wild behavior of the least squares solution. On the other hand, a large value of λ causes the solution to be too smooth, that is, the regression cannot deviate much from the previous value. This causes the amplitude of the oscillation to be dampened. Thus, $\lambda = 1$ appears to create a great fit to the periodic data, as illustrated below right.





The following code can be used to solve this example:

```
import numpy as np

def multi_lstsq(As, bs, lambdas):
from scipy.linalg  import circulant
cyc_diff = lambda n : circulant(np.concatenate(([-1,1], np.zeros(n-2)))).T

############################## Example ##############################
def ozone_data():
ozone = ozone_data() # a vector of length 14*24 = 336
k = 14
N = k*24
lamb = 1

#graph data
import matplotlib.pyplot as plt
plt.ion()
plt.plot(np.arange(N), ozone[:N], 'o-')
plt.yscale('log')
plt.ylim(10**(-2.8),10**(-0.8))
plt.show()

#stack of k many 24x24 identity matrices
A = np.concatenate([np.identity(24) for i in range(k)])
#24x24 cyclic difference matrix
D = cyc_diff(24)

#Identifies where there is complete data, as opposed to missing data "NaN"
ind = [k for k in range(N) if ~np.isnan(ozone[k])]

#solve multi-objective least squares problem
x = multi_lstsq([A[ind],D],[ozone[ind],np.zeros(24)], [1,lamb])

#graph solution
plt.plot(np.arange(N), np.concatenate([x for i in range(k)]), 'r')
plt.plot(np.arange(N), ozone[:N], 'o')
plt.yscale('log')
plt.ylim(10**(-2.8),10**(-0.8))
```

```
179 plt.xlabel('lambda = ' + str(lamb))
180 plt.show()
```

Python Programming

Example 4.7.4. The following code:

```
import numpy as np
import matplotlib.pyplot as plt

def multi_lstsq(As, bs, lambdas):
    k = len(lambdas)
    A = np.concatenate([np.sqrt(lambdas[i])*As[i] for i in range(k)])
    b = np.concatenate([np.sqrt(lambdas[i])*bs[i] for i in range(k)])
    return np.linalg.lstsq(A,b, rcond=None)[0]

As = np.array([np.random.normal(size = (10,5)),
np.random.normal(size = (10,5))])
bs = np.vstack([np.random.normal(size = 10),
np.random.normal(size = 10)])
N = 200
lambdas = np.power(10, np.linspace(-4, 4, 200))
x = np.zeros((5,N))
J1 = np.zeros(N)
J2 = np.zeros(N)
for k in range(N):
    x[:,k] = multi_lstsq(As, bs, [1, lambdas[k]])
    J1[k] = np.linalg.norm(As[0]@x[:,k] - bs[0])**2
    J2[k] = np.linalg.norm(As[1]@x[:,k] - bs[1])**2


plt.ion()
# plot solution versus lambda
plt.plot(lambdas, x.T)
plt.xscale('log')
plt.xlabel('lambda')
plt.xlim((1e-4, 1e+4))
plt.legend(['y_1', 'y_2', 'y_3', 'y_4', 'y_5'], loc= 'upper right')
plt.show()

# plot two objectives versus lambda
plt.plot(lambdas, J1)
plt.plot(lambdas, J2)
plt.xscale('log')
plt.xlabel('lambda')
plt.xlim((1e-4, 1e+4))
plt.legend(['J_1','J_2'])
plt.show()

# plot tradeoff curve
plt.plot(J1,J2)
plt.xlabel('J_1')
plt.ylabel('J_2')

# add (single objective) end points to trade-off curve
x1 = np.linalg.lstsq(As[0], bs[0], rcond=None)[0]
x2 = np.linalg.lstsq(As[1], bs[1], rcond=None)[0]
J1 = [np.linalg.norm(As[0]@x1 - bs[0])**2,
np.linalg.norm(As[0]@x2 - bs[0])**2]
J2 = [np.linalg.norm(As[1]@x1 - bs[1])**2,
np.linalg.norm(As[1]@x2 - bs[1])**2]
plt.scatter(J1, J2)
plt.show()
```

illustrates how different λ affect the solution of a multi-objective least squares problem. ■

Exercises (Go to Solutions)

For Exercises 1-2, for the given objectives J_i, solve the multi-objective least squares problem

$$J = \lambda_1 J_1 + \lambda_2 J_2 + \ldots + \lambda_k J_k,$$

where $J_i = \|A_i \boldsymbol{x} - \boldsymbol{b}_i\|^2$.

♠ 1. $A_1 \begin{bmatrix} 2 & 3 \\ 4 & 9 \end{bmatrix}, \boldsymbol{b}_1 = \begin{bmatrix} 5 \\ 11 \end{bmatrix}, \lambda_1 = 1;\ A_2 = \begin{bmatrix} 1 & 13 \end{bmatrix}, \boldsymbol{b}_2 = \begin{bmatrix} \frac{19}{3} \end{bmatrix}, \lambda_2 = \frac{1}{4}$

♠ 2. $A_1 = \begin{bmatrix} 6 & 10 \\ 12 & 8 \\ 14 & 11 \end{bmatrix}, \boldsymbol{b}_1 = \begin{bmatrix} 5 \\ 11 \\ 4 \end{bmatrix}, \lambda_1 = 1;\ A_2 = \begin{bmatrix} 6 & 50 \\ 2 & 15 \end{bmatrix}, \boldsymbol{b}_2 = \begin{bmatrix} \frac{7}{3} \\ 5 \end{bmatrix}, \lambda_2 = 36$

3. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/multileast.ipynb

"[Regular] is the wrong name often used for average." – Henry S. Haskins

4.8 Regularization

Even in the case where no secondary objective is present, beyond the main objective, of course, the use of multi-objective least squares has many applications that we explore here. The additional objectives $\lambda_2 J_2 + \ldots + \lambda_k J_k$ are called the **regularization** of the main objective J_1. In this situation, the weights λ_i will be very small, thus to not distract too much from J_1, but large enough to have a significant impact on the secondary objectives $J_2, \ldots, J_k$. We illustrate this with the following examples.

Example 4.8.1. Consider the affine transformation $T : \mathbb{R}^n \to \mathbb{R}^m$ given as $T(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$. Suppose that $\boldsymbol{y} \in \mathbb{R}^m$ is some target value. Then our main objective would be to find a vector $\boldsymbol{x} \in \mathbb{R}^n$ so that $T(\boldsymbol{x})$ is close to $\boldsymbol{y}$, that is, $J_1 = \|T(\boldsymbol{x}) - \boldsymbol{y}\|^2$ is minimal. For operational reasons, there may be a vector $\boldsymbol{z} \in \mathbb{R}^n$ for which $\boldsymbol{x}$ ought to be close to, that is, $J_2 = \|\boldsymbol{x} - \boldsymbol{z}\|^2 = \|I_n\boldsymbol{x} - \boldsymbol{z}\|^2$ is minimal. It could be that $\boldsymbol{z}$ is the current value of our direct variable $\boldsymbol{x}$ and adjusting from $\boldsymbol{z}$ to some new $\boldsymbol{x}$ comes with a cost.

An important special case when $\boldsymbol{z} = \boldsymbol{0}$ should be considered. Then our objective simplifies to $J_2 = \|\boldsymbol{x} - \boldsymbol{0}\|^2 = \|\boldsymbol{x}\|^2$, which means we want $\boldsymbol{x}$ to have small norm. This could be resemble to expect. After all, if each entry of $\boldsymbol{x}$ comes with a cost, then a shorter vector would be much cheaper for the operation with perhaps only a small decrease in quality.

For example, consider a cooling/heating system of a large complex. The temperature of m locations in the complex is recorded in a vector. Suppose the desired temperature is 72 F$^\circ\mathbf{1}$. Let $\boldsymbol{x}$ be the control we have over the cooling system from electricty and let $T(\boldsymbol{x})$ the resulting temperature in the m locations based upon this power usage and the resulting operation of the cooling equipment. Then the main objective is certainly to remain room temperature $J_1 = \|T(\boldsymbol{x}) - \boldsymbol{y}\|^2$. As electricity comes with a cost, keeping $\boldsymbol{x}$ small is also important. Therefore, the multi-objective becomes $J = \|T(\boldsymbol{x}) - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{x}\|^2$, where λ is selected to strike the right balance between maintaining comfort and minimizing cost. If $T(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$, then we would seek the least squares solutions to the linear system $\begin{bmatrix} A \\ \sqrt{\lambda} I_n \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} \boldsymbol{y} - \boldsymbol{b} \\ \boldsymbol{0} \end{bmatrix}$. ∎

We have seen that choosing too complicated models when trying to fit a model into data can lead to overfitting. This has the consequence that the model fits the training set very well but performing poorly on any new data set, such as the test set. Validation was a method to help avoid overfitting. Regularization can also be an effective tool to help avoid overfitting by placing more control on the model. Suppose we have considering the following regression model to fit to data:

$$g(\boldsymbol{x}) = c_2 f_1(\boldsymbol{x}) + c_2 f_2(\boldsymbol{x}) + \ldots + c_k f_k(\boldsymbol{x}).$$

Assuming the features f_i are chosen well, we could still overfit the data without regularization. Consider the following. If c_i is a very large coefficient, then even a small change in $f_i(\boldsymbol{x})$ would have a large affect to the product $c_i f_i(\boldsymbol{x})$ and thus have a large affect to the model. But if f_i is fixed as a function we will use in the model, then the fluxation of $f_i(\boldsymbol{x})$ comes from the variable $\boldsymbol{x}$ itself. Thus, for a large coefficient, even a small change of $\boldsymbol{x}$ could distort $g(\boldsymbol{x})$ because of $\boldsymbol{x}$. Therefore, g is better protected from this type of variation when the coefficients of g are small. This is how regularization aides with data fitting.

For *regularized data fitting*, our primary goal is still to minimize $J_1 = \|g(\boldsymbol{x}) - \boldsymbol{y}\|^2$, but we also have the secondary objective of keeping the coefficients c_i small. This means, collectively, that we want to minimize $J_2 = \lambda\|\boldsymbol{c}\|^2$. As one of the features of g is often the constant function, say the first one, $f_1(\boldsymbol{x}) = 1$, there is no need to keep its coefficient small. A change of $\boldsymbol{x}$ will have no affect on $f_1(\boldsymbol{x})$, so there is no need to keep

c_1. Therefore, we replace J_2 with the simpler objective $J_2 = \lambda\|\boldsymbol{c}_{2:k}\|^2$.[i] Then our multi-objective is

$$J = \|g(\boldsymbol{x}) - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{c}_{2:k}\|^2.$$

Example 4.8.2. Recall in Example 4.2.2 we interpolated a cubic polynomial $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ through a scatterplot of 100 points. This boiled down to solve the least squares problem $V\boldsymbol{c} = \boldsymbol{y}$, where V is the 100×4 Vandermonde matrix constructed from the x-coordinates $\boldsymbol{x}$, $\boldsymbol{y}$ is the 100-vector of y-coordinates, and $\boldsymbol{c}$ is the 4-vector of coefficients to be solved for. We will attempt this exercise again using regularization. For some $\lambda > 0$, we solve the multi-least squares problem $\begin{bmatrix} V \\ \sqrt{\lambda}I_4 \end{bmatrix} \boldsymbol{c} = \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{0} \end{bmatrix}$.

The following code can be used to solve this example:

```
import numpy as np
import matplotlib.pyplot as plt

def data_fit_regular(X, y, lamb):
    m,n = X.shape
    A = np.concatenate((np.array(X),np.sqrt(lamb)*np.identity(n)))
    b = np.concatenate((np.array(y),np.zeros(n)))
    return np.linalg.lstsq(A,b, rcond=None)[0]

points = np.array([[ 0.6346537948227042, -0.25053562659185047],
x = points[:,0]
y = points[:,1]

A = np.vander(x,3+1)
f = data_fit_regular(A,y,0.25)
g = np.linalg.lstsq(A,y, rcond=None)[0]

#Compute RMS error
cubics = [f,g]
for h in cubics:
    print(h, f"RMS error = {round(100*np.sqrt(np.mean((y-np.polyval(h,x))**2))/np.linalg.
    norm(y),2)}%,",
          f"coeff norm = {np.linalg.norm(h)}\n")

#Graph Polynoials
t = np.linspace(-1,1,num = 100)
for i in range(2):
    plt.subplot(1,2,i+1)
    plt.axis([-1,1,-0.7,0.7])
    plt.scatter(x,y)
    plt.plot(t,np.poly1d(cubics[i])(t), 'r')
plt.show()
```

[i] Of course, just like the constant function $f_1(\boldsymbol{x}) = 1$, it could be that a small change in $\boldsymbol{x}$ affects $f_i(\boldsymbol{x})$ and $f_j(\boldsymbol{x})$ differently. So, it might be more advantageous to separate some coefficients from each other. The worst case scenario would be $J_2 = \lambda_2|c_2|^2 + \ldots \lambda_k|c_k|^2$. As this level of care may be more complicated than necessary, we will resolve ourselves with $J_2 = \lambda\|\boldsymbol{c}\|^2$ for regularization purposes.

Exercises

(Go to Solutions)

For Exercises 1–1, interpolate the function $f(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ through the given points with a regularization coefficient λ.

1. $\lambda = 0.1$; $(-2, 231)$, $(-1, 19)$, $(0, 5)$, $(1, 9)$, $(2, 19)$, $(3, -169)$, $(-1.5, 100)$, $(-0.5, 20)$, $(0.5, 12)$, $(2.5, 0)$, $(3.5, -25)$

For Exercises 2–2, interpolate the function $f(x) = \dfrac{a_2x^2 + a_1x + a_0}{b_2x^2 + b_1x + 1}$ through the given points with a regularization coefficient λ.

2. $\lambda = 0.25$; $\left(-2, \frac{12}{11}\right)$, $\left(-1, \frac{3}{2}\right)$, $(0, 2)$, $(1, 0)$, $(2, 0)$, $\left(-1.5, \frac{10}{3}\right)$, $\left(-0.5, \frac{20}{15}\right)$, $\left(0.5, -\frac{12}{5}\right)$, $\left(2.5, -\frac{11}{3}\right)$, $\left(3.5, -\frac{25}{7}\right)$

3. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/regular.ipynb

"The shallow consider liberty a release from all law, from every constraint. The wise man sees in it, on the contrary, the potent Law of Laws." – Walt Whitman

4.9 Constrained Least Squares Problem

Up until now, we have gotten pretty good at setting up and solving least squares problems, and we have seen how these solve important optimization problems with the objective of minimizing $\|A\boldsymbol{x}-\boldsymbol{b}\|^2$. In practice, optimization problems ALWAYS, ALWAYS[i] come with a constraint, that is, some relation that restricts the possibilities. This is typically a consequence of a restriction on natural resources. Imagine then we have to consider the problem of minimizing $\|A\boldsymbol{x}-\boldsymbol{b}\|^2$ such that $C\boldsymbol{x}=\boldsymbol{d}$ for some constraining matrix C and constraining vector $\boldsymbol{d}$.

Now we kind of considered this problem before. Consider the multi-objective problem $J=\|A\boldsymbol{x}-\boldsymbol{b}\|^2+\lambda\|C\boldsymbol{x}-\boldsymbol{d}\|^2$. As have seen that the bigger and bigger λ becomes, the multi-objective J tended to stabilize toward some value. For each value λ, let $\widehat{\boldsymbol{x}}(\lambda)$ be the least squares solution to $\begin{bmatrix} A \\ \sqrt{\lambda}C \end{bmatrix}\boldsymbol{x}=\begin{bmatrix} \boldsymbol{b} \\ \sqrt{\lambda}\boldsymbol{d} \end{bmatrix}$.[ii] Let $\lim_{\lambda\to\infty}\widehat{\boldsymbol{x}}(\lambda)=\widehat{\boldsymbol{x}}$, assuming this limit exists. In that case, if $\lambda=\infty$, then $\|C\widehat{\boldsymbol{x}}-\boldsymbol{d}\|^2$ would have to be zero so that $J<\infty$.[iii] This happens only if $C\widehat{\boldsymbol{x}}=\boldsymbol{d}$, that is, $\widehat{\boldsymbol{x}}$ is an actual solution to $C\boldsymbol{x}=\boldsymbol{d}$ as opposed to just an approximation. Then $\lambda\|C\widehat{\boldsymbol{x}}-\boldsymbol{d}\|^2=0$, hence the multi-objective simplifies just to $J=\|A\widehat{\boldsymbol{x}}-\boldsymbol{b}\|^2$.

This honestly sounds like a lot of Calculus (it is), and there are many places where imprecise calculus assumptions can lead us astray (be warned). Fortunately, solving this constrained least squares problem can be encapsulated in a broader constrained optimization problem typically covered in multivariate calculus settings. In such settings, our problem would be solved using a technique known as *Lagrange multipliers*. Using the technique of Lagrange multipliers, one can deduce the exact formulas that we will see below. Fortunately, we can verify our solution technique without any calculus, and thus leave the derivation of the KKT matrix within the fog of multivariate calculus.

Theorem 4.9.1. *For $m\times n$ matrix A, $\boldsymbol{b}\in\mathbb{R}^m$, $p\times n$ matrix C, and $\boldsymbol{d}\in\mathbb{R}^p$, if $(\widehat{\boldsymbol{x}},\widehat{\boldsymbol{\lambda}})$ is a solution to the linear system*

$$\begin{bmatrix} 2A^\top A & C^\top \\ C & 0 \end{bmatrix}\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{bmatrix}=\begin{bmatrix} 2A^\top\boldsymbol{b} \\ \boldsymbol{d} \end{bmatrix}, \tag{4.9.1}$$

then $\widehat{\boldsymbol{x}}$ is a solution to the constrained least squares problem:

$$\begin{aligned} &\text{minimize } \|A\boldsymbol{x}-\boldsymbol{b}\|^2, \\ &\text{subject to } C\boldsymbol{x}=\boldsymbol{d}. \end{aligned}$$

The $(n+p)\times(n+p)$ matrix $\begin{bmatrix} 2A^\top A & C^\top \\ C & 0 \end{bmatrix}$ is called the **Karush-Kuhn-Tucker (KKT) matrix** associated to this constrained least squares problem. The KKT equations are the generalization of the normal equations for a least squares problem without a constraint.

Proof. Let $(\widehat{\boldsymbol{x}},\widehat{\boldsymbol{\lambda}})$ be a linear solution to (4.9.1). So, $2A^\top A\widehat{\boldsymbol{x}}+C^\top\widehat{\boldsymbol{\lambda}}=2A^\top\boldsymbol{b}$ and $C\widehat{\boldsymbol{x}}=\boldsymbol{d}$. Thus, the constraint aspect of this least squares problem is satisfied. We also see that $C^\top\widehat{\boldsymbol{\lambda}}=2A^\top\boldsymbol{b}-2A^\top A\widehat{\boldsymbol{x}}=2A^\top(\boldsymbol{b}-A\widehat{\boldsymbol{x}})$ Suppose that $\boldsymbol{x}$ is some other vector such that $C\boldsymbol{x}=\boldsymbol{d}$. Then $C(\widehat{\boldsymbol{x}}-\boldsymbol{x})=C\widehat{\boldsymbol{x}}-C\boldsymbol{x}=\boldsymbol{d}-\boldsymbol{d}=\boldsymbol{0}$. Next, consider

[i] ALWAYS

[ii] Assuming the linear system has a unique least squares solution. Even if there are multiple least squares solution, the function $\widehat{\boldsymbol{x}}(\lambda)$ can still be constructed to be continuous.

[iii] Here we are banking the assumption that $\infty\cdot 0=0$, although, in general, this indeterminant could be anything.

$$2(A\boldsymbol{x}-A\widehat{\boldsymbol{x}})\cdot(\boldsymbol{b}-A\widehat{\boldsymbol{x}}) = 2(A(\boldsymbol{x}-\widehat{\boldsymbol{x}}))\cdot(\boldsymbol{b}-A\widehat{\boldsymbol{x}}) = 2(\boldsymbol{x}-\widehat{\boldsymbol{x}})\cdot A^\top(\boldsymbol{b}-A\widehat{\boldsymbol{x}}) = (\boldsymbol{x}-\widehat{\boldsymbol{x}})\cdot C^\top\widehat{\boldsymbol{\lambda}} = (C(\boldsymbol{x}-\widehat{\boldsymbol{x}}))\cdot\widehat{\boldsymbol{\lambda}} = \boldsymbol{0}\cdot\widehat{\boldsymbol{\lambda}} = \boldsymbol{0},$$

that is, $A\boldsymbol{x} - A\widehat{\boldsymbol{x}}$ and $\boldsymbol{b} - A\widehat{\boldsymbol{x}}$ are orthogonal. Consider then the objective

$$\|A\boldsymbol{x} - \boldsymbol{b}\|^2 = \|(A\boldsymbol{x} - A\widehat{\boldsymbol{x}}) + (A\widehat{\boldsymbol{x}} - \boldsymbol{b})\|^2 = \|A\boldsymbol{x} - A\widehat{\boldsymbol{x}}\|^2 + \|A\widehat{\boldsymbol{x}} - \boldsymbol{b}\|^2 \geq \|A\widehat{\boldsymbol{x}} - \boldsymbol{b}\|^2.$$

Therefore, $\widehat{\boldsymbol{x}}$ minimizes the objective among linear solutions to $C\boldsymbol{x} = \boldsymbol{d}$. □

In the presence of multiple constraints:

$$\begin{aligned} \text{minimize } & \|A\boldsymbol{x} - \boldsymbol{b}\|^2, \\ \text{subject to } & C_1\boldsymbol{x} = \boldsymbol{d}_1, \\ & C_2\boldsymbol{x} = \boldsymbol{d}_2, \\ & \vdots \\ & C_k\boldsymbol{x} = \boldsymbol{d}_k, \end{aligned}$$

we can consolidate all the constraints into one linear equation: $C\boldsymbol{x} = \boldsymbol{d}$, where $C = \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_k \end{bmatrix}$ and $\boldsymbol{d} = \begin{bmatrix} \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \\ \vdots \\ \boldsymbol{d}_k \end{bmatrix}$.

A special constrained least squares problem is when we seek to find the smallest linear solution to $A\boldsymbol{x} = \boldsymbol{b}$. This linear equation becomes our constraint, and we need to minimize that objective $\|\boldsymbol{x}\|^2$. For this least-norm problem, our KKT equations become

$$\begin{bmatrix} 2I & A^\top \\ A & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{b} \end{bmatrix}.$$

If $(\widehat{\boldsymbol{x}}, \widehat{\boldsymbol{\lambda}})$ is a solution to the KKT equations, then $2\widehat{\boldsymbol{x}} + A^\top\widehat{\boldsymbol{\lambda}} = \boldsymbol{0}$ or $\widehat{\boldsymbol{x}} = -\frac{1}{2}A^\top\widehat{\boldsymbol{\lambda}}$. Multiplying both sides on the left by A gives $A\widehat{\boldsymbol{x}} = -\frac{1}{2}AA^\top\boldsymbol{\lambda}$ or $\boldsymbol{b} = -\frac{1}{2}AA^\top\boldsymbol{\lambda}$. If A is right-invertible, then $\boldsymbol{\lambda} = -2(AA^\top)^{-1}\boldsymbol{b}$ and

$$\widehat{\boldsymbol{x}} = A^\top(AA^\top)^{-1}\boldsymbol{b} = A^\dagger\boldsymbol{b}. \tag{4.9.2}$$

Example 4.9.2. Suppose a company is analyzing its advertisement strategy. Suppose the company advertises to m demographic groups or audiences. The company has a desired amount of views or impressions that it wants members of those groups to experience. Say that vector $\boldsymbol{t} = (t_1, t_2, \ldots, t_m)$ is the *target vector* with regret to these impressions To reach these audiences, the company purchases advertisements in n different channels or media (maybe radio ads, TV ads, web ads, print ads, etc). Let $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ be the cost vector for advertising in these different channels, where c_j is the amount the company pays to channel j for advertisement. Let V_{ij} denote the average number of views produces for group i in channel j, measured in views per dollar spent. These values V_{ij} have been calculated by the company's market research division. Let $V = \begin{bmatrix} V_{ij} \end{bmatrix}$ be the associated $m \times n$ *viewing matrix*. For the company to acquire the targeted number of views $\boldsymbol{t}$, they must solve the least squares problem $V\boldsymbol{c} = \boldsymbol{t}$. This is a least squares problem because this linear system may be inconsistent as there may be no cost vector that perfectly produces the targeted average number of views in all audiences. Of course, a least squares solution gives the company the best marketing strategy to meet its target. Furthermore, if the company has an advertising budget, then there would be a constraint on this least squares problem of $\boldsymbol{1}^\top\boldsymbol{c} = B$.

Recall Example 4.1.7 where a company wants to advertise to $m = 5$ audiences via $n = 4$ advertising channels. Suppose

$$V = \begin{bmatrix} 2.0 & 1.1 & 0.1 & 3.2 \\ 1.2 & 2.2 & 1.3 & 0.2 \\ 0.5 & 1.0 & 3.1 & 2.7 \\ 3.1 & 0.4 & 2.4 & 1.2 \\ 1.1 & 1.1 & 2.2 & 2.3 \end{bmatrix},$$

where V_{ij} is measured in 1000 views per dollar (on average). Suppose that the company wants each of the audiences to have 600,000 views in the next advertising cycle, that is, the target vector is $\boldsymbol{t} = 600\mathbf{1}$, where t_i is the amount of views the company wants group i to see. Let $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ be the cost vector for advertising in these different channels, where c_j is the amount the company pays to channel j for advertisement. Solving the least squares problem using, we find that $\boldsymbol{c} = (\$95.02, \$176.77, \$65.20, \$65.69)$. With this cost vector, the views will be $V\boldsymbol{c} = (601, 601, 604, 601, 594)$, which gives a relative RMS error of 0.57% of the target impressions (near perfect). This will cost the company $\mathbf{1} \cdot \boldsymbol{c} = \$402.68$ in advertisement.

Suppose that the company has a budget on advertisement costs, say $B = \$375$ per cycle. While we still want to minimize $\|V\boldsymbol{c} - \boldsymbol{t}\|^2$, we must do this within the constraint that $\mathbf{1} \cdot \boldsymbol{c} = B$. Solving this constrained least squares problem gives instead $\boldsymbol{c} = (\$90.13, \$148.11, \$67.77, \$68.99)$. With this cost vector, the views will be $V\boldsymbol{c} = (571, 536, 590, 584, 570)$, which gives a relative RMS error of 5.89% of the target impressions. A bit worst than the unconstrained least squares solution, but still very close to the target views and within budget.

The following code can be used to solve this example:

```
import numpy as np
V = np.array([[2.0,1.1,0.1,3.2],
              [1.2,2.2,1.3,0.2],
              [0.5,1.0,3.1,2.7],
              [3.1,0.4,2.4,1.2],
              [1.1,1.1,2.2,2.3]])
t = 600*np.ones(5) #target impressions
B = 375 #Budget

print("Target Impressions in each Demographic Group =\n", t, "\n")

c = np.linalg.lstsq(V,t, rcond=None)[0]
print("No Budget: Cost = $" + str(round(sum(c),2))+"\n")
print("Spending in each Channel = $", [round(cost,2) for cost in c], "\n")
print("Expected Impressions in each Demographic Group =\n", [int(round(impress, 0)) for
    impress in V@c], "\n")
print("Relative RMS Error = " + str(round(100*(np.linalg.norm(V@c - t)/np.linalg.norm(t)),2)
    ) + "%\n\n")

c = lstsq_constraint(V, t, np.ones(4), B)
print("Fixed Budget: Cost = $" + str(B) +"\n")
print("Spending in each Channel = $", [round(cost,2) for cost in c], "\n")
print("Expected Impressions in each Demographic Group =\n", [int(round(impress, 0)) for
    impress in V@c], "\n")
print("Relative RMS Error = " + str(round(100*(np.linalg.norm(V@c - t)/np.linalg.norm(t)),2)
    ) + "%")
```

■

Example 4.9.3. We previously have used the Vandermonde matrix to solve data fitting applications involving polynomials. We have also considered data fitting problems involving piecewise linear functions. We now will combine these techniques. Suppose we wish to model a scatterplot $\{(x_1, y_1), \ldots, (x_M, y_M)\}$ using the function $g : \mathbb{R} \to \mathbb{R}$ given by

$$g(x) = \begin{cases} a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n, & x \leq t \\ b_0 + b_1 x + b_2 x^2 + \ldots + b_m x^m, & x \geq t, \end{cases}$$

for some switching number $x = t$. Call the first and second pieces of our model f_1 and f_2, respectively. Let $\boldsymbol{x} = (x_1, \ldots, x_N, x_{N+1}, \ldots, x_M)$ and $\boldsymbol{y} = (y_1, \ldots, y_M)$, where $x_i \leq c$ if and only if $i \leq N$. If

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^n & 0 & 0 & 0 & \ldots & 0 \\ 1 & x_2 & x_2^2 & \ldots & x_2^n & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \ldots & x_N^n & 0 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 & x_{N+1} & x_{N+1}^2 & \ldots & x_{N+1}^m \\ 0 & 0 & 0 & \ldots & 0 & 1 & x_{N+2} & x_{N+2}^2 & \ldots & x_{N+2}^m \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 1 & x_M & x_M^2 & \ldots & x_M^m \end{bmatrix},$$

where the top left corner of the matrix is an $N \times n$ Vandermonde, the bottom right corner is an $(M-N) \times m$ Vandermonde matrix, and the other two corners are zero matrices, then data fitting requires we solve the least square objective $X\boldsymbol{c} = \boldsymbol{y}$, where $\boldsymbol{c} = (a_0, \ldots, a_n, b_0, \ldots, b_m)$. To improve our model, we first require that there is no jump discontinuity at the switching number $x = t$, that is, when the function transitions between the two pieces we will not allow for a jump. We then require that $f_1(t) = f_2(t)$, or

$$a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n - b_0 - b_1 t - b_2 t^2 - \ldots - b_m t^m = 0.$$

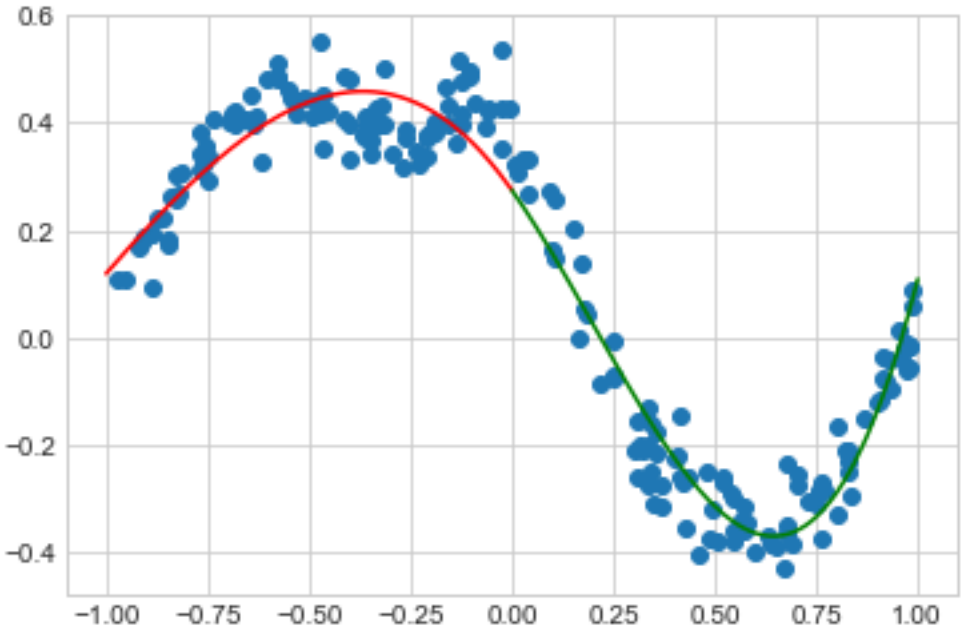
Second, we require that there is a smooth transition, that is, when the function transitions between the two pieces we will not allow for a cusp. We then require that $f_1'(t) = f_2'(t)$, or

$$a_1 + 2a_2 t + \ldots + n a_n t^{n-1} - b_1 - 2b_2 t - \ldots - m b_m t^{m-1} = 0.$$

We can require agreement on higher derivatives too, if desired, but a smooth transition is often suffiicient in practice. If

$$D = \begin{bmatrix} 1 & t & t^2 & \ldots & t^n & -1 & -t & -t^2 & \ldots & -t^m \\ 0 & 1 & 2t & \ldots & nt^{n-1} & 0 & -1 & -2t & \ldots & -mt^{m-1} \end{bmatrix},$$

then g will be a smooth, continuous function only if $\boldsymbol{c}$ satisfies the contraint $D\boldsymbol{c} = \boldsymbol{0}$. Solving this constrained least squares problem provides a smooth, continuous piecewise polynomial function to model the data. This method can be generalized to involve more than two pieces.



The following code can be used to solve this example:

```
import numpy as np
n = 3 #degree of polyonmial
M = 100 #number of data points per half
xleft = np.random.random(M) - 1
xright = np.random.random(M)
x = np.hstack([xleft, xright])
y = np.power(x,3) - x + 0.4/(1+25*np.power(x,2)) + 0.05*np.random.normal(size = 2*M)

A = np.block([[np.vander(xleft,n+1), np.zeros((M,n+1))],
              [np.zeros((M,n+1)), np.vander(xright,n+1)]])
D = np.block([[np.zeros(n), 1, np.zeros(n), -1],
              [np.zeros(n-1), 1, 0, np.zeros(n-1), -1, 0]])
coeffs = lstsq_constraint(A,y,D,np.zeros(2))

# Graphing
import matplotlib.pyplot as plt
xpl_left = np.linspace(-1, 0, 2*M)
ypl_left = np.vander(xpl_left, n+1) @ coeffs[:n+1]
xpl_right = np.linspace(0, 1, 2*M)
ypl_right = np.vander(xpl_right, n+1) @ coeffs[n+1:]
plt.scatter(x, y)
plt.plot(xpl_left, ypl_left, 'red')
plt.plot(xpl_right, ypl_right, 'green')
plt.show()
```

Python Programming

The following code can be used to solve a constrained least squares problem:

```
import numpy as np

def lstsq_constraint(A, b, C, d):
    ATA = A.T @ A
    CT = (C.T).reshape((len(ATA), -1))
    n = len(CT[0])
    KKT = np.block([[2*ATA, CT], [CT.T, np.zeros((n, n))]])
    KKTb = np.hstack((2*A.T @ b, d))
    return np.linalg.lstsq(KKT, KKTb, rcond=None)[0][:-n]
```

Exercises

(Go to Solutions)

For Exercises 1-2, solve the constrained least squares problem $A\boldsymbol{x} = \boldsymbol{b}$ subject to $C\boldsymbol{x} = \boldsymbol{d}$

♠ 1. $A = \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$, $C = \begin{bmatrix} 3 & -1 \end{bmatrix}$, $\boldsymbol{d} = \boldsymbol{0}$

2. $A = \begin{bmatrix} 4 & 1 \\ 2 & -2 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $C = \begin{bmatrix} 1 & -3 \end{bmatrix}$, $\boldsymbol{d} = \boldsymbol{0}$

3. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/constraint.ipynb

"You cannot control what happens to you, but you can control your attitude toward what happens to you, and in that, you will be mastering change rather than allowing it to master you." – Brian Tracy

4.10 Control of Linear Dynamical Systems

4.10.1 Linear Quadratic Control

Consider a linear dynamical system

$$\boldsymbol{x}_{t+1} = A_t\boldsymbol{x}_t + B_t\boldsymbol{u}_t,$$

with state n-vectors $\boldsymbol{x}_t$, control m-vectors $\boldsymbol{u}_t$, and transition matrices A_t and B_t. The term *control vector* suggests these are quantites that we can directly manipulate or choose. As such, we can choose $\boldsymbol{u}_t$ in the Markov chain to move $\boldsymbol{x}_{t+1}$ toward some desired state. Likewise, consider the situation where the p-vector $\boldsymbol{y}_t$ is given by

$$\boldsymbol{y}_t = C_t\boldsymbol{x}_t.$$

Thus, $\boldsymbol{u}_t$ is the input vector for $\boldsymbol{x}_{t+1}$ and $\boldsymbol{y}_t$ is the output vector for $\boldsymbol{x}_t$. In practice, y_t is a deviation between the actual output and desired ones, such as the path of a air- or space-craft. Thus, $\boldsymbol{x}_t$ needs to be controls in order to keep $\boldsymbol{y}_t$ small. Likewise, $\boldsymbol{u}_t$ is a deviation between the current input and desired ones which would control $\boldsymbol{x}_t$ properly. Hence, we shrink $\boldsymbol{u}_t$ to reduce $\boldsymbol{y}_{t+1}$ indirectly via the state vector $\boldsymbol{x}_{t+1}$. **Linear Quadratic (Optimal) Control**[i] is exactly this process, choosing the input vector $\boldsymbol{u}_t$ in order to control the output vector $\boldsymbol{y}_{t+1}$ thus minimizing the deviations between $\boldsymbol{y}_t$ and some desired output vector respective to the dynamical system. In a linear quadratic control (LQ) problem, we choose $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T$ in order to minimize the two objectives:

$$J_1 = \|\boldsymbol{y}_0\|^2 + \|\boldsymbol{y}_1\|^2 + \ldots + \|\boldsymbol{y}_T\|^2 = \|C_0\boldsymbol{x}_0\|^2 + \|C_1\boldsymbol{x}_1\|^2 + \ldots + \|C_T\boldsymbol{x}_T\|^2,$$

$$J_2 = \|\boldsymbol{u}_0\|^2 + \|\boldsymbol{u}_1\|^2 + \ldots + \|\boldsymbol{u}_{T-1}\|^2.$$

Then the multi-objective $J = J_1 + \rho J_2$ is the goal to be minimized. The state vectors $\boldsymbol{x}_t$ are subject to the constraints

$$\boldsymbol{x}_{t+1} = A_t\boldsymbol{x}_t + B_t\boldsymbol{u}_t$$

for all t, as well as the constraints $\boldsymbol{x}_0 = \boldsymbol{a}$ and $\boldsymbol{x}_T = \boldsymbol{b}$, for some initial state vector $\boldsymbol{a}$ and for some terminal desired target vector $\boldsymbol{b}$.

Solving a linear quadratic control problem comes down to reformulating the problem as a constrained least squares problem. We seek to find an $(n(T+1)+mT)$-vector $\boldsymbol{z}$:

$$\boldsymbol{z} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T, \boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{T-1})$$

which minimizes the objective $J = \|A\boldsymbol{z}\|^2$ where A is an $(p(T+1)+mT) \times (n(T+1)+mT)$ block diagonal matrix of the form

$$A = \left[\begin{array}{ccc|ccc} C_0 & & & & & \\ & C_1 & & & & \\ & & \ddots & & & \\ & & C_T & & & \\ \hline & & & \sqrt{\rho}I_m & & \\ & & & & \ddots & \\ & & & & & \sqrt{\rho}I_m \end{array}\right]$$

[i]The term *control* is obvious in its meaning in this context. The term *linear* is describing the linear dynamical system. Finally, the term *quadratic* here is referring to the objective we wish to minimize, which is a sum of squares.

Of course, $J = \|A\boldsymbol{z}\|^2 = J_1 + \rho J_2$, that is, the new objective agrees with the objective for linear quadratic control. The dynamic constraints will then reformulate as $C\boldsymbol{z} = \boldsymbol{d}$, where C is an $n(T+2) \times (n(T+1)+mT)$ matrix and $\boldsymbol{d}$ is an $n(T+2)$-vector of the form

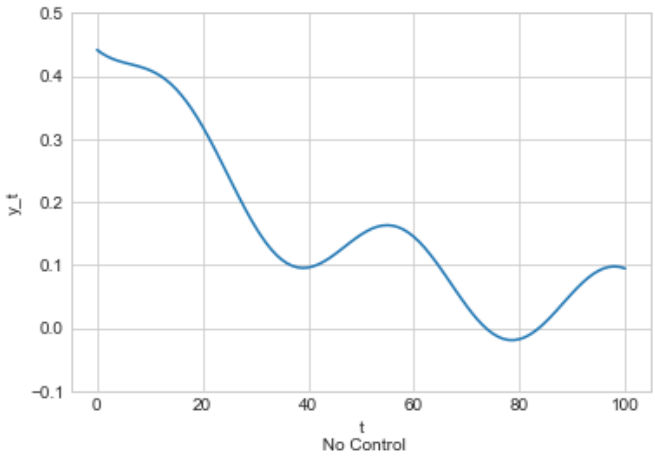
$$C = \left[\begin{array}{ccccc|ccc} A_0 & -I_n & & & & B_0 & & \\ & A_1 & -I_n & & & & B_1 & \\ & & \ddots & \ddots & & & & \ddots \\ & & & A_{T-1} & -I_n & & & B_{T-1} \\ \hline I_n & & & & & & & \\ & & & & I_n & & & \end{array}\right], \qquad \boldsymbol{d} = \left[\begin{array}{c} \boldsymbol{0} \\ \boldsymbol{0} \\ \vdots \\ \boldsymbol{0} \\ \hline \boldsymbol{a} \\ \boldsymbol{b} \end{array}\right].$$

Example 4.10.1. We consider the time-invariant linear dynamical system with

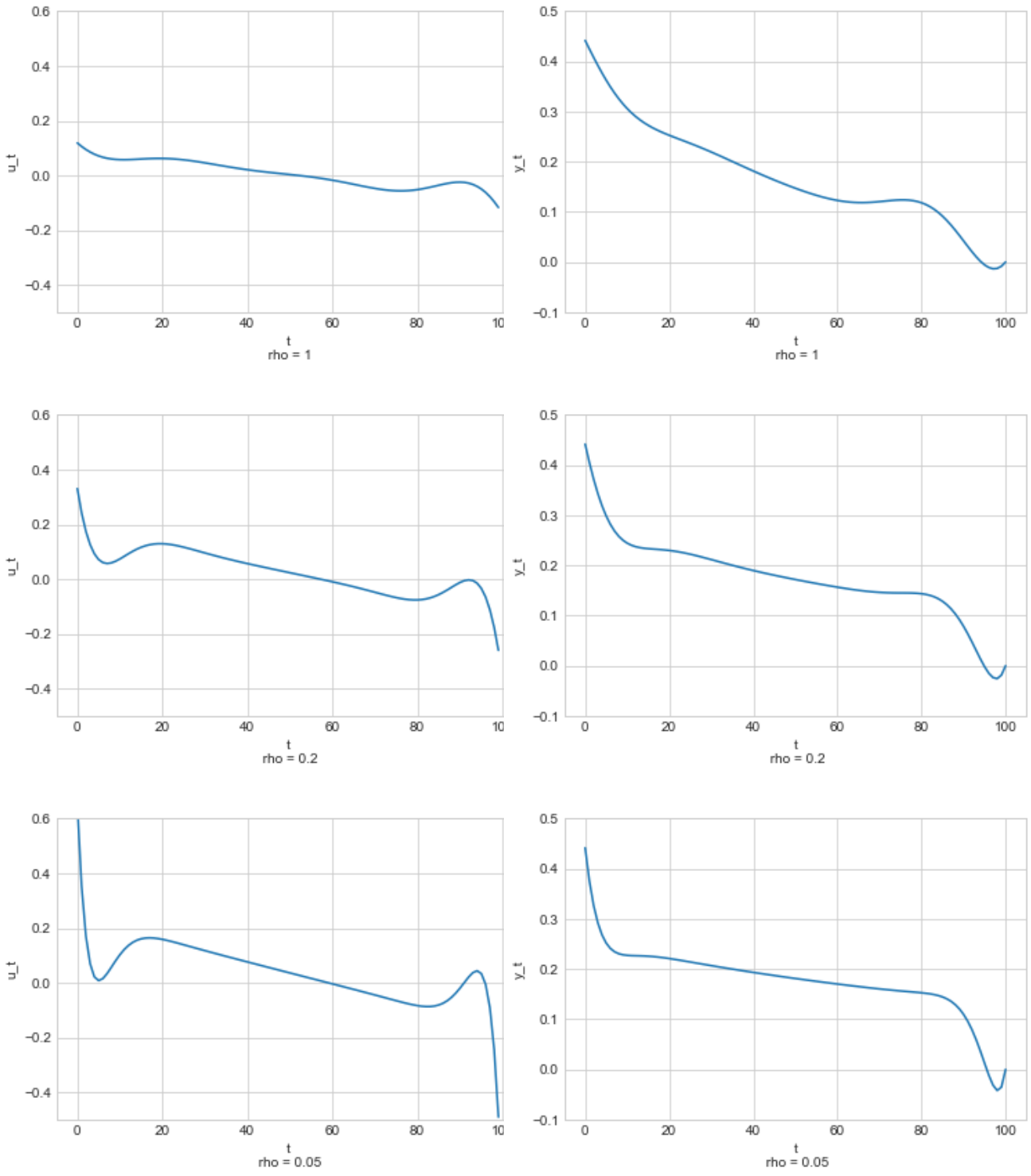
$$A = \begin{bmatrix} 0.855 & 1.161 & 0.667 \\ 0.015 & 1.073 & 0.053 \\ -0.084 & 0.059 & 1.022 \end{bmatrix}, \quad B = \begin{bmatrix} -0.076 \\ -0.139 \\ 0.342 \end{bmatrix},$$

$$C = \begin{bmatrix} 0.218 & -3.597 & -1.683 \end{bmatrix}, \quad \boldsymbol{a} = (0.496, -0.745, 1.394), \quad \boldsymbol{b} = \boldsymbol{0}, \quad T = 100.$$

Then $\boldsymbol{y}_t = C\boldsymbol{x}_t$ is a scalar. If simulate the dynamical system with no control, that is, $\boldsymbol{x}_0 = \boldsymbol{a}$, $\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t$, and $\boldsymbol{y}_t = C\boldsymbol{x}_t$, then we can plot the trajectory of $\boldsymbol{y}_t$ as a variable of time on the graph illustrated to the right. While generally decreasing over time, it does not end with $\boldsymbol{y}_T = C\boldsymbol{0} = 0$, nor is the oscillating behavior desirable.



If we apply linear quadratic control, we need to solve the constrained least squares problem $A\boldsymbol{z} = \boldsymbol{0}$ subject to the constraint $C\boldsymbol{z} = \boldsymbol{d}$, as above. In this case, $n = 3$, $m = p = 1$, which implies that $(p(T+1) + mT) = 1(100+1) + 1(100) = 201$, $n(T+1) + mT = 3(100+1) + 1(100) = 403$, and $n(T+2) = 3(100+2) = 306$. Hence, $\boldsymbol{z}$ is a 403-vector, A is a 201×403 matrix, C is a 306×403 matrix, and $\boldsymbol{d}$ is a 306-vector. Clearly, this calculation will be conducted on Python. See the end of the section for the implementation of the `lqr(A,B,C,a,b,T,rho)` function. The calculation does depend on the weight ρ. For the sake of this example, we include the calculations for $\rho = 1, 0.2, 0.05$ below.



Validation methods similar to those discussed in this chapter can be used to determine an optimal choice for ρ, but we will not discuss them in detail here.

The following code can be used to solve this example:

```
import matplotlib.pyplot as plt
## No Control ##
Xno = np.zeros((T+1, len(a)))  #initialize Xno
Xno[0] = a
for k in range(T):             #simulation without control
    Xno[k+1] = A @ Xno[k]
```

```
# plot output vector y_t (there is no control vector u_t)
Yno = C @ Xno.T
plt.plot(np.arange(T+1),Yno)
plt.xlabel("t\n No Control")
plt.ylabel("y_t")
plt.ylim(-0.1,0.5)
plt.show()

## Linear Quadratic Control ##
rhos = [1, 0.2, 0.05]
#initialize sequences X (states), U (input/control), Y (output)
X,U,Y = np.zeros((len(rhos),T+1,3)),np.zeros((len(rhos),T,1)),np.zeros((len(rhos),T+1))
for k in range(len(rhos)):
    X[k],U[k],Y[k] = lqr(A,B,C,a,b,T,rhos[k])
    # plot input vectors u_t
    plt.plot(np.arange(T),U[k])
    plt.xlabel('t\n rho = ' + str(rhos[k]))
    plt.ylabel('u_t')
    plt.ylim(-0.5,0.6)
    plt.show()
    # plot output vectors y_t
    plt.plot(np.arange(T+1),Y[k])
    plt.xlabel('t\n rho = ' + str(rhos[k]))
    plt.ylabel('y_t')
    plt.ylim(-0.1,0.5)
    plt.show()
```

4.10.2 State Feedback Control

We have seen previously, in Section 3.7.3, that the simplifest model for control is linear state feedback control, that is,

$$\boldsymbol{u}_t = K_t \boldsymbol{x}_t,$$

that is, the control vector $\boldsymbol{u}_t$ is a linear function of the current state vector. Then the dynamic equation simplifies toward

$$\boldsymbol{x}_{t+1} = A_t \boldsymbol{x}_t + B_t \boldsymbol{u}_t = A_t \boldsymbol{x}_t + B_t K_t \boldsymbol{x}_t = (A_t + B_t K_t)\boldsymbol{x}_t.$$

Thus, obtaining the state feedback gain matrices K_t is highly desirable. Now solving a linear quadratic control problem actually provides a way to contruct the state feedback gain matrices. After all, given a solution vector

$$\widehat{\boldsymbol{z}} = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_t, \ldots, \boldsymbol{x}_T, \boldsymbol{u}_0, \ldots, \boldsymbol{u}_t, \ldots, \boldsymbol{u}_{T-1}),$$

this vector induces a function assigning to each $\boldsymbol{x}_t$ some input vector $\boldsymbol{u}_t$. If this function is linear, then it must be represented by a matrix, namely K_t. If it is not linear, then we are still offered a linear approximation.

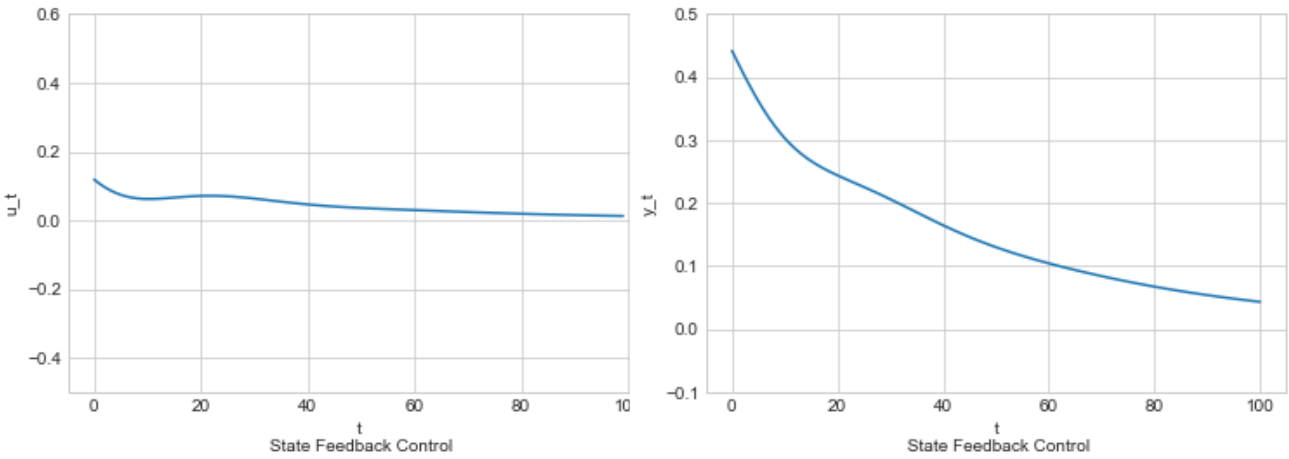
Let us consider the time-invariant case, $\boldsymbol{u}_t = K\boldsymbol{x}_t$. In this case, we have $\boldsymbol{u}_0 = K\boldsymbol{x}_0$. As the linear quadratic problem treats $\boldsymbol{x}_0 = \boldsymbol{a}$ as a constraint, if we treat the other constraint $\boldsymbol{x}_T = \boldsymbol{0}$ as a constant and allow $\boldsymbol{a}$ to vary, then we can easily extract the function relationship on $\boldsymbol{x}_0 \mapsto \boldsymbol{u}_0$. To construct the state feedback gain matrix K, we solve the linear quadratic problem repeatedly using the new constraint $\boldsymbol{x}_0 = \boldsymbol{e}_i$ for $1 \leq i \leq n$. Let $\boldsymbol{u}_0^{(i)}$ denote the input vector produced by the constraint $\boldsymbol{x}_0 = \boldsymbol{e}_i$. Then

$$K = \begin{bmatrix} \boldsymbol{u}_0^{(1)} & \boldsymbol{u}_0^{(2)} & \ldots & \boldsymbol{u}_0^{(n)} \end{bmatrix}.$$

Example 4.10.2. Continuing Example 4.10.1, using $\rho = 1$, we solve the linear quadratic control problem using $\boldsymbol{x}_0 = \boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3$ to produce $\boldsymbol{u}_0^{(1)} = 0.308, \boldsymbol{u}_0^{(2)} = -2.659, \boldsymbol{u}_0^{(3)} = -1.446$, respectively. Hence, the state feedback matrix K is given by

$$K = \begin{bmatrix} 0.308 & -2.659 & -1.446 \end{bmatrix}.$$

We re-run the simulation using K (the state feedback control simulation) this time and compare it to the linear quadratic simulation for $\rho = 1$. See below.



The following code can be used to solve this example:

```
#Linear State Feedback Control
K = np.zeros(3)              #initialize/compute state-feedback gain matrix
for  i in range(3):
    X,U,Y = lqr(A,B,C,np.eye(1,3,i)[0],b,T,1)
    K[i] = U[0]
print(K)

ABK = A + np.outer(B,K)     #form matrix A + BK, outer product used to avoid having to
    reshape vectors as matrices
Xst = np.zeros((T+1, 3))    #initialize Xst
Xst[0] = a
for k in range(T):          #simulation with state-feedback control
    Xst[k+1] = ABK @ Xst[k]
yst = C @ Xst.T
# plot input vectors u_t
plt.plot(np.arange(T), [K @ x for x in Xst[:-1]])
plt.xlabel("t\n State Feedback Control")
plt.ylabel('u_t')
plt.ylim(-0.5,0.6)
plt.show()
 # plot output vectors y_t
plt.plot(np.arange(T+1), yst)
plt.xlabel("t\n State Feedback Control")
plt.ylabel("y_t")
plt.ylim(-0.1,0.5)
plt.show()
```

Python Programming

The following code can be used to solve a linear quadratic control problem:

```
import numpy as np

def lstsq_constraint(A, b, C, d):   #|Ax-b|^2 is the objective to minimize
    ATA = A.T @ A                   #subject to the constraint Cx=d
    CT = (C.T).reshape((len(ATA), -1))
    p = len(CT[0])
    KKT = np.block([[2*ATA, CT], [CT.T, np.zeros((p, p))]])
    KKTb = np.hstack((2*A.T @ b, d))
    return np.linalg.solve(KKT, KKTb)[:-p]

#Linear Quadratic Regulator
def lqr(A, B, C, a, b, T, rho):
    n = len(A)
    Bmtx,Cmtx = B.reshape((n,-1)),C.reshape((-1,n))
    m,p = len(Bmtx[0]),len(Cmtx)
    Ablock = np.block([
        [np.kron(np.eye(T+1),Cmtx),np.zeros((p*(T+1),m*T))],
        [np.zeros((m*T,n*(T+1))),np.sqrt(rho)*np.eye(m*T)]])
    Cblock = np.block([
            [np.kron(np.eye(T,T+1),A) - np.kron(np.eye(T,T+1,1),np.eye(n)),
                  np.kron(np.eye(T),Bmtx)],
            [np.kron(np.vstack([np.eye(1,T+1,0),np.eye(1,T+1,T)]),np.eye(n)),
                  np.zeros((2*n,m*T))]])
    dblock = np.concatenate((np.zeros(n*T),a,b))
    z = lstsq_constraint(Ablock,np.zeros(len(Ablock)),Cblock,dblock)

    x = np.array([z[i*n:(i+1)*n] for i in range(T+1)])
    u = np.array([z[n*(T+1)+(i)*m: n*(T+1)+(i+1)*m] for i in range(T)])
    y = np.array([C @ xt for xt in x])
    return x, u, y
```

Exercises

(Go to Solutions)

1. Complete all of the Python-based exercises found at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/lqr.ipynb

Python Projects

1. Recall in project 2.1 that we were tasked with finding a function of the form

$$g(x) = \frac{1}{2} + c_1 \sin(x) + c_2 \sin(3x) + c_3 \sin(5x) + c_4 \sin(7x) + \ldots + c_{25} \sin(49x)$$

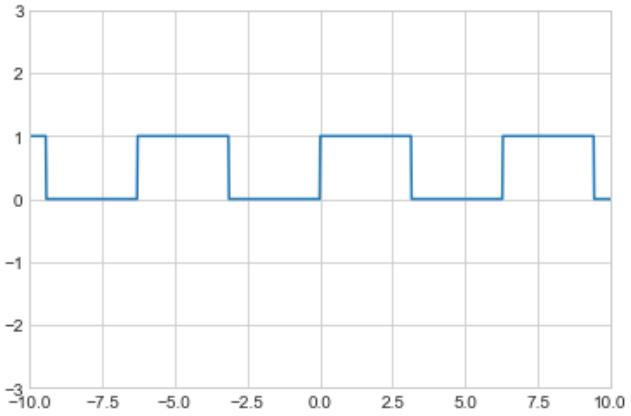
to interpolate the square-wave function

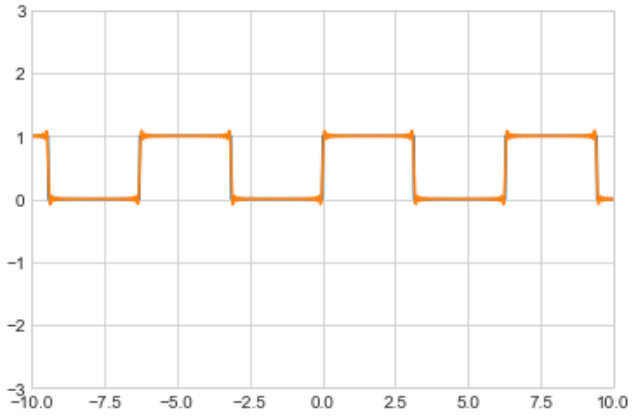
$$f(x) = \begin{cases} 1 & 0 \leq x < \pi \\ 0 & \pi \leq x < 2\pi \\ f(x + 2\pi) & = f(x) \end{cases}.$$

The graph of the square-wave is illustrated below left. Unlike your first attempt at this problem, you will need to use the least squares method to find your interpolation this time, such as explained in Section 4.1. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/fourier.py

(a) Choose 100 points on the graph of f and set up a linear system that will fit $g(x)$ to these 100 points.

(b) Graph your Fourier approximation on the same screen as the graph of the square-wave to determine how good your estimate is. Below right you will find the graph of the Fourier approximation of f using the Calculus-method alluded to in project 2.1. You will seek to produce such a plot, although I do not expect yours to be as good as this one, as you will NOT be using the Calculus-method here. But the least squares method will be a huge improvement from the25×25 linear system you solved inproject 2.1. Compare the two methods.





2. Professors at Southern State University are evaluated for each course they teach by their students on a five-point scale (with 5 being a perfect evaluation). A math professor conjectures that class ranking, student average GPA, and class size seem to positively correlate to student evaluations, such as explained in Section 4.3.1. To test his hypothesis, he collects this data for the classes he has taught the last few years. Note that GPA is on a 4-point scale (where 4.00 is straight A's for the whole class), and class ranking 1-4 where 1 is for freshmen classes, 2 is for sophomore classes, etc. The regression model he uses is $y = \boldsymbol{a} \cdot \boldsymbol{x} + b$, where y is the evaluation prediction, x_1 is the class ranking, x_2 is the average GPA of the course, and x_3 is the number of students in the class,

$$\boldsymbol{a} = (0.07329327, -0.03836403, -0.00551211),$$

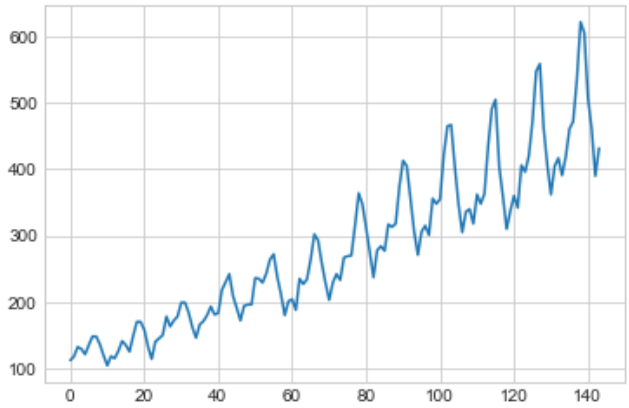
and $b = 4.5567198$. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/SouthernStateUniversity.py

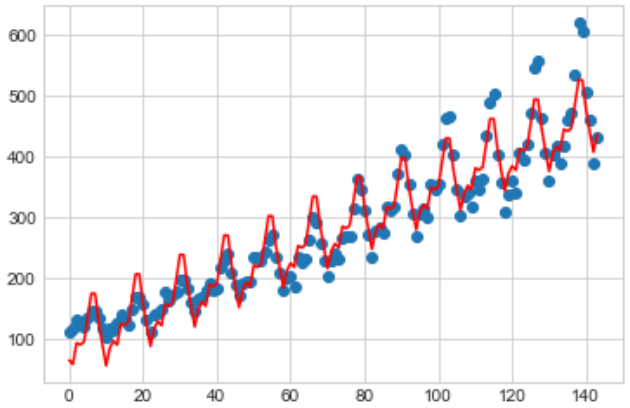
(a) Use the Least Squares method to confirm why the professor used these coefficients. Predict the evaluation the professor would receive from a sophomore applied linear algebra class consisting of 8 students and an average GPA of 3.00. Compute the Pearson correlation coefficient for this regression model and make a judgement whether this predictor is effective or not.

(b) Use Cross Validation to measure the quality of this model. Is it any good? Why or why not? Print your analysis to the screen. Include calculation of prediction error, on the training set and the test set.

(c) Next, employ some feature engineering to improve upon this model. After all, the class ranking is a qualitative variable, not quantitative. It should be hot-button encoded. Are there nonlinear features that could be added for improvement? What else could be improved? Regularization? Experimentation is the key but watch out for overfitting.

3. The link below with take you to a data set listing the number of flights out of an airport by month from 1949 to 1960. This forms an approximately periodic time-series with period $P = 12$. I say that it is approximately periodic because the number of flights from year-to-year is increasing, as illustrated below left. Construct an auto-regressive model to model the number of flights per month at this airport, such as explained in Section 4.3.2. Graph your solution onto the screen. It should look similar to the graph below right. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/airport.py



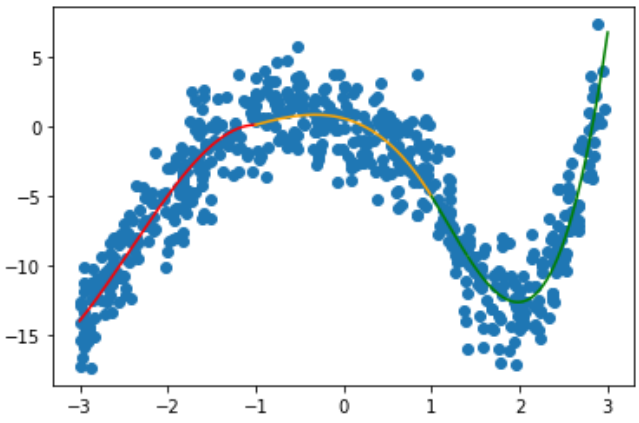


4. The link below will take you to a data set about 569 breast cancer patients from Wisconsin. The data set contains 30 distinct attributes measured from the patient's cancer cells (each measurement will be a positive real number). You are tasked to build a least-squares classifier to determine whether the cancer is `"malignant"` or `"benign"`, such as explained in Section 4.6.2. Use however many of these attributes as you deem fit to build your classifier. The Python code below already has functions for Cross-Validation and the Classifier itself. You need to engineer a model and find its coefficients. Cross-validate to determine how good your model is. Engineer a model with a prediction error of less than 10%. Verify on the two examples include in the Python code. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/cancer.py

5. The link below will take you to a data set about 179 Italian wine samples. The data set contains 13 distinct attributes measured from chemical analysis of the wine samples (each measurement will be a positive real number). You are tasked to build a least-squares multi-class classifier to determine whether the wine is `"class_0"`, `"class_1"`, or `"class_2"`,[ii] such as explained in Section 4.6.3. Use however many of these attributes as you deem fit to build your classifier. The Python code below already has functions for Cross-Validation and the Classifier itself. You need to engineer a model and find its coefficients. Cross-validate to determine how good your model is. Engineer a model with a prediction error of less than 10%. Verify on the two examples include in the Python code. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/finewine.py

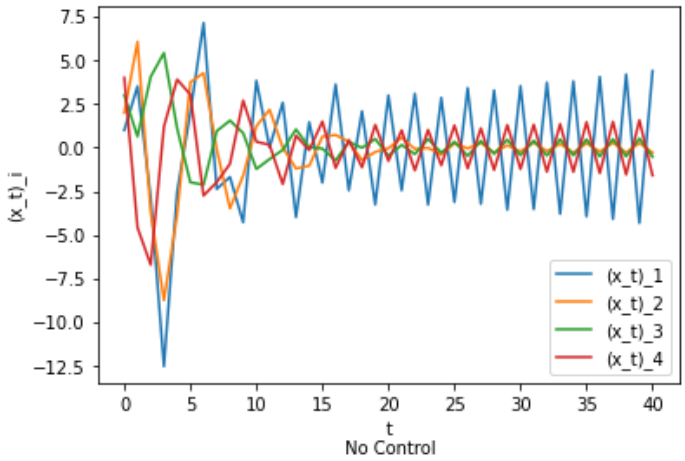
6. In the data set, linked below, you can find a data set of 600 points of the form (x, y). Create a cubic spline with three branches, and hence two switching numbers, namely $x = -1$ and $x = 1$, similar to how it was considered in Example 4.9.3. You will graph the scatterplot and 3-piece cubic spline to the same screen and produce a plot similar to what you see below. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/cubic_spline.py

[ii] Those aren't really fancy names for wines, are they?

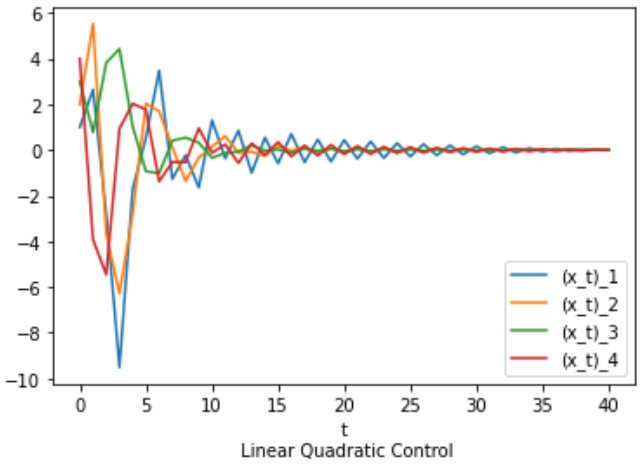


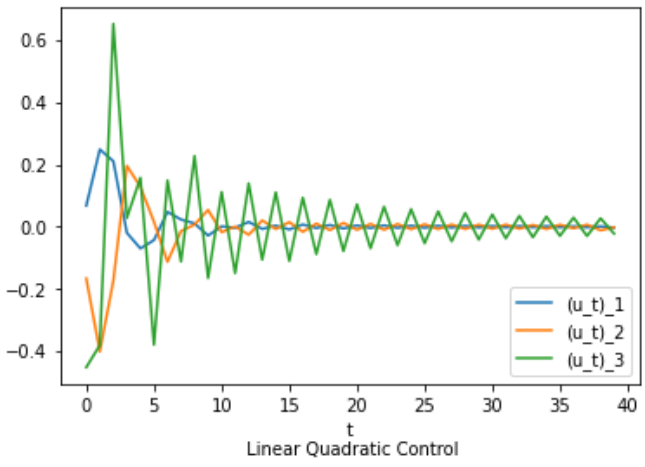
7. Consider the linear dynamical system $\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t + B\boldsymbol{u}_t$, where is the state 4-vector and is the control 3-vector at time t_i, with initial vector $\boldsymbol{x}_0 = (1, 2, 3, 4)$ for $0 \leq t \leq 40$, where the matrices A and B are provided in the link below, such as explained in Section 4.10. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/controlfreak.py

   (a) Simulate the dynamical system with no control (that is, $\boldsymbol{u}_t = \mathbf{0}$ for each t). Graph each coordinate x_i against time t on the same graph, that is, produce the plot illustrated below.

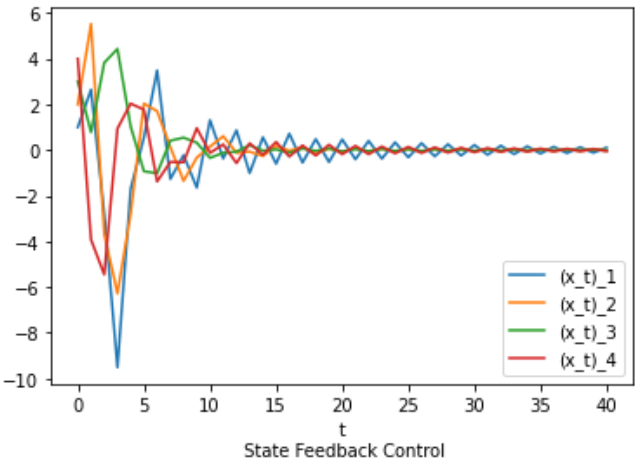


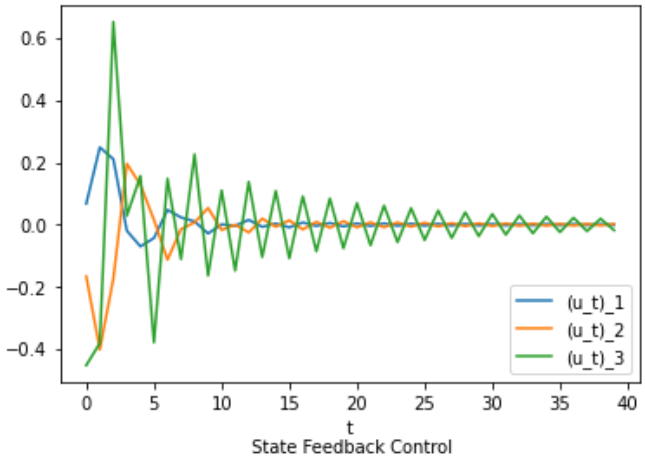
   (b) Next, simulate the dynamical system now using linear quadratic control where $\rho = 100$ and terminal vector $\boldsymbol{x}_{40} = \mathbf{0}$. Graph each state coordinate x_i against time t on the same graph. On a second graph, graph each control coordinate u_i against time t on the same graph. This will look like the following graphs below left and right, respectively.





(c) Finally, simulate the dynamical system now using state feedback where $\rho = 100$ and terminal vector $\boldsymbol{x}_{40}$. You will need to find a 3×4 matrix K such that $\boldsymbol{u}_t = K\boldsymbol{x}_t$. Print this matrix K to screen. Graph each state coordinate x_i against time t on the same graph. On a second graph, graph each control coordinate u_i against time t on the same graph. This will look like the following graphs below left and right, respectively.





Chapter 5

Eigenvalues

The German word *eigen* translates into English as *characteristic*. We will see this German adjective throughout this chapter. As the translated name suggests, the *eigen*values of a matrix are those scalar which characterize the matrix. Together with its *eigen*vectors, we can do exactly that. The *eigen*values and *eigen*vectors characterize their matrices. If we analyze these then we analyze the matrix itself. With the right coordinate system, the *eigen*values show us that the matrix is just a collection of scalar multiplications, and scalar multiplication is so much easier than generic matrix multiplication. These *eigen*values represent the intrinsic properties of a linear transformation or a matrix. They reveal how much a vector is scaled when multiplied by the matrix. *Eigen*vectors are the directions along which the transformation only stretches or compresses, without changing direction. Applications of *eigen*values are everywhere in mathematics and science, including stability analysis, quantum mechanics, and vibration modes.

"Try not to become a man of success, but rather try to become a man of value." – Albert Einstein

5.1 Eigenvalues and Eigenvectors

While a linear transformation can act on all vectors in its domain, not all vectors are created equal here. Certain vectors optimize the image of the transformation. More specificially, the eigenvectors are those vectors in the domain whose norms are changed the most by the transformation, for good or for ill. While other vectors can be distorted and tilted by the transformations, its eigenvectors are those which are moved at all by the function. As such, the full force of the linear transformation is placed into forming a scalar product of the original vector.

5.1.1 Eigenvalues

Example 5.1.1. Let $A = \begin{bmatrix} 3 & -2 \\ 1 & 0 \end{bmatrix}$, $\boldsymbol{u} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, and $\boldsymbol{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$. Then

$$A\boldsymbol{u} = \begin{bmatrix} 3 & -2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -5 \\ -1 \end{bmatrix} \quad \text{and} \quad A\boldsymbol{v} = \begin{bmatrix} 3 & -2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix} = 2\boldsymbol{v}.$$

Notice that while multiplication by A sends $\boldsymbol{u}$ off to who-knows-where, multiplication by A only stretches the vector $\boldsymbol{v}$. This chapter will explore more deeply this type of phenomenon. ∎

Definition 5.1.2. Let A be an $n \times n$ matrix. Then a nonzero vector $\boldsymbol{x}$ is an **eigenvector** of A if there is a scalar λ such that

$$A\boldsymbol{x} = \lambda\boldsymbol{x}.$$

In this case, λ is called an **eigenvalue** of A corresponding to $\boldsymbol{x}$.

It is straight forward to check if a vector is an eigenvector.

Example 5.1.3. Let $A = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix}$, $\boldsymbol{u} = \begin{bmatrix} 6 \\ -5 \end{bmatrix}$, and $\boldsymbol{v} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$. Then

$$A\boldsymbol{u} = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 6 \\ -5 \end{bmatrix} = \begin{bmatrix} -24 \\ 20 \end{bmatrix} = -4 \begin{bmatrix} 6 \\ -5 \end{bmatrix} = -4\boldsymbol{u}.$$

On the other hand,

$$A\boldsymbol{v} = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ -2 \end{bmatrix} = \begin{bmatrix} -9 \\ 11 \end{bmatrix} \neq \lambda \begin{bmatrix} 3 \\ -2 \end{bmatrix},$$

since

$$\begin{cases} -9\lambda = 3 \\ 11\lambda = -2 \end{cases}$$

has no solution. Therefore, $\boldsymbol{u}$ is an eigenvector of A with eigenvalue $\lambda = -4$. The vector $\boldsymbol{v}$ is not an eigenvector of A. ∎

5.1.2 Eigenvectors

It can also be checked whether a scalar is an eigenvalue of a matrix, although this requires row reduction.

Example 5.1.4. Let A be the same matrix as the previous example. Is 7 an eigenvalue of A?

Now, suppose that λ is an eigenvalue of A with eigenvector $\boldsymbol{x}$. Then

$$\begin{aligned} A\boldsymbol{x} &= \lambda\boldsymbol{x} \\ A\boldsymbol{x} - \lambda\boldsymbol{x} &= \mathbf{0} \\ A\boldsymbol{x} - \lambda I\boldsymbol{x} &= \mathbf{0} \\ (A - \lambda I)\boldsymbol{x} &= \mathbf{0} \end{aligned}$$

Therefore, $\lambda = 7$ is an eigenvalue if and only if $A - 7I$ has a nontrivial solution to the homogeneous system $(A - 7I)\boldsymbol{x} = \mathbf{0}$. Now,

$$A - 7I = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix} - \begin{bmatrix} 7 & 0 \\ 0 & 7 \end{bmatrix} = \begin{bmatrix} -6 & 6 \\ 5 & -5 \end{bmatrix}.$$

But,

$$\left[\begin{array}{cc|c} -6 & 6 & 0 \\ 5 & -5 & 0 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & -1 & 0 \\ 5 & -5 & 0 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & -1 & 0 \\ 0 & 0 & 0 \end{array}\right].$$

Therefore, $\boldsymbol{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ is a nontrivial solution to this homogeneous system and hence an eigenvector of A. Note that

$$A\boldsymbol{x} = \begin{bmatrix} 1 & 6 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 7 \end{bmatrix} = 7 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 7\boldsymbol{x}.$$

Therefore, 7 is an eigenvalue of A with eigenvector $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. ∎

Notice in the last example, that $\boldsymbol{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ was an eigenvector for A with eigenvalue 7. Now, this is not the only eigenvector for 7. In fact, $\boldsymbol{y} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$ is also an eigenvector. Note that $A\boldsymbol{y} = \begin{bmatrix} 14 \\ 14 \end{bmatrix} = 7\boldsymbol{y}$. Now, $\boldsymbol{y} = 2\boldsymbol{x}$. In fact, any nonzero multiple of $\boldsymbol{x}$ is an eigenvector of A since

$$A(c\boldsymbol{x}) = c(A\boldsymbol{x}) = c(\lambda x) = \lambda(c\boldsymbol{x}).$$

Furthermore, any nontrivial solution to the homogeneous system $(A - \lambda I)\boldsymbol{x} = \mathbf{0}$ is an eigenvector. But this is the null space of $(A - \lambda I)$, a subspace of $\mathbb{R}^n$. This leads to the next definition.

Definition 5.1.5. Let A be an $n \times n$ matrix with eigenvalue λ. Then the null space of $(A - \lambda I)$ is called the **eigenspace** of A corresponding to λ, denoted $\text{Eigen}(A, \lambda) = \text{Nul}(A - \lambda I_n)$. The dimension of the eigenspace is called the **geometric multiplicity** of the eigenvalue λ and corresponds to the nullity of the matrix $A - \lambda I$.

Since eigenspaces are null spaces of a matrix, they are always subspaces of $\mathbb{R}^n$.

Example 5.1.6. Let $A = \begin{bmatrix} 4 & -1 & 6 \\ 2 & 1 & 6 \\ 2 & -1 & 8 \end{bmatrix}$. An eigenvalue of A is $\lambda = 2$. Find a basis for the eigenspace Eigen$(A, 2)$.

We form

$$A - 2I = \begin{bmatrix} 4 & -1 & 6 \\ 2 & 1 & 6 \\ 2 & -1 & 8 \end{bmatrix} - \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 2 & -1 & 6 \\ 2 & -1 & 6 \\ 2 & -1 & 6 \end{bmatrix}.$$

Thus,

$$\begin{bmatrix} 2 & -1 & 6 \\ 2 & -1 & 6 \\ 2 & -1 & 6 \end{bmatrix} \sim \begin{bmatrix} 2 & -1 & 6 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

This implies that the null space[i] of $A - 2I$ are the solutions to the equation

$$2x_1 - x_2 + 6x_3 = 0.$$

In other words,

$$\text{Eigen}(A, 2) = \text{Nul}(A - 2I) = \text{Span}\left\{ \begin{bmatrix} 1/2 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -3 \\ 0 \\ 1 \end{bmatrix} \right\} = \text{Span}\left\{ \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} -3 \\ 0 \\ 1 \end{bmatrix} \right\}.$$

In fact, the eigenspace is 2-dimensional with basis $\left\{ \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} -3 \\ 0 \\ 1 \end{bmatrix} \right\}$. ■

5.1.3 Eigenvalues of Triangular Matrices

Theorem 5.1.7. *The eigenvalues of a triangular matrix are the entries on its main diagonal.*

Example 5.1.8. Let $A = \begin{bmatrix} 3 & 6 & -8 \\ 0 & 0 & 6 \\ 0 & 0 & 2 \end{bmatrix}$ and $B = \begin{bmatrix} 4 & 0 & 0 \\ -2 & 1 & 0 \\ 5 & 3 & 4 \end{bmatrix}$. Since both matrices are triangular, the eigenvalues of A are $\lambda = 3, 0, 2$ and the eigenvalues of B are $\lambda = 4, 1$. ■

Notice from the previous example that 4 appeared twice along the diagonal of B. This is a repeated eigenvalue of multiplicity two. This idea of multiplicity will also return later in this chapter.

[i] Please review Section 2.8.1 for further study of finding a basis for the null space of a matrix, if necessary. This is a calculation we use heavily in this chapter to find eigenvectors of a matrix.

5.1.4 The Characteristic Polynomial

This is an optional subsection, as no other materials from this subsection will be used in the remainder of this chapter. Our reason to include it is to give some idea behind the blackbox of computing eigenvalues of an $n \times n$ matrix A. In a traditional linear algebra textbook, the notion of the determinant of a square matrix A is introduced (we have only alluded to it in this text) and this provides an effective tool for finding eigenvalues when n is small, much more effective than the strategy described below. But for large n, both of these methods, finding eigenvalues via determinants or via row reductions as illustrated below, prove to be very ineffective. To discuss algorithms for finding, or approximating, eigenvalues for large matrices requires computational linear algebra which goes beyond the scope of this textbook.

As observed previously, for an $n \times n$ matrix A, a scalar λ is an eigenvalue if and only if $(A - \lambda I)\boldsymbol{x} = \mathbf{0}$ has a nontrivial solution if and only if $A - \lambda I$ is singular. As we are considering a square matrix, it is singular if and only if every echelon form of the matrix has a row of zeros. This is a strategy we can pursue.

Example 5.1.9. Let $A = \begin{bmatrix} -6 & -2 & -2 \\ 17 & 6 & 5 \\ 7 & 2 & 3 \end{bmatrix}$. Then $\lambda I_3 - A = \begin{bmatrix} \lambda+6 & 2 & 2 \\ -17 & \lambda-6 & -5 \\ -7 & -2 & \lambda-3 \end{bmatrix}$. Then

$$\lambda I_3 - A \sim \begin{bmatrix} \lambda+6 & 2 & 2 \\ -17 & \lambda-6 & -5 \\ -7 & -2 & \lambda-3 \end{bmatrix} \sim \begin{bmatrix} -7 & -2 & \lambda-3 \\ -17 & \lambda-6 & -5 \\ \lambda+6 & 2 & 2 \end{bmatrix}$$

$$\sim \begin{bmatrix} 1 & \frac{2}{7} & \frac{2-\lambda}{7} \\ -17 & \lambda-6 & -5 \\ \lambda+6 & 2 & 2 \end{bmatrix} \sim \begin{bmatrix} 1 & \frac{2}{7} & \frac{2-\lambda}{7} \\ 0 & \lambda-\frac{8}{7} & -\frac{17\lambda+1}{7} \\ 0 & -\frac{2}{7}(\lambda-1) & \frac{1}{7}(\lambda-1)(\lambda+4) \end{bmatrix}$$

Note that at this moment, if $\lambda = 1$, then $-\frac{2}{7}(\lambda-1) = 0$ and $\frac{1}{7}(\lambda-1)(\lambda+4) = 0$. This is a row of zeros. Hence, $\lambda = 1$ is an eigenvalue. If $\lambda \neq 1$, then we may divide by $-\frac{2}{7}(\lambda-1)$ and we proceed in row reduction.

$$\lambda I_3 - A \sim \begin{bmatrix} 1 & \frac{2}{7} & \frac{2-\lambda}{7} \\ 0 & \lambda-\frac{8}{7} & -\frac{17\lambda+1}{7} \\ 0 & 1 & -\frac{1}{2}(\lambda+4) \end{bmatrix} \sim \begin{bmatrix} 1 & \frac{2}{7} & \frac{2-\lambda}{7} \\ 0 & 1 & -\frac{1}{2}(\lambda+4) \\ 0 & \lambda-\frac{8}{7} & -\frac{17\lambda+1}{7} \end{bmatrix} \sim \begin{bmatrix} 1 & \frac{2}{7} & \frac{2-\lambda}{7} \\ 0 & 1 & -\frac{1}{2}(\lambda+4) \\ 0 & 0 & \frac{1}{2}(\lambda^2-2\lambda) \end{bmatrix}$$

Thus, if $\frac{1}{2}(\lambda^2 - 2\lambda) = 0$, then the third row is again a row of zeros. Solving, we have $\lambda^2 - 2\lambda = 0$, that is, $\lambda(\lambda-2) = 0$. Hence, $\lambda = 0, 2$. Combining with what we saw previously, the eigenvalues of this matrix are $\lambda = 0, 1, 2$. ∎

Combining the two polynomials we saw in the previous example we have $\lambda(\lambda-1)(\lambda-2) = \lambda^3 - 3\lambda^2 + 2\lambda$. Note that the roots of this polynomial is exactly the eigenvalues of A. Mimicking the technique of the previous example will always produce a polynomial such as this whose roots are the eigenvalues of the matrix. This polynomial is called the **characterisitic polynomial** of the matrix. The number of times a root is repeated in the characteristic polynomial is called the **algebraic multiplicity** of that eigenvalue. Because of the shear complexity of finding the characteristic polynomial for even such a small example, in practice an numerical approximation of eigenvalues is satisificatory as they can be found much easier.

Python Programming

The function `np.eigvals(A)` computes the eigenvalues of A.

Example 5.1.10. The code can be used to find the eigenvalues of $A = \begin{bmatrix} -2 & 4 & -2 & 1 \\ -15 & 29 & -13 & 23 \\ -30 & 44 & -19 & 34 \\ 0 & -4 & 2 & -3 \end{bmatrix}$:

```
import numpy as np

A = np.array([[-2,4,-2,1],  # Define the matrix A
              [-15,29,-13,23],
              [-30,44,-19,34],
              [0,-4,2,-3]])
eigen = np.linalg.eigvals(A) # Return the eigenvalues, each repeated according to its
    multiplicity.
print(eigen)
```

namely: `[-2.  3.  3.  1.]` ■

Exercises

(Go to Solutions)

For Exercises 1-3, determine if the vector is an eigenvector for $A = \begin{bmatrix} 2 & 6 \\ 3 & -1 \end{bmatrix}$. If so, what is the eigenvalue?

1. $\begin{bmatrix} 6 \\ 3 \end{bmatrix}$ 2. $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ 3. $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$

For Exercises 4-6, determine if the vector is an eigenvector for $A = \begin{bmatrix} 1 & 2 \\ 4 & -1 \end{bmatrix}$. If so, what is the eigenvalue?

4. $\begin{bmatrix} -1 \\ 3 \end{bmatrix}$ 5. $\begin{bmatrix} 4 \\ 7 \end{bmatrix}$ 6. $\begin{bmatrix} 2 \\ 2 \end{bmatrix}$

For Exercises 7-9, determine if the vector is an eigenvector for $A = \begin{bmatrix} -6 & -21 & -16 \\ 6 & 17 & 12 \\ -5 & -12 & -8 \end{bmatrix}$. If so, what is the eigenvalue?

♠ 7. $\begin{bmatrix} -2 \\ 0 \\ 1 \end{bmatrix}$ ♠ 8. $\begin{bmatrix} 2 \\ -1 \\ -1 \end{bmatrix}$ ♠ 9. $\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$

For Exercises 10-12, determine if the vector is an eigenvector for $A = \begin{bmatrix} 2 & -5 & -2 \\ 2 & -7 & -3 \\ -4 & 14 & 6 \end{bmatrix}$. If so, what is the eigenvalue?

♠ 10. $\begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix}$ ♠ 11. $\begin{bmatrix} -1 \\ -2 \\ 4 \end{bmatrix}$ ♠ 12. $\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$

For Exercises 13-17, find an eigenvector for the matrix A and eigenvalue λ. Answers may vary.

13. $\begin{bmatrix} 3 & 6 \\ 0 & 1 \end{bmatrix}$, $\lambda = 3$

♠ 14. $\begin{bmatrix} 3 & 1 \\ 0 & -2 \end{bmatrix}$, $\lambda = 3$

♠ 15. $\begin{bmatrix} 1 & 3 \\ -1 & 5 \end{bmatrix}$, $\lambda = 2$

♠ 16. $\begin{bmatrix} -2 & -1 \\ 1 & -4 \end{bmatrix}$, $\lambda = -3$

17. $\begin{bmatrix} 3 & 5 & 7 \\ 1 & 5 & 9 \\ 4 & 5 & 6 \end{bmatrix}$, $\lambda = 0$

For Exercises 18-20, find a basis for the eigenspace for the matrix and each eigenvalue λ listed. Answers may vary. Also, determine the geometric multiplicities of each listed eigenvalue.

18. $\begin{bmatrix} 0 & 2 \\ 2 & 3 \end{bmatrix}$, $\lambda = -1, 4$

♠ 19. $\begin{bmatrix} -7 & 2 & 32 \\ -8 & 1 & 0 \\ -2 & 1 & 13 \end{bmatrix}$, $\lambda = 5, -3$

♠ 20. $\begin{bmatrix} -8 & -5 & 5 \\ 20 & 12 & -10 \\ 10 & 5 & -3 \end{bmatrix}$, $\lambda = -3, 2$

21. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/eigen.ipynb

"The line of life is a ragged diagonal between duty and desire." – William R. Alger

5.2 Diagonalization

Similar matrices always result in a matrix factorization. When a matrix A is similar to a diagonal matrix D, we obtain a very important factorization called its diagonalization. This factorization is obtained by analyzing the eigenvalues and eigenvectors of the matrix A. If used correctly, it simplifies matrix exponentiation, powers, and solving linear systems.

5.2.1 Similar Matrices

Definition 5.2.1. Let A and B be $n \times n$ matrices. We say that two matrices are **similar** if there exists a nonsingular $n \times n$ matrix P such that $PAP^{-1} = B$.

Example 5.2.2. The matrices $A = \begin{bmatrix} 1 & 0 & -7 \\ 5 & 1 & 2 \\ -4 & 2 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 15 & -18 & -2 \\ 17 & -17 & -4 \\ 7 & -22 & 4 \end{bmatrix}$ are similar since $P = \begin{bmatrix} 1 & -2 & -1 \\ 1 & -1 & 0 \\ 1 & -4 & -2 \end{bmatrix}$ is nonsingular with $P^{-1} = \begin{bmatrix} 2 & 0 & -1 \\ 2 & -1 & -1 \\ -3 & 2 & 1 \end{bmatrix}$ and

$$PAP^{-1} = \begin{bmatrix} 1 & -2 & -1 \\ 1 & -1 & 0 \\ 1 & -4 & -2 \end{bmatrix} \begin{bmatrix} 1 & 0 & -7 \\ 5 & 1 & 2 \\ -4 & 2 & 0 \end{bmatrix} \begin{bmatrix} 2 & 0 & -1 \\ 2 & -1 & -1 \\ -3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -2 & -1 \\ 1 & -1 & 0 \\ 1 & -4 & -2 \end{bmatrix} \begin{bmatrix} 23 & -14 & -8 \\ 6 & 3 & -4 \\ -4 & -2 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 15 & -18 & -2 \\ 17 & -17 & -4 \\ 7 & -22 & 4 \end{bmatrix} = B. \qquad \blacksquare$$

5.2.2 Diagonalizable Matrices

Definition 5.2.3. Let A be an $n \times n$ matrix. We say that A is **diagonalizable** if A is similar to a diagonal matrix D, that is, $A = PDP^{-1}$ for some invertible matrix P.

Example 5.2.4. Let $A = \begin{bmatrix} 7 & 2 \\ -4 & 1 \end{bmatrix}$. Then A is a diagonalizable matrix. Let $P = \begin{bmatrix} 1 & 1 \\ -1 & -2 \end{bmatrix}$ and $D = \begin{bmatrix} 5 & 0 \\ 0 & 3 \end{bmatrix}$. Then $P^{-1} = \begin{bmatrix} 2 & 1 \\ -1 & -1 \end{bmatrix}$ and

$$A = PDP^{-1} = \begin{bmatrix} 1 & 1 \\ -1 & -2 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ -1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -1 & -2 \end{bmatrix} \begin{bmatrix} 10 & 5 \\ -3 & -3 \end{bmatrix} = \begin{bmatrix} 7 & 2 \\ -4 & 1 \end{bmatrix}.$$

Next, observe that

$$A^2 = (PDP^{-1})(PDP^{-1}) = PD(P^{-1}P)DP^{-1}$$
$$= PD^2P^{-1} = \begin{bmatrix} 1 & 1 \\ -1 & -2 \end{bmatrix}\begin{bmatrix} 5^2 & 0 \\ 0 & 3^2 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ -1 & -1 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 1 \\ -1 & -2 \end{bmatrix}\begin{bmatrix} 2\cdot 5^2 & 5^2 \\ -3^2 & -3^2 \end{bmatrix} = \begin{bmatrix} 2\cdot 5^2 - 3^2 & 5^2 - 3^2 \\ -2\cdot 5^2 + 2\cdot 3^2 & -5^2 + 2\cdot 3^2 \end{bmatrix}$$
$$= \begin{bmatrix} 41 & 16 \\ -32 & -7 \end{bmatrix} \neq \begin{bmatrix} 1^2 & 1^2 \\ (-1)^2 & (-2)^2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 4 \end{bmatrix}.$$

This might seem, at first, overkill for squaring a 2×2 matrix, but this pattern continues for power. In fact, it follows by induction that

$$A^k = PD^kP^{-1} = \begin{bmatrix} 1 & 1 \\ -1 & -2 \end{bmatrix}\begin{bmatrix} 5^k & 0 \\ 0 & 3^k \end{bmatrix}\begin{bmatrix} 2 & 1 \\ -1 & -1 \end{bmatrix} = \begin{bmatrix} 2\cdot 5^k - 3^k & 5^k - 3^k \\ -2\cdot 5^k + 2\cdot 3^k & -5^k + 2\cdot 3^k \end{bmatrix}.$$

Among other reasons, diagonalization provides an effective method to compute powers of matrices. ∎

From the previous example, note that

$$A\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 7 & 2 \\ -4 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 5 \\ -5 \end{bmatrix} = 5\begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

and

$$A\begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 7 & 2 \\ -4 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 3 \\ -6 \end{bmatrix} = 3\begin{bmatrix} 1 \\ -2 \end{bmatrix}.$$

Thus, the column vectors of P are eigenvectors of A! Furthermore, the diagonal entries of D are the eigenvalues of A!

5.2.3 Eigenbases

Theorem 5.2.5. *If $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_r$ are eigenvectors of an $n \times n$ matrix A which correspond to distinct eigenvalues $\lambda_1, \ldots, \lambda_r$, respectively, then $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_r\}$ is linearly independent.*

Theorem 5.2.6 (The Diagonalization Theorem)**.** *An $n \times n$ matrix A is diagonalizable if and only if A has n linearly independent eigenvectors. In this case, there is a basis of $\mathbb{R}^n$ consisting of eigenvectors of A, called an **eigenvector basis** (or **eigenbasis**). If $A = PDP^{-1}$, then the diagonal entries of D are the eigenvalues of A with multiplicity, the columns of P are eigenvectors, and the eigenvectors in P correspond to the eigenvalues in the same column in D .*

Theorem 5.2.7. *An $n \times n$ matrix with n distinct eigenvalues is diagonalizable.*

Example 5.2.8. If possible, diagonalize the matrix $A = \begin{bmatrix} 1 & 3 & 3 \\ -3 & -5 & -3 \\ 3 & 3 & 1 \end{bmatrix}$.

A quick calculation in Python:

```
import numpy as np

print(np.linalg.eigvals(
    np.array([[1,3,3],
              [-3,-5,-3],
              [3,3,1]])))
```

gives the eigenvalues as `[ 1.  -2.  -2.]`.

Next, we compute the eigenvectors of A. We begin with $\lambda = 1$.

$$A - I = \begin{bmatrix} 0 & 3 & 3 \\ -3 & -6 & -3 \\ 3 & 3 & 0 \end{bmatrix} \sim \begin{bmatrix} 0 & 3 & 3 \\ -3 & -6 & -3 \\ 0 & 0 & 0 \end{bmatrix} \sim \begin{bmatrix} 0 & 1 & 1 \\ 1 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

Therefore, $\text{Eigen}(A, 1) = \text{Nul}(A - I) = \text{Span}\left\{\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}\right\}$. Next, we use $\lambda = -2$.

$$A + 2I = \begin{bmatrix} 3 & 3 & 3 \\ -3 & -3 & -3 \\ 3 & 3 & 3 \end{bmatrix} \sim \begin{bmatrix} 3 & 3 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Therefore, $\text{Eigen}(A, -2) = \text{Nul}(A + 2I) = \text{Span}\left\{\begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}\right\}$. Since A has a basis of eigenvectors, A is diagonalizable. Let $P = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$.

To finish, we need to compute P^{-1}.

$$\left[\begin{array}{ccc|ccc} 1 & -1 & -1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|ccc} 1 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & 1 & 2 & -1 & 0 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|ccc} 1 & -1 & -1 & 1 & 0 & 0 \\ 0 & 1 & 2 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 & 1 & 0 \end{array}\right]$$

$$\sim \left[\begin{array}{ccc|ccc} 1 & -1 & -1 & 1 & 0 & 0 \\ 0 & 1 & 2 & -1 & 0 & 1 \\ 0 & 0 & 1 & -1 & -1 & 0 \end{array}\right] \sim \left[\begin{array}{ccc|ccc} 1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 & 2 & 1 \\ 0 & 0 & 1 & -1 & -1 & 0 \end{array}\right]$$

$$\sim \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 2 & 1 \\ 0 & 0 & 1 & -1 & -1 & 0 \end{array}\right].$$

Therefore, $P^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ -1 & -1 & 0 \end{bmatrix}$ and

$$A = PDP^{-1} = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ -1 & -1 & 0 \end{bmatrix}. \qquad \blacksquare$$

Note that a matrix is diagonalizable if and only if each geometric multiplicity is equal to its algebraic multiplicity.

5.2.4 Stable Solutions

As we have studied linear dynamical systems previously, given by the dynamic equation $\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t$ with transition matrix A. What do the eigenvalues of A illustrate about this linear dynamical system? Imagine $\boldsymbol{v}$ is an eigenvalue of the transition matrix A with eigenvalue λ, that is, $A\boldsymbol{v} = \lambda\boldsymbol{v}$. If ever a state vector $\boldsymbol{x}_t$ were equal to $\boldsymbol{v}$, then $\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t = A\boldsymbol{v} = \lambda\boldsymbol{v} = \lambda\boldsymbol{x}_t$, that is, the next state vector is just a multiple of the previous. Similarly after T states, $\boldsymbol{x}_{t+T} = \lambda^T\boldsymbol{x}_t$. The future state of an eigenvector is within the same linear span, the multiple being some power of λ. As $T \to \infty$, we see three possibilities. First, if $|\lambda| > 1$, then $\lim_{T\to\infty} |\lambda| = \infty$, that is, the norm of the state vector $\|\boldsymbol{x}_{t+T}\|$ explodes. Second, if $|\lambda| < 1$, then $\lim_{T\to\infty} |\lambda| = 0$, that is, the norm of the state vector $\|\boldsymbol{x}_{t+T}\|$ collapses. Third, if $|\lambda| = 1$, then $\lim_{T\to\infty} |\lambda| = 1$, that is, the norm of the state vector $\|\boldsymbol{x}_{t+T}\|$ stabilizes. An eigenvector $\boldsymbol{v}$ with eigenvalue $\lambda = 1$ is called a **stable solution** or **steady state** to the linear dynamical system. The eigenvectors of A represent the optimal directions of growth, collapse, or stability. No state vector explodes as quickly as an eigenvector with large λ; likewise, no vector stabilizes as well as an eigenvector with $\lambda = 1$. Generally speaking, state vectors which are not eigenvectors will converge toward "nearby" eigenvectors, often the eigenvectors with largest $|\lambda|$.

Example 5.2.9. With the litany of streaming services and new content being regularly released, these companies are constantly competing for customers. Consider to popular streaming services, PyFlix and LinearMax. Every month, PyFlix retains only 70% of its subscribers and 30% drop their PyFlix subscription and sign up for LinearMax (PyFlix's chief competitor in the math-related K-drama genre). Similarly, each month LinearMax retains 80% of its subscribers, with 20% migrating to PyFlix. If we consider a dynamical system $\boldsymbol{x}_t = \begin{bmatrix} pf_t \\ lm_t \end{bmatrix}$, where pf_t and lm_t are the numbers of subscribers to PyFlix and LinearMax on month t (measured in thousands of subscribers), and ignore any new subscribers to these services, then

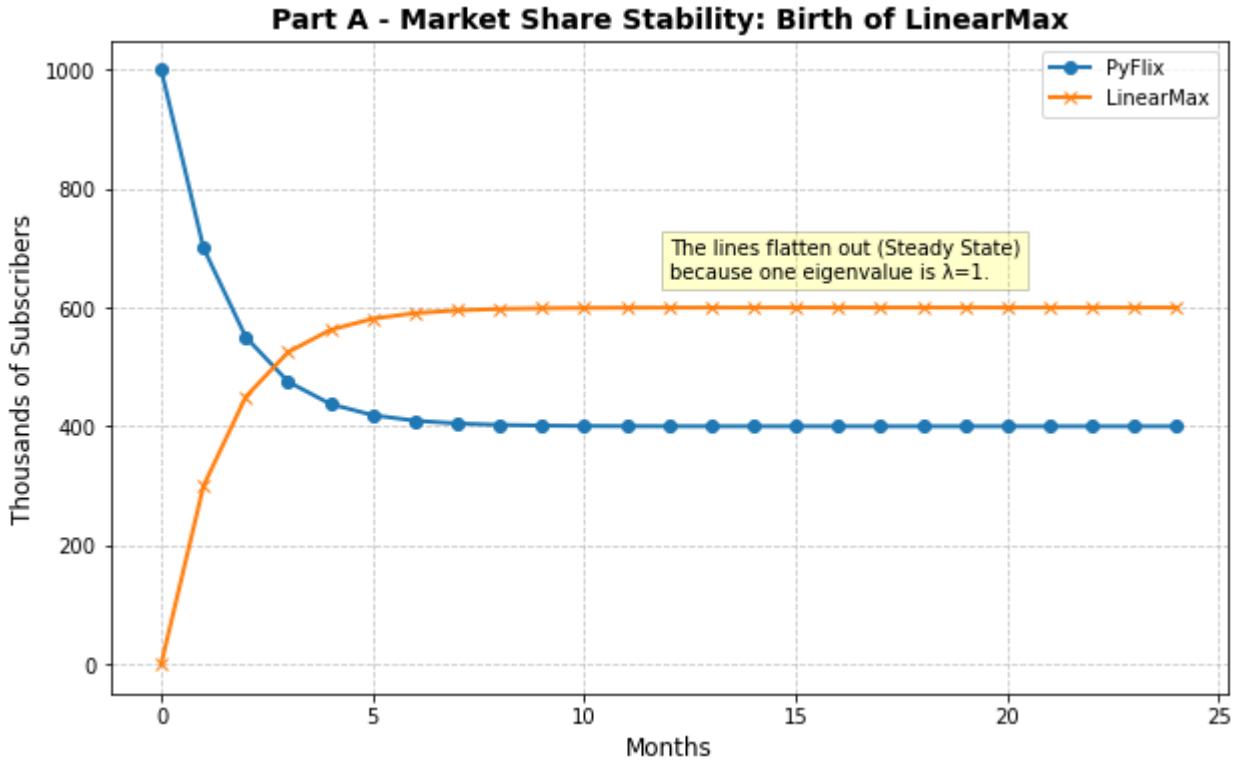
$$A = \begin{bmatrix} 0.70 & 0.20 \\ 0.30 & 0.80 \end{bmatrix}$$

is the transition matrix for this Markov chain.

Suppose that when LinearMax first opened, PyFlix had 1 million subscribers and these trends existed. The graph below demonstrates how the two companies grow over the next 2 years. We see that the number of subscribers for each company converges very quickly toward their steady states, approximately 400,000 for PyFlix and 600,000 for LinearMax. Note that the eigenvalues for A are $\lambda = 1, \frac{1}{2}$ and

$$\begin{bmatrix} 400 \\ 600 \end{bmatrix} \in \text{Eigen}(A, 1) = \text{Span}\left\{ \begin{bmatrix} 2 \\ 3 \end{bmatrix} \right\}.$$

Hence, $\lim_{t\to\infty} \boldsymbol{x}_t = \begin{bmatrix} 400 \\ 600 \end{bmatrix}$, the stable solution to $\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t$ with the same sum as $\boldsymbol{x}_0 = \begin{bmatrix} 1000 \\ 0 \end{bmatrix}$.



The following code simulates this dynamical system:

```
import numpy as np
import matplotlib.pyplot as plt

# ========================================
# PART A: Birth of LinearMax
# ========================================
# The Transition Matrix A
# Column 0: Transitions FROM PyFlix
# Column 1: Transitions FROM LinearMax
A = np.array([
    [0.70, 0.20],  # TO PyFlix
    [0.30, 0.80]   # TO LinearMax
    ])

# Initial State vector x_0 (Start of first year)
# 1000K subscribers for PyFlix, 0 for LinearMax
x = np.array([1000, 0])

# Calculate eigenvalues to predict behavior before running simulation
print('Birth of LinearMax - Eigenvalues of Transition Matrix:', np.linalg.eigvals(A), '\n',
    '-' * 30)

# Run simulation
T = 24
pf = [x[0]]
lm = [x[1]]

for t in range(T):
    x = A @ x
    pf.append(x[0])
    lm.append(x[1])

# Plot 1: Stabilization (Convergence to Steady State)
```

```
plt.figure(figsize=(10, 6))
plt.plot(range(T + 1), pf, label='PyFlix', marker='o', linewidth=2, color='tab:blue')
plt.plot(range(T + 1), lm, label='LinearMax', marker='x', linewidth=2, color='tab:orange')

plt.title('Part A - Market Share Stability: Birth of LinearMax', fontsize=14,fontweight='
    bold')
plt.xlabel('Months', fontsize=12)
plt.ylabel('Thousands of Subscribers', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.text(T/2, 650,
         'The lines flatten out (Steady State)\nbecause one eigenvalue is ' + r'$\lambda=1.$
    ',
         bbox=dict(facecolor='yellow', alpha=0.2))
plt.show()
```

Why does $\boldsymbol{x}_t$ converge toward the stable solution? The second eigenspace of A is given as $\text{Eigen}(A, 0.5) = \text{Span}\left\{\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right\}$. As it is not feasible for a state vector to have negative entries, no state vector $\boldsymbol{x}_t$ could ever be "near" this eigenspace. If $\boldsymbol{x}_0$ is a generic initial state vector, since $\left\{\begin{bmatrix} 2 \\ 3 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix}\right\}$ forms an eigenbasis for $\mathbb{R}^2$ associated to A, there exists coordinates $r, s \in \mathbb{R}$ such that

$$\boldsymbol{x}_0 = r\begin{bmatrix} 2 \\ 3 \end{bmatrix} + s\begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

. Then

$$\boldsymbol{x}_1 = A\boldsymbol{x}_0 = A\left(r\begin{bmatrix} 2 \\ 3 \end{bmatrix} + s\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right) = r\left(A\begin{bmatrix} 2 \\ 3 \end{bmatrix}\right) + s\left(A\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right) = r\begin{bmatrix} 2 \\ 3 \end{bmatrix} + s(0.5)\begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Repeating this process t times, we see that

$$\boldsymbol{x}_t = r\begin{bmatrix} 2 \\ 3 \end{bmatrix} + s(0.5)^t\begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

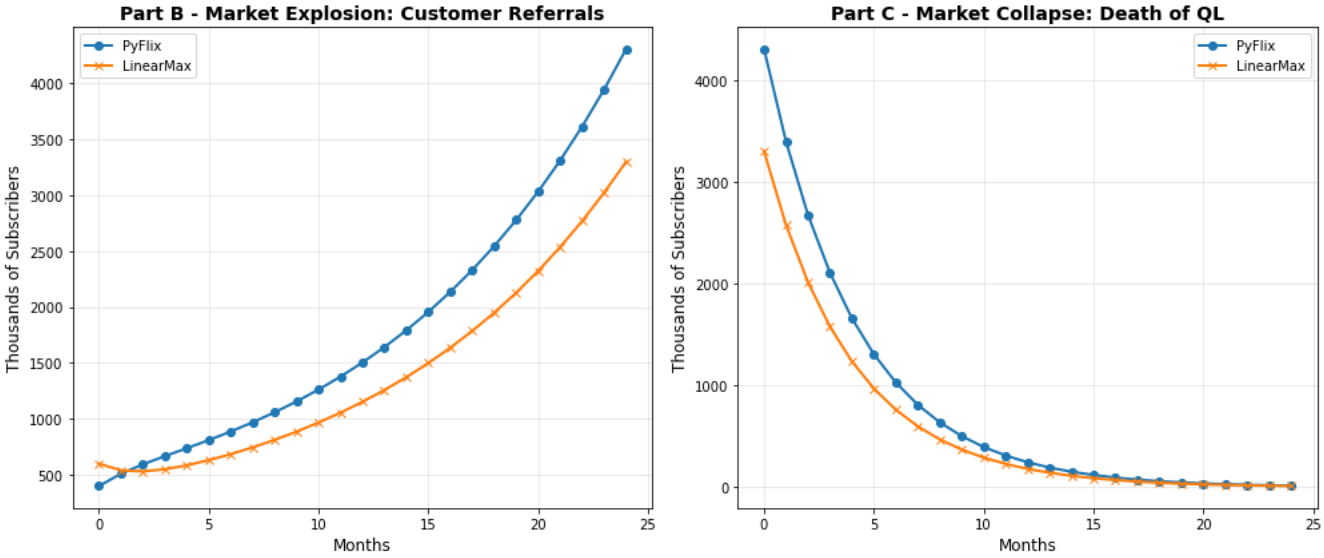
Since $\lim_{t\to\infty}(0.5)^t = 0$, we have $\lim_{t\to\infty}\boldsymbol{x}_t = r\begin{bmatrix} 2 \\ 3 \end{bmatrix}$, that is, every initial vector will cause the dynamical system to converge toward the stable solution.

Continuing with the above competition, suppose now that PyFlix offers a generous customer referral program, which causes 20% of their current subscribers to add a new customer not currently using either streaming service. Also, controversy has hit LinearMax since releasing a documentary advocating that the high school math sequence Secondary Math I, Secondary Math II, and Secondary Math III is more beneficial for high school students than the traditional Algebra I, Geometry, Algebra II sequence. This causes LinearMax to only retain 70% of its subscribers, and while 25% of LinearMax's subscribers move over to PyFlix, 5% do not subscribe with LinearMax nor PyFlix. Therefore, the transition matrix A is modified to

$$A = \begin{bmatrix} 0.90 & 0.25 \\ 0.30 & 0.70 \end{bmatrix}.$$

If we simulate the next 24 months with this new transition matrix, continuing where the last 2 years left off, we see the trend illustrated to the below left. This trend is an explosion of customers for both services.

While LinearMax is losing more subscribers, because PyFlix is adding several customers, many of these PyFlix customers later decide to move over to LinearMax. Overall, both companies explode in the market. Note that the eigenvalues of this new transition matrix are $\lambda = 1.09154759, 0.50845241$.



The following code simulates this dynamical system:

```
# ========================================
# PART B: Customer Referrals
# ========================================
# The Transition Matrix A_ref
# Column 0: Transitions FROM PyFlix
# Column 1: Transitions FROM LinearMax
A_ref = np.array([
    [0.9, 0.25],  # TO PyFlix
    [0.30, 0.70]   # TO LinearMax
    ])

# Calculate eigenvalues to predict behavior before running simulation
print('Customer Referrals - Eigenvalues of Transition Matrix:', np.linalg.eigvals(A_ref), '\
    n', '-' * 30)

# Run simulation
# Initial State vector x_25 (start of 3rd year)
# approximately 400K subscribers for PyFlix, 600K for LinearMax
T_ref = 24
pf_ref = [x[0]]
lm_ref = [x[1]]

for t in range(T_ref):
    x = A_ref @ x
    pf_ref.append(x[0])
    lm_ref.append(x[1])
# Run simulation
# Initial State vector x_25 (start of 3rd year)
# approximately 400K subscribers for PyFlix, 600K for LinearMax
T_ql = 24
pf_ql = [x[0]]
lm_ql = [x[1]]

for t in range(T_ql):
    x = A_ql @ x
    pf_ql.append(x[0])
    lm_ql.append(x[1])

# --- Visualize Parts B and C ---
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

```

```
# Plot 2: Growth (Divergence)
ax1.plot(range(T_ref + 1), pf_ref, label='PyFlix', marker='o', linewidth=2, color='tab:blue')
ax1.plot(range(T_ref + 1), lm_ref, label='LinearMax', marker='x', linewidth=2, color='tab:orange')
ax1.set_title('Part B - Market Explosion: Customer Referrals', fontsize=14, fontweight='bold')
ax1.set_xlabel('Months', fontsize=12)
ax1.set_ylabel('Thousands of Subscribers', fontsize=12)
ax1.grid(True, alpha=0.3)
ax1.legend()
```

Controversy continues, but now the government changes education policies that high school graduates now only need one year of math, instead of three. As such, many subscribers drop their service with PyFlex and LinearMax. This new trend is given by a new transition matrix

$$A = \begin{bmatrix} 0.75 & 0.05 \\ 0.10 & 0.65 \end{bmatrix}.$$

If we simulate the next 24 months with this new transition matrix, continuing where the last 2 years left off, we see the trend illustrated to the above right. This trend is a collapse of customers for both services. Note that the eigenvalues of this new transition matrix are $\lambda = 0.78660254, 0.61339746$. The following code simulates this dynamical system:

```
# ========================================
# PART C: Death of Quantitative Literacy
# ========================================
# The Transition Matrix A_ql
# Column 0: Transitions FROM PyFlix
# Column 1: Transitions FROM LinearMax
A_ql = np.array([
    [0.75, 0.05],  # TO PyFlix
    [0.10, 0.65]   # TO LinearMax
    ])

# Calculate eigenvalues to predict behavior before running simulation
print('Death of QL - Eigenvalues of Transition Matrix:', np.linalg.eigvals(A_ql), '\n', '-' * 30)

# Run simulation
# Initial State vector x_25 (start of 3rd year)
# approximately 400K subscribers for PyFlix, 600K for LinearMax
T_ql = 24
pf_ql = [x[0]]
lm_ql = [x[1]]

for t in range(T_ql):
    x = A_ql @ x
    pf_ql.append(x[0])
    lm_ql.append(x[1])

# --- Visualize Parts B and C ---
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Plot 2: Growth (Divergence)
ax1.plot(range(T_ref + 1), pf_ref, label='PyFlix', marker='o', linewidth=2, color='tab:blue')
ax1.plot(range(T_ref + 1), lm_ref, label='LinearMax', marker='x', linewidth=2, color='tab:orange')
ax1.set_title('Part B - Market Explosion: Customer Referrals', fontsize=14, fontweight='bold')
ax1.set_xlabel('Months', fontsize=12)
ax1.set_ylabel('Thousands of Subscribers', fontsize=12)
ax1.grid(True, alpha=0.3)
ax1.legend()

# Plot 3: Decay (Convergence to Zero)
ax2.plot(range(T_ql + 1), pf_ql, label='PyFlix', marker='o', linewidth=2, color='tab:blue')
```

```
ax2.plot(range(T_ql + 1), lm_ql, label='LinearMax', marker='x', linewidth=2, color='tab:
    orange')
ax2.set_title('Part C - Market Collapse: Death of QL', fontsize=14, fontweight='bold')
ax2.set_xlabel('Months', fontsize=12)
ax2.set_ylabel('Thousands of Subscribers', fontsize=12)
ax2.grid(True, alpha=0.3)
ax2.legend()

plt.tight_layout()
plt.show()
# Plot 3: Decay (Convergence to Zero)
ax2.plot(range(T_ql + 1), pf_ql, label='PyFlix', marker='o', linewidth=2, color='tab:blue')
ax2.plot(range(T_ql + 1), lm_ql, label='LinearMax', marker='x', linewidth=2, color='tab:
    orange')
ax2.set_title('Part C - Market Collapse: Death of QL', fontsize=14, fontweight='bold')
ax2.set_xlabel('Months', fontsize=12)
ax2.set_ylabel('Thousands of Subscribers', fontsize=12)
ax2.grid(True, alpha=0.3)
ax2.legend()

plt.tight_layout()
plt.show()
```

Python Programming

The function `np.linalg.eig(A)` computes the eigenvalues of a matrix A as well as its eigenvectors, sort of. The first output `eigenvalues` is a 1d-array of eigenvalues, repeated according to their multiplicities. The second output `eigenvectors` is as a 2d-array of "eigenvectors" for A, but the columns of `eigenvectors`, not the rows, are the eigenvectors of A, listed in the same order as the eigenvalues in `eigenvalues`. Thus, the 2d-array `eigenvectors` is the matrix P in the diagonalization of A, although this command will always normalize the columns of `eigenvectors`. While `eigenvalues` is only a list of the eigenvalues, it can be easily converted into the diagonal matrix D with the command `np.diag(eigenvalues)`. Lastly, the matrix P^{-1} is also easily found with the command `np.linalg.inv(eigenvectors)`. Thus, we can compute the diagonalization of a matrix by combining these NumPy commands. Be warned though, this method will only work (and can only be trusted) for diagonalizable matrices.

What is the command `np.eig()` doing then? If A is not diagonalizable, then `np.linalg.eig(A)` will still find the correct eigenvalues, but the matrix `eigenvectors` will not exactly be a set of eigenvectors. While the eigenvectors of A will be among the columns of `eigenvectors`, some of the columns will be *generalized eigenvectors*. For each eigenvalue of a matrix, there is always at least one associated eigenvector. The *algebraic multiplicity*, which comes from the characteristic polynomial, tells us the upper bound on the number of linearly independent eigenvectors for that eigenvalue, but the algebraic multiplicity is not always obtained. On the other hand, the *geometric multiplicity*, which is the dimenion of the eigenspace $\text{Eigen}(A, \lambda)$, is the number of linearly independent eigenvectors that actually occur. A matrix is diagonalizable if and only if these multiplicities agree for all eigenvalues. If A is not diagonalizable, then it can be almost diagonalized, that is, it is similar to what would otherwise be a diagonal matrix except for a few 1's along the subdiagonal, e.g. $A = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix}$. This matrix cannot be diagonalized, which means it does not have an eigenbasis. This is because the matrix $A - \lambda I_n$ does not have a large enough null space. But taking powers of this matrix, that is, $(A - \lambda I_n)^k$, the null space can enlarge, enough to include more vectors that sort of act like eigenvectors, so called **generalized eigenvectors** . Adding these generalized eigenvectors to the authentic eigenvectors allows for a **generalized eigenbasis**, which will convert the matrix into an almost diagonal matrix, called its **Jordan canonical form**. This topic of generalized eigenvectors and Jordan forms goes beyond the scope of this text, but it is important for the reader to know what these "extra" eigenvectors are in the case a matrix is not diagonalizable. In particular, the NumPy software can be fouled into believing a defective matrix is diagonalizable when it is not because of rounding errors from the imprecision of floating points.

Example 5.2.10. The code computes the diagonalization of the matrix $A = \begin{bmatrix} 1 & 3 & 3 \\ -3 & -5 & -3 \\ 3 & 3 & 1 \end{bmatrix}$ as seen in Example 5.2.8.

```
import numpy as np

A = np.array([[1,3,3],  #  Define matrix A
              [-3,-5,-3],
              [3,3,1]])
D, P = np.linalg.eig(A) # Calculate eigenvalues (D) and eigenvectors (P) of matrix A
Pinv = np.linalg.inv(P) # Calculate the inverse of matrix P
print(P,"\n",           # Print the results
      np.diag(D),"\n",
      Pinv,"\n=",
      P @ np.diag(D) @ Pinv,"\n")

B = np.array([[3,0,0],  # Define matrix B
       [0,2,1],
       [0,0,2]])
```

```
D, P = np.linalg.eig(B) # Calculate eigenvalues (D) and eigenvectors (P) of matrix B
Pinv = np.linalg.inv(P) # Calculate the inverse of matrix P
print(P,"\n",           # Print the results
      np.diag(D),"\n",
      Pinv,"\n=",
      P @ np.diag(D) @ Pinv,"\n!=",
      B)
```

namely:

```
[[-0.57735027 -0.28310993 -0.39631702] [[ 1.  0.  0.]    [[-1.73205081 -1.73205081 -1.73205081]
 [ 0.57735027 0.80479442 -0.42006444] * [ 0.  -2.  0.]  * [ 0.90508519 2.76948852 1.86440333]
 [-0.57735027 -0.52168449 0.81638146]] [ 0.  0.  -2.]]    [-0.64654958 0.54484238 1.19139196]]
   [[ 1.  3.  3.]
= [-3.  -5.  -3.]
   [ 3.  3.  1.]]
```

■

Exercises

(Go to Solutions)

For Exercises 1-12, diagonalize the matrix A. Answers may vary.

1. $\begin{bmatrix} -3 & -4 \\ 5 & 6 \end{bmatrix}$
$\lambda = 1, 2$

♠ 2. $\begin{bmatrix} 3 & -2 \\ 1 & 0 \end{bmatrix}$
$\lambda = 2, 1$

3. $\begin{bmatrix} 19 & -8 \\ 40 & -17 \end{bmatrix}$
$\lambda = 3, -1$

4. $\begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$
$\lambda = 5, 2$

5. $\begin{bmatrix} 4 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$
$\lambda = 4, 2, 3$

♠ 6. $\begin{bmatrix} 8 & -20 & 10 \\ 5 & -12 & 5 \\ 5 & -10 & 3 \end{bmatrix}$
$\lambda = -2, -2, 3$

7. $\begin{bmatrix} 1 & 2 & 1 \\ 6 & -1 & 0 \\ -1 & -2 & -1 \end{bmatrix}$
$\lambda = -4, 0, 3$

♠ 8. $\begin{bmatrix} 6 & 10 & 0 & -4 \\ 3 & 5 & 0 & -2 \\ 12 & 24 & -1 & -8 \\ 15 & 30 & 0 & -11 \end{bmatrix}$
$\lambda = 1, -1, -1, 0$

9. $\begin{bmatrix} 8 & 20 & 10 \\ 5 & -12 & 5 \\ 5 & 10 & 3 \end{bmatrix}$
$\lambda = 18, -17, -2$

10. $\begin{bmatrix} -4 & 0 & -4 \\ 3 & -4 & 3 \\ 2 & 0 & 2 \end{bmatrix}$
$\lambda = 0, -2, -4$

11. $\begin{bmatrix} 23 & -22 & 20 \\ 30 & -30 & 30 \\ 15 & -16 & 18 \end{bmatrix}$
$\lambda = 2, 3, -8$

12. $\begin{bmatrix} -4 & -3 & -2 \\ 0 & -4 & 0 \\ 4 & -3 & 2 \end{bmatrix}$
$\lambda = 0, -2, -4$

QUICK! For Exercises 13-13, given a diagonalization of matrix A, find a distinct diagonalization of A in LESS THAN 30 SECONDS! Answers may vary.

13. $\begin{bmatrix} 0 & -2 & -1 \\ 1 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 6 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -2 \end{bmatrix} \begin{bmatrix} -1 & 1 & -1 \\ -1 & 0 & -1 \\ 1 & 0 & 2 \end{bmatrix}$

For Exercises 14-14, determine whether the two matrices, A and B, are similar or not. Justify your answer.

14. $\begin{bmatrix} -4 & 0 & -4 \\ 2 & -4 & 2 \\ 2 & 0 & 2 \end{bmatrix}, \begin{bmatrix} -4 & -3 & -2 \\ 0 & -4 & 0 \\ 4 & -3 & 2 \end{bmatrix}$

15. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/diagonable.ipynb

"It's always good to take an orthogonal view of something. It develops ideas." – Ken Thompson

5.3 Orthogonal Diagonalization

Symmetric matrices are extremely simple to describe but come with extremely powerful matrix properties. Their analysis lies at the intersection of the eigenvalues of this chapter and the orthogonality of previous. Specializing the techniques of Section 5.2, the eigenbasis of a symmetric matrix can be always made orthonormal. We can also expect the eigenvalues of a symmetric matrix to always be real. These properties, as well as others, empower symmetric matrices to become important tools in linear algebra and its applications, such as the singular value decomposition which is an orthogonal decomposition of the Gram matrix.

5.3.1 Orthogonally Diagonalizable Matrices

Symmetric matrices interestingly bring together the theory of eigenvectors and inner products. Recall a real square matrix U is called *orthogonal* if $U^\top U = I$, that is, $U^T = U^{-1}$.

Theorem 5.3.1. *If A is a symmetric matrix, then any two eigenvectors with distinct eigenvalues are orthogonal.*

Example 5.3.2. We note that $A = \begin{bmatrix} 6 & -2 & -1 \\ -2 & 6 & -1 \\ -1 & -1 & 5 \end{bmatrix}$ is symmetric. It can also be checked that

$$\lambda_1 = 8 : \boldsymbol{v}_1 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}; \qquad \lambda_2 = 6 : \boldsymbol{v}_2 = \begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix}; \qquad \lambda_3 = 3 : \boldsymbol{v}_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

are all eigenvectors of A. Furthermore, $\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 = \boldsymbol{v}_1 \cdot \boldsymbol{v}_3 = \boldsymbol{v}_2 \cdot \boldsymbol{v}_3 = 0$. Normalizing these vectors:

$$\boldsymbol{u}_1 = \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{bmatrix}, \qquad \boldsymbol{u}_2 = \begin{bmatrix} -1/\sqrt{6} \\ -1/\sqrt{6} \\ 2/\sqrt{6} \end{bmatrix}, \qquad \boldsymbol{u}_3 = \begin{bmatrix} 1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix},$$

we get a diagonalization of A:

$$A = PDP^{-1} = \begin{bmatrix} -1/\sqrt{2} & -1/\sqrt{6} & 1/\sqrt{3} \\ 1/\sqrt{2} & -1/\sqrt{6} & 1/\sqrt{3} \\ 0 & 2/\sqrt{6} & 1/\sqrt{3} \end{bmatrix} \begin{bmatrix} 8 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ -1/\sqrt{6} & -1/\sqrt{6} & 2/\sqrt{6} \\ 1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} \end{bmatrix}$$

using an orthogonal matrix P. ∎

Definition 5.3.3. We say a real matrix A is **orthogonally diagonalizable** if there exists an orthogonal matrix P and diagonal matrix D such that $A = PDP^\top = PDP^{-1}$.

The matrix in the previous example is orthogonally diagonalizable. In fact, if A is orthogonally diagonalizable, then $A = PDP^\top$ and $A^\top = (PDP^\top)^\top = (P^\top)^\top D^\top P^\top = PD^\top P^\top = PDP^\top$, that is, $A^\top = A$. Thus, A is symmetric. The converse is also true.

Theorem 5.3.4. *A matrix A is orthogonally diagonalizable if and only if A is symmetric.*

The method of computing an orthogonal diagonalization is the same as any other diagonalization except we require that our basis of eigenvectors be orthonormal. Normalizing the eigenvectors is simple enough. On the other hand, we cannot simply apply the Gram-Schmidt procedure to an eigenbasis, because the end result may not be an eigenbasis. Instead, we must apply the Gram-Schmidt procedure to a basis for each distinct eigenspace. Since different eigenspaces are mutually orthogonal, the union of these orthogonal bases gives an orthogonal eigenbasis.

Example 5.3.5. Let $A = \begin{bmatrix} 3 & -2 & 4 \\ -2 & 6 & 2 \\ 4 & 2 & 3 \end{bmatrix}$. It can be shown that $\lambda = 7, -2$ are the eigenvalues of A and

$$\text{Nul}(A - 7I) = \text{Span}\left\{\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1/2 \\ 1 \\ 0 \end{bmatrix}\right\}, \qquad \text{Nul}(A + 2I) = \text{Span}\left\{\begin{bmatrix} -1 \\ -1/2 \\ 1 \end{bmatrix}\right\}$$

are the eigenspaces. Applying Gram-Schmidt to the eigenspace of $\lambda = 7$, we get the orthogonal basis

$$\text{Nul}(A - 7I) = \text{Span}\left\{\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1/4 \\ 1 \\ 1/4 \end{bmatrix}\right\}.$$

Therefore,

$$\left\{\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1/4 \\ 1 \\ 1/4 \end{bmatrix}, \begin{bmatrix} -1 \\ -1/2 \\ 1 \end{bmatrix}\right\}$$

is an orthogonal eigenbasis. After normalizing, the set

$$\left\{\begin{bmatrix} 1/\sqrt{2} \\ 0 \\ 1/\sqrt{2} \end{bmatrix}, \begin{bmatrix} -1/\sqrt{18} \\ 4/\sqrt{18} \\ 1/\sqrt{18} \end{bmatrix}, \begin{bmatrix} -2/3 \\ -1/3 \\ 2/3 \end{bmatrix}\right\}$$

is an orthonormal eigenbasis. Therefore,

$$A = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{18} & -2/3 \\ 0 & 4/\sqrt{18} & -1/3 \\ 1/\sqrt{2} & 1/\sqrt{18} & 2/3 \end{bmatrix} \begin{bmatrix} 7 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & -2 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & 0 & 1/\sqrt{2} \\ -1/\sqrt{18} & 4/\sqrt{18} & 1/\sqrt{18} \\ -2/3 & -1/3 & 2/3 \end{bmatrix}$$

is an orthogonal diagonalization of A. ∎

5.3.2 The Spectral Theorem

The set of eigenvalues of a matrix A is called the **spectrum** of A.

Theorem 5.3.6 (The Spectral Theorem for Symmetric Matrices)**.** *An $n \times n$ symmetric matrix A has the following properties:*

(i) A has n real eigenvalues, counting multiplicities;

(ii) The geometric and algebraic multiplicities of each eigenvalue of A are equal;

(iii) The eigenspaces of A are mutually orthogonal;

(iv) A is orthogonally diagonalizable.

5.3.3 Spectral Decomposition

Let A be a symmetric matrix. Then it has an orthogonal diagonalization given as:

$$\begin{aligned} A &= PDP^\top = \begin{bmatrix} \boldsymbol{u}_1 & \dots & \boldsymbol{u}_n \end{bmatrix} \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_1^\top \\ \vdots \\ \boldsymbol{u}_n^\top \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_1 & \dots & \boldsymbol{u}_n \end{bmatrix} \begin{bmatrix} \lambda_1 \boldsymbol{u}_1^\top \\ \vdots \\ \lambda_n \boldsymbol{u}_n^\top \end{bmatrix} \\ &= \lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^\top + \dots + \lambda_n \boldsymbol{u}_n \boldsymbol{u}_n^\top = \lambda_1 (\boldsymbol{u}_1 \otimes \boldsymbol{u}_1) + \dots + \lambda_n (\boldsymbol{u}_n \otimes \boldsymbol{u}_n). \end{aligned}$$

This last line is called a **spectral decomposition** of A. Each of matrices $B_i = \boldsymbol{u}_i \boldsymbol{u}_i^\top = \boldsymbol{u}_i \otimes \boldsymbol{u}_i$, the outer product of $\boldsymbol{u}_i$ with itself, is an $n \times n$ symmetric matrix with rank 1. The range of B_i is $\text{Span}\{\boldsymbol{u}_i\}$. Furthermore, $B_i B_j = 0$ if $i \neq j$ and $B_i^2 = B_i$, since $\{\boldsymbol{u}_1, \dots, \boldsymbol{u}_n\}$ is a orthonormal set.

Example 5.3.7. The matrix $A = \begin{bmatrix} 7 & 2 \\ 2 & 4 \end{bmatrix}$ is symmetric and has an orthogonal diagonalization given by

$$A = \begin{bmatrix} 2/\sqrt{5} & -1/\sqrt{5} \\ 1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix} \begin{bmatrix} 8 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 2/\sqrt{5} & 1/\sqrt{5} \\ -1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix}.$$

Let $P = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 \end{bmatrix} = \begin{bmatrix} 2/\sqrt{5} & -1/\sqrt{5} \\ 1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix}$. Then

$$\begin{aligned} \boldsymbol{u}_1 \otimes \boldsymbol{u}_1 = \boldsymbol{u}_1 \boldsymbol{u}_1^\top &= \begin{bmatrix} 2/\sqrt{5} \\ 1/\sqrt{5} \end{bmatrix} \begin{bmatrix} 2/\sqrt{5} & 1/\sqrt{5} \end{bmatrix} = \begin{bmatrix} 4/5 & 2/5 \\ 2/5 & 1/5 \end{bmatrix}, \\ \boldsymbol{u}_2 \otimes \boldsymbol{u}_2 = \boldsymbol{u}_2 \boldsymbol{u}_2^\top &= \begin{bmatrix} -1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix} \begin{bmatrix} -1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix} = \begin{bmatrix} 1/5 & -2/5 \\ -2/5 & 4/5 \end{bmatrix} \\ 8\boldsymbol{u}_1 \boldsymbol{u}_1^\top + 3\boldsymbol{u}_2 \boldsymbol{u}_2^\top &= 8 \begin{bmatrix} 4/5 & 2/5 \\ 2/5 & 1/5 \end{bmatrix} + 3 \begin{bmatrix} 1/5 & -2/5 \\ -2/5 & 4/5 \end{bmatrix} = \begin{bmatrix} 7 & 2 \\ 2 & 4 \end{bmatrix} = A. \quad \blacksquare \end{aligned}$$

Python Programming

The function `np.linalg.eigh(A)` computes the eigenvalues of a symmetric matrix A as well as its eigenvectors. It works similar to `np.linalg.eig(A)`, as the first output is the list of eigenvalues (`eigenvalues`) and the second output is a matrix whose columns are normalized eigenvectors (`eigenvectors`). Unless the defect of `np.linalg.eig()` discussed in Section 5.2, `np.linalg.eigh(A)` will always find the correct eigenvectors of A, since symmetric matrices always have an eigenbasis (no generalized eigenvectors needed here). Also, `eigenvectors` this time will be an orthogonal matrix, useful in orthogonal diagonalization. The reason for the `h` is that the complex analog of a (real) symmetric matrix is called a *Hermitian* matrix.

Example 5.3.8. The code computes the orthogonal diagonalization of the matrix $A = \begin{bmatrix} 3 & -2 & 4 \\ -2 & 6 & 2 \\ 4 & 2 & 3 \end{bmatrix}$ as seen in Example 5.3.5.

```
import numpy as np

A = np.array([[3,-2,4],   # Define matrix A
              [-2,6,2],
              [4,2,3]])
D, P = np.linalg.eigh(A)  # Calculate eigenvalues (D) and eigenvectors (P) of matrix A
print(P,"\n",             # Print the results
      np.diag(D),"\n",
      P.T,"\n=",
      P @ np.diag(D) @ P.T,"\n")
```

namely:

```
[[ 0.66666667 -0.74535599 0.        ]   [[ -2.  0.  0.]    [[ 0.66666667 0.33333333 -0.66666667]
[ 0.33333333 0.2981424 0.89442719   ] * [ 0.  7.  0. ]  * [-0.74535599 0.2981424 -0.59628479]
[-0.66666667 -0.59628479 0.4472136 ]]   [ 0.  0.  7. ]]   [ 0.  0.89442719 0.4472136 ]]
  [[ 3.  -2.  4.]
= [-2.  6.  2.]
  [ 4.  2.  3.]]
```

■

Exercises

(Go to Solutions)

For Exercises 1-6, compute an orthogonal diagonalization for the matrix A.

1. $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$
$\lambda = 3, 1$

2. $\begin{bmatrix} 17 & 5 \\ 5 & -7 \end{bmatrix}$
$\lambda = 18, -8$

3. $\begin{bmatrix} 4 & 3 & 4 \\ 3 & 4 & 0 \\ 4 & 0 & 4 \end{bmatrix}$
$\lambda = 9, 4, -1$

4. $\begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}$
$\lambda = 2, 2 \pm \sqrt{2}$

♠ 5. $\begin{bmatrix} 5 & 17 & 8 \\ 17 & 5 & 8 \\ 8 & 8 & 14 \end{bmatrix}$
$\lambda = 30, -12, 6$

♠ 6. $\begin{bmatrix} 5 & 3 & -3 & 3 \\ 3 & 5 & 3 & -3 \\ -3 & 3 & 5 & 3 \\ 3 & -3 & 3 & 5 \end{bmatrix}$
$\lambda = -4, 8, 8, 8$

For Exercises 7-12, compute a spectral decomposition for the matrix A, whose orthogonal diagonalization was computed in Exercises 1-6.

7. Exercise 1

8. Exercise 2

9. Exercise 3

10. Exercise 4

♠ 11. Exercise 5

♠ 12. Exercise 6

13. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/orthodiagonable.ipynb

"The ultimate value of life depends upon awareness and the power of contemplation rather than upon mere survival." – Aristotle

5.4 Singular Value Decomposition

Diagonalization, especially of symmetric matrices, is a powerful tool to study and work with matrices but comes with a serious restriction. A matrix is diagonalizable if and only if it has an eigenbasis, for which it is necessary the matrix be square. The singular value decomposition, or SVD, allows us to retain most of the benefit of a diagonalization without the matrix having an eigenbasis or even being square by instead finding the orthogonal diagonalization of the Gram matrix, which is always symmetric. The SVD is one of the most important linear algebraic tools of a data scientist, particularly because SVD has several applications in data analysis. For example, we can use SVD to analyze large data sets and find in those data sets. SVD is fundamental in machine learning tasks, data compression, image processing, and collaborative filtering. These are only a few possible applications. Some others will be discussed in Section 5.5.

5.4.1 Singular Values

Let A be an $m \times n$ matrix, that is, A might not be necessarily square. Can it be diagonalized like its square compatriots? Even if A cannot be diagonalized, its Gram matrix $A^\top A$ can be diagonalized. After all, $A^\top A$ is symmetric, meaning its diagonalization is the coolest. It is orthogonally diagonalizable! This orthogonal diagonalization of $A^\top A$ can modified into a pseudo-orthogonal diagonalization for A.

Let $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ be an orthonormal basis of eigenvectors of $A^\top A$ with corresponding eigenvalues $\lambda_1, \ldots, \lambda_n$. Note that

$$\|A\boldsymbol{v}_i\|^2 = (A\boldsymbol{v}_i)^\top (A\boldsymbol{v}_i) = (\boldsymbol{v}_i^\top A^\top)(A\boldsymbol{v}_i) = \boldsymbol{v}_i^\top (A^\top A\boldsymbol{v}_i) = \lambda_i(\boldsymbol{v}_i^\top \boldsymbol{v}_i) = \lambda_i(\boldsymbol{v}_i \cdot \boldsymbol{v}_i) = \lambda_i.$$

Since $\|A\boldsymbol{v}_i\|^2 \geq 0$, each eigenvalue of $A^\top A$ is a nonnegative real number. Reordering them if necessary, we may assume that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$. Let $\sigma_i = \sqrt{\lambda_i}$. The **singular values** of A are these values $\sigma_1, \sigma_2, \ldots, \sigma_n$. Thus, $\|A\boldsymbol{v}_i\| = \sigma_i$.

Theorem 5.4.1. *Let A be an $m \times n$ matrix. Suppose $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ is an orthonormal eigenbasis of $A^\top A$ such that $A^\top A\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i$ and $\lambda_1 \geq \ldots \geq \lambda_n \geq 0$. Then $\sigma_i = 0$ implies that $\boldsymbol{v}_i \in \mathrm{Nul}(A)$.*

Suppose r of these eigenvalues are nonzero. Then $\left\{\frac{1}{\sigma_1}A\boldsymbol{v}_1, \ldots, \frac{1}{\sigma_r}A\boldsymbol{v}_r\right\}$ is an orthonormal basis of $\mathrm{Col}\, A$. *In particular,* $\mathrm{rank}\, A = r$.

Proof. If $\sigma_i = 0$, then $\|A\boldsymbol{v}_i\| = 0$, which implies that $A\boldsymbol{v}_i = \boldsymbol{0}$. Thus, $\boldsymbol{v}_i \in \mathrm{Nul}(A)$, which proves the first result.

Let σ_r be the last nonzero singular value. Then $\mathrm{Col}\, A = \mathrm{Span}\{A\boldsymbol{v}_1, \ldots, A\boldsymbol{v}_n\} = \mathrm{Span}\{A\boldsymbol{v}_1, \ldots, A\boldsymbol{v}_r\}$. Since $\boldsymbol{v}_i \cdot \boldsymbol{v}_j = 0$ for $i \neq j$, we have

$$(A\boldsymbol{v}i) \cdot (A\boldsymbol{v}_j) = (A\boldsymbol{v}_i)^\top (A\boldsymbol{v}_j) = (\boldsymbol{v}_i^\top A^\top)(A\boldsymbol{v}_j) = \boldsymbol{v}_i^\top (A^\top A\boldsymbol{v}_j) = \boldsymbol{v}_i^\top (\lambda_j \boldsymbol{v}_j) = \lambda_j(\boldsymbol{v}_i \cdot \boldsymbol{v}_j) = 0.$$

Thus, $\{A\boldsymbol{v}_1, \ldots, A\boldsymbol{v}_r\}$ is orthogonal, which implies it is also linearly independent. We have then constructed a basis for $\mathrm{Col}\, A$. Lastly, as observed above $\|A\boldsymbol{v}_i\| = \sigma_i$, we conclude that $\left\{\frac{1}{\sigma_1}A\boldsymbol{v}_1, \ldots, \frac{1}{\sigma_r}A\boldsymbol{v}_r\right\}$ is an orthonormal basis for $\mathrm{Col}\, A$. □

Theorem 5.4.2. *Let A be an $m \times n$ matrix. If $\boldsymbol{v}$ is an eigenvector for $A^\top A$, then $A\boldsymbol{v}$ is an eigenvector for $AA^\top$.*

Proof. Let $u = A\boldsymbol{v}$, and let $A^\top A\boldsymbol{v} = \lambda\boldsymbol{v}$. Then

$$AA^\top u = (AA^\top)(A\boldsymbol{v}) = A(A^\top A\boldsymbol{v}) = A(\lambda\boldsymbol{v}) = \lambda(A\boldsymbol{v}) = \lambda\boldsymbol{u}.$$

Therefore, $\boldsymbol{u}$ is an eigenvector of $AA^\top$. □

Theorem 5.4.3 (The Singular Value Theorem)**.** *Let A be an $m \times n$ matrix with rank r. Let Σ be the $m \times n$ matrix whose diagonal entries are the singular values $\sigma_1, \ldots, \sigma_r$ of A, in descending order, and all other entries are zero, let $U = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \ldots & \boldsymbol{u}_m \end{bmatrix}$ be the $m \times m$ orthogonal matrix where $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m\}$ is an orthonormal eigenbasis of $AA^\top$, and let $V = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \ldots & \boldsymbol{v}_n \end{bmatrix}$ be the $n \times n$ orthogonal matrix where $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ is an orthonormal eigenbasis of $A^\top A$. Furthermore, assume that $\boldsymbol{u}_i = \dfrac{1}{\sigma_i} A\boldsymbol{v}_i$ for $1 \leq i \leq r$. Then*

$$A = U\Sigma V^\top.$$

*This factorization of A is called a **singular value decomposition** (or **SVD**).*

Proof. Note

$$U\Sigma = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \ldots & \boldsymbol{u}_m \end{bmatrix} \left[\begin{array}{ccc|c} \sigma_1 & & 0 & \\ & \ddots & & 0 \\ 0 & & \sigma_r & \\ \hline & 0 & & 0 \end{array}\right] = \begin{bmatrix} \sigma_1\boldsymbol{u}_1 & \sigma_2\boldsymbol{u}_2 & \ldots & \sigma_r\boldsymbol{u}_r & \mathbf{0} & \ldots & \mathbf{0} \end{bmatrix}$$

$$= \begin{bmatrix} A\boldsymbol{v}_1 & A\boldsymbol{v}_2 & \ldots & A\boldsymbol{v}_r & A\boldsymbol{v}_{r+1} & \ldots & A\boldsymbol{v}_n \end{bmatrix} = AV.$$

Therefore, $A = U\Sigma V^{-1} = U\Sigma V^T$. □

Example 5.4.4. Let $A = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 2 & 2 \end{bmatrix}$. Then $A^TA = \begin{bmatrix} 9 & 9 \\ 9 & 9 \end{bmatrix}$, which has eigenvalues $18 \geq 0$, with corresponding eigenvectors $\boldsymbol{v}_1 = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$, $\boldsymbol{v}_2 = \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$. Set $V = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}$.

We have that $\sigma_1 = \sqrt{18} = 3\sqrt{2}$ and $\sigma_2 = 0$ are the singular values of A. Set $\Sigma = \begin{bmatrix} 3\sqrt{2} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$.

Finally, $A\boldsymbol{v}_1 = \begin{bmatrix} 2/\sqrt{2} \\ 4/\sqrt{2} \\ 4/\sqrt{2} \end{bmatrix}$, $A\boldsymbol{v}_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$. Let $\boldsymbol{u}_1 = \dfrac{1}{3\sqrt{2}} A\boldsymbol{v}_1 = \begin{bmatrix} 1/3 \\ 2/3 \\ 2/3 \end{bmatrix}$. We next will extend $\{\boldsymbol{u}_1\}$ to an orthonormal basis of $\mathbb{R}^3$. The remaining eigenvectors of $AA^\top$ can be found to be $\boldsymbol{u}_2 = \begin{bmatrix} 2/3 \\ -2/3 \\ 1/3 \end{bmatrix}$ and

$\boldsymbol{u}_3 = \begin{bmatrix} 2/3 \\ 1/3 \\ -2/3 \end{bmatrix}$. Set $U = \begin{bmatrix} 1/3 & 2/3 & 2/3 \\ 2/3 & -2/3 & 1/3 \\ 2/3 & 1/3 & -2/3 \end{bmatrix}$. Therefore,

$$A = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 1/3 & 2/3 & 2/3 \\ 2/3 & -2/3 & 1/3 \\ 2/3 & 1/3 & -2/3 \end{bmatrix} \begin{bmatrix} 3\sqrt{2} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} = U\Sigma V^\top$$

is a singular value decomposition of A. ■

Python Programming

The function `np.linalg.svd(A)` computes the singular value decomposition of A. It has three output, namely `U`, `sigma`, and `VT`. The 2d-array `U` is the matrix U. The 2d-array `VT` is the matrix $V^\top$. Note that it does not output V directly. The 1d-array `sigma` is the list of singular values in descending order, including the correct number of zero values. To construct the matrix Σ, use the command `np.diag(sigma)`, although you may need to stack on some extra rows or columns of zeros to match the shape of A.

Example 5.4.5. The code computes the singular value decomposition of the matrix $A = \begin{bmatrix} -9 & 18 & 54 \\ -38 & -49 & -22 \end{bmatrix}$:

```
import numpy as np

A = np.array([[-9,18,54],   # Define matrix A
              [-38,-49,-22]])

U,sigma,VT = np.linalg.svd(A) # Perform Singular Value Decomposition (SVD)
print(U,"\n",                 # Print the results
      np.diag(sigma),"\n",
      VT.T,"\n")
```

namely:

```
[[ 0.6 0.8]   [[75.  0.]     [[ 0.33333333 -0.66666667 0.66666667]
[-0.8 0.6]] * [ 0.  45.]]  * [ 0.66666667 -0.33333333 -0.66666667]
                             [ 0.66666667 0.66666667 0.33333333]]
```

■

Exercises

(Go to Solutions)

For Exercises 1-6, compute a singular value decomposition for the matrix A.

1. $\begin{bmatrix} 1 & 1 & 3 & 5 & 8 \end{bmatrix}$
$\sigma_1 = 10$

♠ 2. $\begin{bmatrix} 1 & 2 & 2 & 4 \end{bmatrix}$
$\sigma_1 = 5$

♠ 3. $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$
$\sigma_1 = 2, \sigma_2 = 0$

♠ 4. $\begin{bmatrix} 3 & 4 \\ 0 & 3 \\ -4 & 0 \end{bmatrix}$
$\sigma_1 = \sqrt{37}, \sigma_2 = \sqrt{13}$

5. $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$
$\sigma_1 = \sqrt{6}, \sigma_2 = 1$

6. $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
$\sigma_1 = 3, \sigma_2 = 0, \sigma_3 = 0$

7. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/svd.ipynb

"Manifest plainness, embrace simplicity, reduce selfishness, have few desires." – Lao Tzu

5.5 Dimension Reduction

Data sets are often filled with extra "noise." Dimension reduction, techniques like principal component analysis or PCA, reduce high-dimensional data to a lower-dimensional space. This reduction is meant to reduce the complexity and retain essential information while minimizing noise and redundancy.

5.5.1 Singular Reduction

Let A be a $m \times n$ matrix of rank r. We saw previously how to compute the eigenvectors of $AA^\top$ using the eigenvectors of $A^\top A$, namely $\underline{u} = \dfrac{1}{\sigma} A\boldsymbol{v}$ when $\boldsymbol{v} \notin \text{Nul}(A)$. This gets you the first r eigenvectors of $AA^\top$. Because the remaining eigenvectors of $AA^\top$ correspond to zero eigenvalues, the product between eigenvalue and eigenvector will be $\mathbf{0}$. Since the product of any vector with $\mathbf{0}$ is likewise $\mathbf{0}$, instead of searching for the remaining eigenvectors of $AA^\top$ one could extend the orthonormal set $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r\}$ into any basis of $\mathbb{R}^m$, such as $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r, \boldsymbol{x}_{r+1}, \ldots, \boldsymbol{x}_m\}$. Applying the Gram-Schmidt process to this latter set produces the orthonormal basis $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r, \boldsymbol{u}_{r+1}, \ldots, \boldsymbol{u}_m\}$ for $\mathbb{R}^m$, which can be used to form the columns of U alternatively for no consequence on the SVD.

This analysis actually suggests that the eigenvectors of $AA^\top$ associated to zero singular values are irrelevant. These latter columns of U correspond to zero rows of Σ. We can remove these zero rows in Σ as well as these latter columns of U with no effect on the product as they are contributing nothing as it is. Likewise, the zero columns of Σ contribution nothing to the final product, and we could prune these zero columns from Σ by likewise pruning the latter rows of $V^\top$, or the latter columns of V. Let $0 \leq k \leq r$. Then we define the $m \times k$ matrix U_k as the submatrix of U consisting of only the first k columns of U, the $k \times k$ diagonal matrix Σ_k consisting of the first k singular values of A, and the $n \times k$ matrix V_k consisting of only the first k columns of V. Then

$$\begin{aligned} A &= U_r \Sigma_r V_r^\top = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \ldots & \boldsymbol{u}_r \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^\top \\ \boldsymbol{v}_2^\top \\ \vdots \\ \boldsymbol{v}_r^\top \end{bmatrix} \\ &= \sum_{i=1}^{r} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top = \sigma_1 \boldsymbol{u}_1 \boldsymbol{v}_1^\top + \sigma_2 \boldsymbol{u}_2 \boldsymbol{v}_2^\top + \ldots + \sigma_r \boldsymbol{u}_r \boldsymbol{v}_r^\top. \end{aligned}$$

This is known as the **reduced singular value decomposition**. Note that the matrix A is unaltered. It is the factors U, Σ, and V which are reduced in this formula.

Example 5.5.1. In Example 5.4.4, we saw

$$A = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 1/3 & 2/3 & 2/3 \\ 2/3 & -2/3 & 1/3 \\ 2/3 & 1/3 & -2/3 \end{bmatrix} \begin{bmatrix} 3\sqrt{2} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} = U\Sigma V^\top.$$

As $\text{rank}(A) = 1$, we can instead us the reduced SVD, namely

$$A = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 2/3 \\ 2/3 \end{bmatrix} \begin{bmatrix} 3\sqrt{2} \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} = U_1 \Sigma_1 V_1^\top. \qquad \blacksquare$$

If zero singular values contribute nothing to the matrix, then it begs to reason that small singular values only contribute a little to the matrix itself. It is for this reason that we arrange the singular values in descending order $\sigma_1 \geq \ldots \geq \sigma_r$. If $1 \leq k \leq r$, then let

$$\begin{aligned} A_k &= U_k \Sigma_k V_k^\top = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \ldots & \boldsymbol{u}_k \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^\top \\ \boldsymbol{v}_2^\top \\ \vdots \\ \boldsymbol{v}_k^\top \end{bmatrix} \\ &= \sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top = \sigma_1 \boldsymbol{u}_1 \boldsymbol{v}_1^\top + \sigma_2 \boldsymbol{u}_2 \boldsymbol{v}_2^\top + \ldots + \sigma_k \boldsymbol{u}_k \boldsymbol{v}_k^\top \end{aligned}$$

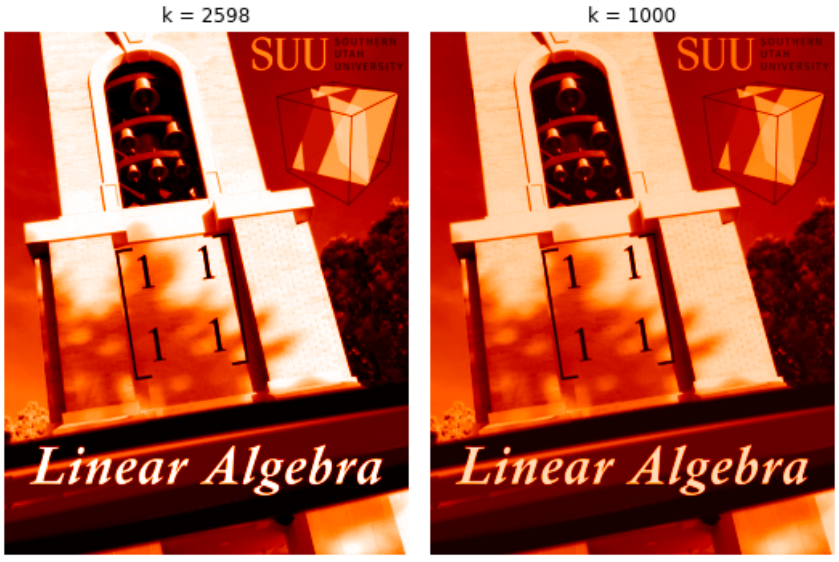
be the **rank-k singular reduction** of A. When the omitted singular values are small then $A_k \approx A$. More precisely, $\|A - A_k\| \approx 0$.

This singular reduction has an application toward data storage. Imagine a data file, such as an image, can be represented as an $m \times n$ matrix A. To store all of A would require storing the mn entries of A. On the the other hand, $A_k = U_k \Sigma_k V_k^\top$ could be stored instead by storing U_k, Σ_k, and V_k separately. Storing all the entries of U_k and V_k would be mk and nk values, respectively. As Σ_k is a diagonal matrix, we need only store the k diagonal entries. Thus, storing A_k would require $mk+nk+k = k(m+n+1)$ values. If k is chosen close to m or n, then $k(m+n+1) > mn$, which represents a poor storage scheme. In other words, choosing k too large provides to storage benefit. On the other hand, if $k < \dfrac{mn}{m+n+1}$, then $k(m+n+1) < mn$ and it would be cheaper to store A_k compared to A. Of course, if k is too small than the loss of data between A_k and A might be too much. Thus, a balance could be made between storage and data integrity.

Example 5.5.2. Consider the following image of the Bell Tower located on Southern Utah University's campus, below left. This image monochromatically can be expressed as a 3366×2598 matrix. This image then has $mn = 3366(2598) = 8{,}744{,}868$ pixels, or about 8.74 megapixels. The SVD of this image determines the rank of the matrix is (unsurprisingly) 2598. Note that

$$\frac{mn}{m+n+1} = \frac{3366(2598)}{3366+2598+1} \approx 1466.029841.$$

Thus, a singular reduction of rank less than 1466 would lead to data storage savings. Examining a few different singular reductions below, we see that $k = 1000$ produces a high quality alternative to the original image but only used $k(m+n+1) = 1000(3366+2598+1) = 5{,}965{,}000$ pixels, or about 5.97 megapixels. This is about a 31.8% reduction.





The following code can be used to produce the images in this example:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread as img
import urllib.request

#import image
url = r'https://raw.githubusercontent.com/emisseldine/MisselBooks/main/LinearAlgebra/images/Bell_Tower.jpg'
bell = np.asarray(img(urllib.request.urlopen(url),'jpg'))[:,:,0] #change 'jpg' based upon the file type of image
color = 'gist_heat'                                              #select a color map

# processing image
plt.figure(figsize=(9,6))
plt.imshow(bell,cmap=color)
plt.axis('off')
plt.show()
print(bell.shape,np.linalg.matrix_rank(bell))
#%%
U,sigma,VT = np.linalg.svd(bell) # Perform Singular Value Decomposition (SVD) on the image

# Define a function for image reconstruction using reduced SVD components
reduced_svd = lambda U,sigma,VT,k : U[:,:k] @ np.diag(sigma[:k]) @ VT[:k,:]
#%%
# Loop through different ranks k and display the reconstructed images
num_vals = np.concatenate((np.arange(1,5),
                           np.arange(5,50,5),
                           np.arange(50,525,25),
                           np.arange(1000,3000,500),
                           [len(sigma)]))
for k in num_vals:
```

```
    bell_blur = reduced_svd(U,sigma,VT,k)
    plt.figure(figsize=(9,6))
    plt.imshow(bell_blur,cmap=color)
    plt.title(f"k = {k}")
    plt.axis('off')
    plt.show()
```

5.5.2 Principal Component Analysis

Regression models grow exponentially more complex the more features a data set contains. Consider just a multivariate polynomial model. If the data set has n features $x_1, x_2, \ldots, x_n$, then a quadratic polynomial of this many variables:

$$a_{1,2}x_1^2 + a_{1,1}x_1 + a_{2,2}x_2^2 + a_{2,1}x_2 + b_{1,2}x_1x_2 + a_{3,2}x_3^2 + a_{3,1}x_3 + b_{1,3}x_1x_3 + b_{2,3}x_2x_3 \\ + a_{4,2}x_4^2 + a_{4,1}x_4 + b_{1,4}x_1x_4 + b_{2,4}x_2x_4 + b_{3,4}x_3x_4 + \ldots + a_0$$

would have $2n + \dfrac{n(n-1)}{2} + 1 = \dfrac{(n+2)(n+1)}{2}$ coefficients, which grows by a factor of roughly n^2. Similar reasoning shows that a multivariate polynomial of degree d would require rough n^d coefficients. This is already out of hand and we have not even considered non-polynomial features. More complicated models tend toward overfitting. Likewise, simpler models and smaller data sets make for faster calculations. But what can be done? Use less data?

Principal Component Analysis, or PCA, is a statistical method designed to reduce the dimension of a data set to avoid the complexities of a high-dimensional data set as discussed above, among other motives. This method relies on the linear algebra of the *covariance matrix*.

Definition 5.5.3. Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^m$. Then the **covariance** of $\boldsymbol{x}, \boldsymbol{y}$ is defined as

$$\operatorname{cov}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\widetilde{\boldsymbol{x}} \cdot \widetilde{\boldsymbol{y}}}{m}.$$

The **variance** of $\boldsymbol{x}$ is then defined as

$$\operatorname{var}(\boldsymbol{x}) = \operatorname{cov}(\boldsymbol{x}, \boldsymbol{x}) = \frac{\|\widetilde{\boldsymbol{x}}\|}{m}.$$

Given a set of vectors $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\} \subseteq \mathbb{R}^m$, the **covariance matrix** of X is the $n \times n$ matrix

$$\operatorname{cov}(X) = \begin{bmatrix} & \operatorname{cov}(\boldsymbol{x}_i, \boldsymbol{x}_j) & \end{bmatrix}.$$

Note that $\operatorname{var}(\boldsymbol{x}) = \operatorname{std}(\boldsymbol{x})^2$. As such, variance is a measure of the spread of a variable. Similarly, covariance is a measure of how two quantities vary with respect to each other. Inner products are linear algebra's way of measuring spread, as the inner product is directly correlated to the angle between two vectors. As such, if two vectors are parallel, then their inner product is exactly the product of their norms (ignoring signs), which is the maximum value of an inner product. If the vectors are normalized, this product is ± 1. Demeaning the vectors helps remove the bias that comes from vector norms in such products. Conversely, if two vectors are orthogonal, then their inner product is zero and they could not be any "farther apart" from each other.

When working with covariance, it is important to standardize the data, as hinted above. Let $T : \mathbb{R}^n \to \mathbb{R}^n$ be the map

$$T(\boldsymbol{x}_i) = \boldsymbol{z}_i = \frac{\widetilde{\boldsymbol{x}}}{\operatorname{std}(\boldsymbol{x})},$$

its *z-score*. Standardization removes the bias of comparing different quantities that have wildly different centers and spreads. It allows us to compare apples to apples, which is essential for covariance considerations.

We may identify the set of data vectors $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$ as an $m \times n$ matrix, whose columns are the vectors $\boldsymbol{x}_i$ themselves. Each column X would represent an attribute of the data and each row would represent a profile. Let the $m \times n$ matrix $Z = \begin{bmatrix} \boldsymbol{z}_1 & \boldsymbol{z}_2 & \ldots & \boldsymbol{z}_m \end{bmatrix}$ be X's standardization. Then

$$\text{cov}(Z) = \frac{1}{m} Z^\top Z,$$

that is, $\text{cov}(Z)$ is a scalar multiple of the Gram matrix of Z. Frankly, this scalar could be dropped with no signficant loss as scalar multiples only affect the eigenvalues of a matrix, not their eigenvectors. The latter of which will be our goal. Let then $Z = U\Sigma V^\top$ be an SVD of Z. Then the eigenvectors of $\text{cov}(Z)$ represent the directions of the vector space of greatest spread with respect to X, and the largest singular values of Z would represent the most powerful spread. The rank-k singular reduction of Z, namely Z_k, is then an approximation of the data set which forgets the least valuable directions of the data spread. Furthermore, ZV_k is a lower-dimensional projection of Z that maintains the same less of approximation as Z_k. While Z is an $m \times n$ matrix, ZV_k is an $m \times k$, that is, the same number of profiles are retained but k new coordinate features are created (the first k columns of V) to reduce the shape of Z while maintaining most of the diversity of Z. The column vectors of V are the eigenvectors of $\text{cov}(Z)$ and are called the **principal components** of Z. These are the newly created features by PCA, hence the name.

We summarize the method of Principal Component Analysis:

(i) Standardize the data set X, forming Z;

(ii) Compute an SVD of $Z = U\Sigma V^\top$, which is equivalent to finding the eigenvectors of $\text{cov}(Z)$;

(iii) Determine the appropriate level of reduction k;

(iv) Compute the projection ZV_k;

(v) Build a regression model on ZV_k.

Example 5.5.4. Consider the Iris data set of 150 flowers with 4 features x_1, x_2, x_3, x_4. Let X be this data set, and let Z be its standardization. Computing the singular values of Z gives

```
[2.26723243e+01 9.19449451e+00 1.19456361e+00 4.27451210e-15].
```

The four singular value is essentially 0 and will be discarded. The first two singular values are approximately 23 and 9 versus the third value which is approximately 1. Thus, nearly all the spread is accounted in the first two principal components. Thus, Z is reduced to $(ZV_2)^\top$, which is an 150×2 matrix, a data set of half the size but with nearly all the same diversity.

The following code can be used to run PCA on this data set:

```
import numpy as np
from sklearn.datasets import load_iris
X = iris_data
Z = np.array([(x-np.mean(x))/np.std(x) for x in X]) #standardize the iris data

_,sigma,VT = np.linalg.svd(Z)                       #PCA of X is found by computing SVD of Z
print(sigma)
#%%
k = 2                                               #rank-2 singular reduction is selected
PCA = Z @ VT[:k]                                  #projection onto rank-2 principal components
print(PCA)
```

■

Python Programming

Recall that the function `np.linalg.svd(A)` computes the singular value decomposition of A, with three output `U, sigma, VT`. The 1d-array `sigma` is only the list of singular values, not the matrix Σ. To construct the matrix Σ, we may use the command `np.diag(sigma)`, although you may need to stack on some extra rows or columns of zeros to match the shape of A. Because of this shaping issue and to more readily obtain the reduced SVD, use instead the command `U[:,:r] @ np.diag(sigma[:r]) @ VT[:r,:]` where $r = \text{rank}(A)$. For the purposes of singular reduction, the parameter `r` may be lowered to whichever rank chosen.

Example 5.5.5. The code computes the singular value decomposition of the matrix $A = \begin{bmatrix} -9 & 18 & 54 \\ -38 & -49 & -22 \end{bmatrix}$.

As $\text{rank}(A) = 2$, we also compute its singular reductions A_0, A_1, and $A_2 = A$:

```
import numpy as np

# computes the rank-k singular reduction of the matrix A, namely A_k, given the SVD of A,
    parameters U, sigma (list of singular values), VT, and the desired rank k
reduced_svd = lambda U,sigma,VT,k : U[:,:k] @ np.diag(sigma[:k]) @ VT[:k,:]

A = np.array([[-9,18,54],   # Define matrix A
              [-38,-49,-22]])

U,sigma,VT = np.linalg.svd(A) # Perform Singular Value Decomposition (SVD)
print(U,"\n",              # Print the results
      np.diag(sigma),"\n",
      VT.T,"\n")
for k in range(np.linalg.matrix_rank(A)+1):
    print(k, reduced_svd(U,sigma,VT,k),"\n")
```

namely:

```
[[ 0.6 0.8]   [[75.  0.]     [[ 0.33333333 -0.66666667 0.66666667]
[-0.8 0.6]] * [ 0.  45.]]  * [ 0.66666667 -0.33333333 -0.66666667]
                             [ 0.66666667 0.66666667 0.33333333]]

  0 [[0.  0.  0.]
   [0.  0.  0.]]

  1 [[ 15.  30.  30.]
    [-20. -40. -40.]]

  2 [[ -9.  18.  54.]
     [-38. -49. -22.]]
```

■

Exercises

(Go to Solutions)

For Exercises 1-6, compute a rank-k singular value decomposition for the matrix A. Note these same matrices appeared in Exercises 1-6 in Section 5.4. Please use the SVD computed there to assist here.

♠ 1. $\underset{k=1}{\begin{bmatrix} 1 & 1 & 3 & 5 & 8 \end{bmatrix}}$

♠ 2. $\underset{k=1}{\begin{bmatrix} 1 & 2 & 2 & 4 \end{bmatrix}}$

♠ 3. $\underset{k=1}{\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}}$

♠ 4. $\underset{k=1}{\begin{bmatrix} 3 & 4 \\ 0 & 3 \\ -4 & 0 \end{bmatrix}}$

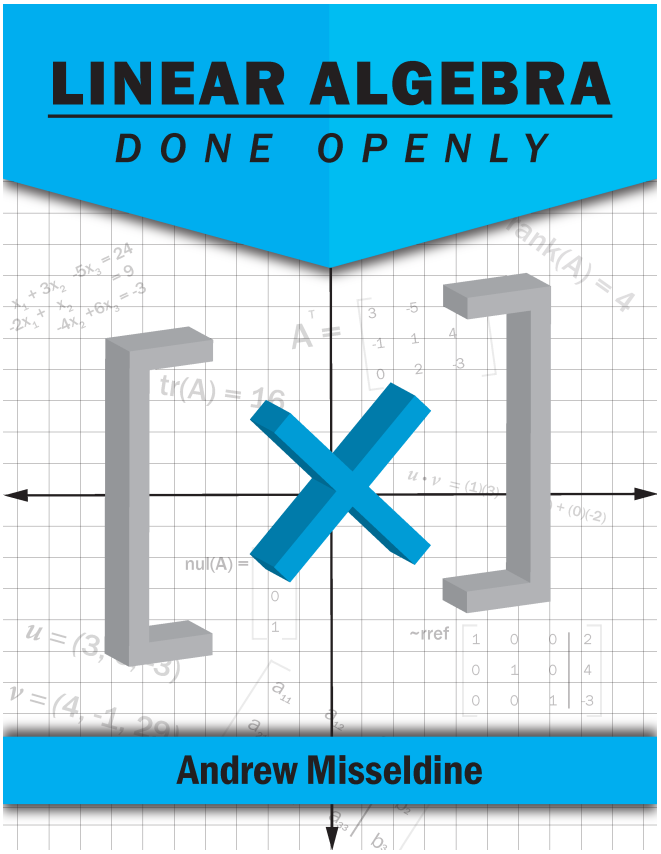
♠ 5. $\underset{k=1}{\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}}$

♠ 6. $\underset{k=1}{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}}$

7. Complete all of the Python-based exercises found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/dimreduce.ipynb

Python Projects

1. The image to the right displays a book cover that a student once submitted as a possible cover for our class textbook. The image is 3300×2550 square pixels. The image is primarily white and blue, which allows us to convert it to a monochromatic colormap with little loss to the original image. In this project, you will reduce the dimension of this image using a singular value reduction, such as explained in 5.5.1.



   (a) Compute the Singular Value Decomposition of the image matrix.

   (b) Using this SVD, compute the rank-k singular reduction of this matrix for

$$k = 1, 2, 3, 4, 5, 10, 25,$$
$$100, 200, 300, 400, 500,$$
$$1000, 1500, 2000.$$

   Display each image to the screen. Explain briefly how the quality of these images compare to the original.

   (c) Find a Singular Reduction of the image which reduces the data by approximately 10%, that is, the reduction is 90% the size of the original data. Display this image to the screen.

   Python code is started for you at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/bookcover.py

2. Recall in project **??**.5 a data set about 179 Italian wine samples. The data set contains 13 distinct attributes measured from chemical analysis of the wine samples (each measurement will be a positive real number). In this project, you will use Principle Component Analysis to analyze this data set, such as explained in Section 5.5.2. Python code is started for you at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/swinewine.py

   (a) Conduct Principal Component Analysis on these 13 atrributes. Print the 13 singular values to the screen.

   (b) Reduce the rank of the data using PCA on the affine model to make the training/test error reliably under 10%. Explicitly print the chosen rank to the screen. For comparison purposes, the prediction error for the affine model on the original data is presented too.

3. In this project, you continue the programs of Chapters **??** and **??** and write Python functions to compute a basis for Eigen(A, λ) and a diagonalization of A, when possible, such as explained in Sections 5.1.2 and 5.2.2. Python code is started for you at:
   https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/eigenspace.py

   (a) Write a Python function `Eigenspace(matrix, eigenvalue)` which takes input parameters: `matrix` is A and `eigenvalue` is λ. The function will return an array of vectors (hence a 2d-np.array): `eigenspace`, which is a basis for Eigen(A, λ) calculated from the RREF of $A - \lambda I_n$, accomplished by using the `rref` function written previously. Verify your code on Example 5.1.6.

   (b) Write a Python function `Diagonalize(matrix, eigenvalues)` which takes input parameters: `matrix` is A and `eigenvalues` is the set of eigenvalues of A. The function will return three matrices: P, D, P^{-1}, such that $A = PDP^{-1}$ and D is a diagonal matrix, accomplished by using

the functions `Eigenspace` and `matrix_inverse(matrix)`[i] written previously. Verify your code on Example 5.2.8.

[i]It is also acceptable to use `np.linalg.inv` to compute P^{-1}.

Appendix A

Python Primer

This chapter provides a short introduction to Python programming.[i] While Python is a general-purpose programming language, we will focus on the essential features needed to follow the examples and computations in this book. Our main numerical tool is `NumPy`[ii], a library for fast array and matrix operations. To include the `NumPy` package, include the following line as your first line of code (or at least before using in `NumPy` based objects or functions):

```
import numpy as np
```

The `import` command includes the named library to be used when executing code. The `as` command gives the library a nickname to make future calls shorter. Be aware that in Python the label `np` is used by virtually all programmers for `NumPy`.

A.1 Variables

Python is an **interpreted language**, meaning you can type commands into a Python prompt (or Jupyter notebook cell) and execute them immediately. Unlike other programming languages that use punctuation, such as semicolons or braces, to represent end of lines, Python uses **indentation** to define code blocks. Indentation must be consistent. In other programming languages, indentation is a best practice to make code more readible, such as indenting within a procedure or loop, but in Python it is required. If you misalign indentation, such as from a copy-and-paste that carries over text but not whitespace, Python will raise an error.

Example A.1. The code:

```
for i in range(3):
    print("Hello World", i) # command is inside for loop
print("End of Loop")        # command is outside for loop
```

produces the output:

```
Hello World 0
Hello World 1
Hello World 2
End of Loop
```

Note that because the first `print` is indented, it is included within the `for` loop and is executed three times. Conversely, the second `print` is without indentation, which tells the compiler to not execute that line until after the `for` loop is completed. Lastly, the pound/sharp sign `#` starts a **comment**. All text after `#` will be ignored by the compiler and is an effectively way to communicate with other programmers (including yourself) to explain code.

[i] Anaconda is a free distribution of Python that comes with scientific libraries (NumPy, pandas, matplotlib, etc.) and tools (Jupyter Notebook, Spyder) already included. It is the easiest way to get started with Python for data science. Go to https://www.anaconda.com/download to download Anaconda.

[ii] Pronounced "*Numb-Pie*."

Commonly, Python commands are computing various quantities, which when stored are called **variables**. Variables in Python are created by assignment or declaration, using an equal sign `=`. Multiple variables can be declared in the same line. To do so, separate the variable names on the left-hand side by commas and do the same separation of their assignments on the right-hand side.

There are many variable types in Python, but we mention four that will be commonly used. First, there is an `int` or integer, which represents a quantity that must be a whole number: positive, negative, or zero (think elements of $\mathbb{Z}$).

Second, a `float` or **floating point number** is a quantity in Python which represents a real number written in decimal form (think elements of $\mathbb{R}$). Floats are stored in computer memory using a fixed number of binary digits (according to the IEEE 754 standard), which means they can represent very large or very small numbers but only approximately. Python will automatically cast numerical variables as a float if there is a decimal or if in scientific notation, e.g. `1.23e4` is cast as `12300.0`.

Because floats are stored in binary with limited precision, some decimal numbers cannot be represented exactly. This can lead to small rounding errors in calculations. For example, the boolean `0.1 + 0.2 == 0.3` may surprisingly result as `False` since Python computes `0.1+0.2` to equal `0.30000000000000004`. This does not mean Python is "wrong," but that floating point arithmetic is approximate. In practice, comparisons of floats should allow for a tolerance, e.g. using `abs(x - y) < 1e-9` or with `NumPy`'s built-in function, `np.isclose(<float1>, <float2>)`.

Arithmetic for `int`'s or `float`'s can be computed using the standard operational symbols, namely, `+`, `-`, and `*` for addition, subtraction, and multiplication, respectively. The symbol `**` is used for exponents, as ^does a very different operation that we will not consider here. Python uses two different division operators for `float` and `int` division. The symbol `/` always performs **float division**, returning a decimal result (even if the numbers divide evenly). Conversely, the symbol `//` performs **integer division**, returning the quotient rounded down to the nearest whole number.

Example A.2. Note that `7 / 2` is float division and will compute as `3.5`. Conversely, `7 // 2` is integer division and will compute as `3`, since `3.5` rounds down to `3`. Note that given any integers $a, b \in \mathbb{Z}$ such that $a > 0$,[i], there exists unique integers $q, r \in \mathbb{Z}$ such that $b = aq + r$ and $0 \leq r < a$. The existence of q (quotient) and r (remainder) are guaranteed by the (long) division algorithm. Note that in Python, `b // a = q`. For example, since $7 = 2(3) + 1$, the integer quotient of 7 by 2 is 3.

How about `-7//2`? Since $-7 = 2(-4) + 1$, the integer quotien of -7 by 2 is -4. Note that -4 is rounded down from -3.5. Hence, Python computes `-7//2` as `-4`, not `-3`.

To obtain the integer remainder, use the symbol `%`. Hence, `7%2` is equal to 1.

Third, a `bool` or **boolean** is a data type with only two possible values: `True` or `False`. Booleans are typically the result of comparisons or logical operations, and they are fundamental in controlling program flow with `if` statements and loops, e.g. `3 < 5` (meaning the statement *three is less than five*) results in `True` and `7 >= 10` (meaning *seven is greater than or equal to ten*) results in `False`. Note, that since the equal sign `=` is used for declaring variables, a double equal sign `==` is necessary to check for equality, e.g. `10==2*5` (meaning *ten is equal to two times five*) results in `True`.

Python supports the standard logical operators: `and` which is `True` only if all conditions are `True`, `or` which `True` if at least one condition is `True`, and `not` which negates the boolean value. The symbol `!=` gives *not equal*.

Fourth, a `str` or **string** in Python is a sequence of characters (letters, digits, symbols, or spaces) enclosed in either single quotes '...' or double quotes "...".[ii] Strings are used to represent textual data. Note that the string "\n" creates a linebreak. Note that the command `print` displays to screen the string-cast of the

[i] If $a < 0$, then we can run the division algorithm for $-a$ and $-b$ and multiply the final equation by -1.

[ii] There is no effectual difference between single or double quotes in Python.

variable.

If is often useful to build strings using variables, which gives rise to an **f-string**.[iii] An `f`-string is formated similarly to a standard string, but we begin with `f“` instead of just “ (both types of string will end with ”). The new feature of an `f`-string is that inside of the quotes one can place {`<variable>`}. At this location of the string, `<variable>` will be printed.

Example A.3. The code:

```
x,y = 3,2
print(f"{x} + {y} = {x+y}")
```

produces the output:

```
3 + 2 = 5
```

There is no need to declare a variable type explicitly when the variable is assigned as Python automatically infers types, but you can **cast** or convert variable types from one type to another.

Example A.4. The code:

```
x = 5            # integer
y = 3.14         # float
z = True         # boolean
print(x,y,z)

x = int(3.7)     # 3
y = float(2)     # 2.0
z = bool(0)      # False
print(x,y,z)
```

produces the output:

```
5 3.14 True
3 2.0 False
```

Note that in the first declaration of variables, the compiler interpreted the quantities as an integer, floating point, and boolean, respectively. In the second declaration, we explicitly cast each quantity. For `x`, since we cast it as an `int`, the decimal portion was dropped (note that it was not rounded). Conversely, for `y`, since we cast it as a `float`, a `.0` decimal portion was concatenated onto the quantity. Finally, while Python reads boolean quantites as `True` or `False`, the integers `1`[iv] and `0` can be cast as a `bool`.

We mention two other observations about the above code. When declaring a variable in Python, write the name of the variable followed by an equal sign `=`, then followed by the quantity you are assigning. Like we saw in this example, a user can override the previous value of a variable by making a new declaration.

In a `np.array`, all entries must be cast as the same type.[v] Use the command `np.astype(<variable_type>)` to cast an entire array.

Example A.5. The code:

```
import numpy as np

a = np.array([1, 2, 3])                    # default cast as int
a_float = a.astype(float)                  # recast as float
print(a_float)
print(np.array([1.1,2.7,3.14]).astype(int)) # cast as int initially
```

produces the output:

```
[1.  2.  3.]
[1 2 3]
```

[iii] Here, the `f` is short for `format`, which is a built-in function for this exact situation, although `f`-string abbreviates `format` quite well and is thus more preferred.

[iv] Actually, any nonzero number casts as `True` when converted to `bool`.

[v] Lists on the other hand can have different casting for different entries in the same lists. This is both a asset and a bug, so watch out.

Note that a .astype is a function of a `np.array`. Hence, the vector $\boldsymbol{a}$ was automatically cast as an `int`-vector, but `a.astype(float)` creates a new vector, based upon $\boldsymbol{a}$, as a `float`-vector. Conversely, `np.array([1.1,2.7,3.14]).astype(int)` creates an `int`-vector based upon the temporary reference `np.array([1.1,2.7,3.14])`, which would have been cast as a `float`-vector without the `.astype` call.

When printed, lists will have entries separated by commas, but the commas are omitted in `np.array`'s.

Assigning arrays or lists creates **references**, not copies, in Python. As such, one needs to be cautious about how changing one variable could affect another.

Example A.6. The code:

```
import numpy as np

a = np.array([1, 2, 3])
b = a         # b points to same array
b[0] = 0
print(a)      # changed
print(b)

a = np.array([1, 2, 3])
b = a.copy() # To copy explicitly
b[0] = 0
print(a)      # unchanged
print(b)
```

produces the output:

```
[0 2 3]
[0 2 3]
[1 2 3]
[0 2 3]
```

Note that in the first try, $\boldsymbol{b}$ did not copy $\boldsymbol{a}$. Instead, it referenced $\boldsymbol{a}$. Hence, a change to $\boldsymbol{a}$ affected $\boldsymbol{b}$. In the second try, we copied $\boldsymbol{a}$ to create $\boldsymbol{b}$, so that a change to $\boldsymbol{a}$ did not affect $\boldsymbol{b}$. The second attempt requires more memory, but secured the integrity of the variables.

A.2 Loops and Conditionals

A `for` loop is a standard programming tool helpful when one needs to repeat a task, perhaps with some variation. The basic structure of a `for` loop is as follows:

```
for <index> in <range>:
     <commands>
```

It begins with the keyword `for`, indicating the start of the loop. After `for` comes the name of the iterator, which is a variable keeping track where we are in the loop. It is common to use a single Roman letter for this, e.g. `i`, `j`, or `k`. This is followed by the keyword `in` and then the list the iterator will range over. The command line of a `for` loop then ends with a colon. The body of the `for` loop is then contained within a single indent immediately following the colon. Removing this indent will end the loop at that spot.

Example A.1. The code:

```
grocery_list = ["apple", "orange", "carrot", "potato"]
for i in grocery_list:
    print(i)            # print each item individually
print(grocery_list)     # print the entire list
```

produces the output:

```
apple
orange
carrot
potato
['apple', 'orange', 'carrot', 'potato']
```

As seen in the previous example, the range of a `for` loop can be any list or array. There are three useful tools for creating index sets for a `for` loop. The simplest is `range(n)`, which creates the list `[0,1,2,...,n-1]`.

Next, there is `np.arange(<start>,<stop>,<step>)`, which works similar to `range` by generating a set of evenly spaced numbers. Unlike `range`, `np.arange` can control the starting and stoping bounds (as usual, `<start>` is included by `<stop>` is not included), as well as allow for a step-size different from one. If `<step>` is omitted, it defaults to 1. If only a single parameter is entered, then `<start>` defaults to 0.

Finally, there is `np.linspace(<start>,<stop>,<num>)`, which works very similar to `np.arange`, but instead of specifying the size of each step, `<num>` determines how many evenly-spaced steps there will be. Unlike `np.arange`, `np.linspace` does include `<stop>`, meaning that `np.linspace` will contain `<num>` $+ 1$ many entries. Both `<start>` and `<stop>` are required for `np.linspace`, although if `<num>` is omitted it defaults to 50. When using a non-integer step, such as 0.1, it is often better to use `np.linspace`.

Example A.2. The code:

```
import numpy as np

print(list(range(5)))                # creates list [0,1,2,3,4]
print(np.arange(0, 1, 0.2))          # creates array [0. 0.2 0.4 0.6 0.8]
print(np.linspace(0, 1, 5), "\n")    # creates array [0. 0.25 0.5 0.75 1.]

for i in range(5):                   # use range to iterate for loop
    print(i)
```

produces the output:

```
[0, 1, 2, 3, 4]
[0.  0.2 0.4 0.6 0.8]
[0.  0.25 0.5 0.75 1.  ]

0
1
2
3
4
```

It is often usual to a loop inside another loop, which is called **nesting**. For example, if running a `for` loop performs a *line* of commands, then a nested `for` loop, that is, a `for` loop inside a `for` loop, performs a *rectangle* of commands.

Example A.3. The code:

```
for i in range(4):
    for j in range(3):
        print(f"{i} x {j} =", i*j)  # inside the j-loop
    print("\n")                     # outside the j-loop
```

produces the output:

```
0 x 0 = 0
0 x 1 = 0
0 x 2 = 0

1 x 0 = 0
1 x 1 = 1
1 x 2 = 2

2 x 0 = 0
2 x 1 = 2
2 x 2 = 4

3 x 0 = 0
3 x 1 = 3
```

```
3 x 2 = 6
```

Note that, because of the indentation, the command `print("\n")` only executes after the entire `j`-loop is finished. This results in creating a linebreak after each grouping.

Often a certain block of code should only execute if a prerequisite condition is satisfied. This is the purpose of an `if` statement:

```
if <condition>:
    <commands>
```

The keyword `if` triggers the `if` statement. The compiler will then check the `<condition>`, which is a `bool`.[i] If `<condition>` is `True`, then `<commands>` will execute. Otherwise, the compiler skips this block entirely.

A modification of the `if` statement is an `if-else` statement:

```
if <condition>:
    <true-commands>
else:
    <false-commands>
```

If `<condition>` is `True`, then `true-commands>` will execute. If `<condition>` is `False`, then `false-commands>` will execute instead.

Like `for` loops, a nested `if` statement might be helpful, perhaps because sequential conditions ought to be checked. To simplify the nested conditionals, Python provides the keyword `elif`, which nest `if` statements:

```
if <1st-condition>:
    <1st-commands>
elif  <2nd-condition>:
    <2nd-commands>
else:
    <false-commands>
```

To process this nested `if`, the compiler first checks `<1st-condition>`. If it is true, then it executes `<1st-commands>` and skips the remaining blocks. If `<1st-condition>` is `False`, then the compiler checks `<2nd-condition>`. If it is true, then it executes `<2nd-commands>` and skips the remaining block(s). If `<2nd-condition>` is `False`, then it executes `<false-commands>`. The number of `elif`-additional conditionals is unlimited, and the final `else` is optional. Note that a colon is required after each conditional check.

Example A.4. The code:

```
x = 10
if x > 0:               # checks if x is positive
    print("Positive")   # if non-positive, checks if zero
elif x == 0:
    print("Zero")
else:                   #if not positive or zero, must be negative
    print("Negative")
```

produces the output:
`Positive`

A `while` loop repeatedly executes a block of code as long as a condition remains `True`. The syntax is:

```
while <condition>:
    <commands>
```

The loop will check the condition before each iteration. If the condition ever becomes `False`, the loop body will not execute. If `<condition>` is `False` to begin with, then the loop will be skipped entirely.

Example A.5. The code:

```
x = 0
while x < 5:
    print(x)
    x = x + 1  # a command line exists that changes the condition
```

[i] If the condition is not a `bool`, then it will be automatically cast as one.

produces the output:
`Positive`

Be careful to update variables inside a `while` loop; otherwise, the condition may never become `False`, creating an **infinite loop**.

Example A.6. The code:

```
while True:
    print("This will run forever...")
```

will produce an infinite loop. The compiler will continue to rewrite ``This will run forever...'' over and over again, until either the user force-stops it or the compiler crashes when it consumes all its memory.

In practice, `for` loops are often preferred when the number of iterations is known in advance, while `while` loops are useful when you repeat until a condition is met.

A.3 Functions

In programming, it is useful to reuse blocks of code. This is what a loop does, but it repeats the blocks back-to-back. A loop is not helpful when the blocks need to be repeated at far away branches of code. Instead, a **function** is a reusable block of code that takes inputs, performs some computation, and returns an output. Functions help you organize code, avoid repetition, and clarify the meaning of computations — much like functions in mathematics. To define a function in Python, we use the keyword `def`:

```
def <name>(<list of variables>):
    <commands>
    return <result>
```

Functions can take more than one input, but they must be separated by commas. Likewise, the return statement can have multiple output. The variables can be given a default values by following the variable name with an equal sign and assignment. If the parameter is skipped in the call of the function, then this variable gains the default value, which allows many parameters to be optionals. Because some parameters can be skipped in the call, it might be necessary to declare which parameter is which when calling the functions. If one omits `return`, the function executes but does not give back a value. These are often used for printing or modifying objects without a need to report what happened.

Example A.1. The code:

```
def square(x):
    return x**2

print(square(5))
```

produces the output:
25

A `lambda` **function** is a compact, one-line function, which is triggered by the keyword `lambda`:

```
<name_of_function> = lambda <variables separated by commas> : <return_statement>
```

`Lambda` functions are extremely helpful for simplifying Python code by encapsulating repeated calculations into a single command. They are commonly used to create a macro for mathematical formulas, among other reason.

Example A.2. The code:

```
square = lambda x : x**2

print(square(5))

def power(base, exponent=2):
    return base ** exponent

print(power(5))      # 25 (uses default exponent = 2)
print(power(5, 3))   # 125
```

```
10
11 def greet(name):
12     print("Hello,", name)
13
14 greet("Alice")   # prints "Hello, Alice"
```

produces the output:
25

Python has many **built-in** functions, for example, `len(x)` returns the length of the list or array, `sum(x)` returns the sum of elements of the list or array, `min(x)` or `max(x)` returns the minimum or maximum of elements of the list or array, respectively, `abs(x)` returns the absolute values of the quantity, `round(x,n)` returns x rounded to n decimal places, and `type(x)` returns the type of object. Many packages, like `NumPy`, have built-ins. One, generally speaking, should resort to using built-ins before building a new function, as this will likely save both the programmer and the compiler time. For example, `np.sum` or `np.min` execute much faster than the standard `sum` or `min`, which are faster than coding it oneself.

A Jupyter notebook of exercises from this section can be found at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/workbooks/python.ipynb

Appendix B

The Substitution and Elimination Methods

Solving linear systems by graphing is very error-prone and problematic, even with the use of technology! Although the geometric approach might be necessary to comprehend solutions to linear systems, algebraic methods prove far superior to solving linear systems. In this appendix, we will discuss two such methods typically presented in a pre-linear algebra setting, beginning with substitution.

B.1 The Substitution Method

The Substitution Method

1. Solve one equation for one of the variables.
2. Substitute the expression for this variable into the other equation.
3. Once you have determined one assignment, back substitute it into the equation from 1.

Example B.1. Solve the system by substitution.

$$\begin{cases} 2x + y = 1 \\ 3x + 4y = 14, \end{cases}$$

We will solve the first equation with respect to y. This gives us $y = 1 - 2x$. We next will substitute this value of y into the second equation, giving:

$$3x + 4y = 14 \quad \Rightarrow \quad 3x + 4(1 - 2x) = 14 \quad \Rightarrow \quad 3x + 4 - 8x = 14 \quad \Rightarrow \quad -5x = 10 \quad \Rightarrow \quad x = -2.$$

We next back substitute $x = -2$ into the above equation. This gives $y = 1 - 2(-2) = 1 + 4 = 5$. Therefore, the solution is $\boxed{(-2, 5)}$.

Example B.2. Solve

$$\begin{cases} x + y - z = -1 \\ 4x - 3y + 2z = 16 \\ 2x - 2y - 3z = 5. \end{cases}$$

We will solve the first equation with respect to z : $z = x + y + 1$. We now substitute this expression into *both* of the remaining equations.

$$\begin{cases} 4x - 3y + 2(x + y + 1) = 16 \\ 2x - 2y - 3(x + y + 1) = 5 \end{cases} \sim \begin{cases} 6x - y + 2 = 16 \\ -x - 5y - 3 = 5 \end{cases} \sim \begin{cases} 6x - y = 14 \\ -x - 5y = 8. \end{cases}$$

We now have to solve a system of 2 equations with 2 unknowns. Again using substitution, notice that $y = 6x - 14$. Plugging this into the last equation gives

$$-x - 5(6x - 14) = 8 \quad \Rightarrow \quad x - 30x + 70 = 8 \quad \Rightarrow \quad -31x = -62 \quad \Rightarrow \quad x = 2.$$

We now substitute this value into the above equation for y and get $y = 6(2) - 14 = 12 - 14 = -2$. Lastly, we substitute both of these values back into above equation for x and get $z = (2) + (-2) + 1 = 1$. Therefore, the solution of the system is $\boxed{(2, -2, 1)}$.

Algebraically, one discovers that a particular system is inconsistent if during the process of solving one runs into a contradiction or a individual solution which has no solution.

Example B.3. Find all solutions for

$$\begin{cases} 8x - 2y = 5 \\ -12x + 3y = 7. \end{cases}$$

We will solve the variable x in the first equation, giving $x = \frac{1}{4}y + \frac{5}{8}$. Substituting this into the second equation gives

$$-12\left(\frac{1}{4}y + \frac{5}{8}\right) + 3y = 7 \quad \Rightarrow \quad -3y - \frac{15}{2} + 3y = 7 \quad \Rightarrow \quad -\frac{15}{2} = 7,$$

which is a contradiction. Therefore, the system is $\boxed{\text{inconsistent}}$ and has no solutions.

B.2 The Elimination Method

The Elimination Method

1. Choose a variable to eliminate and adjust the coefficients of said variable so they are alternating.
2. Add the equations and solve for the remaining variable.
3. Back substitute the variable value into one of the original equations.

Now, we try the elimination method to solve a linear system.

Example B.1. Find all solutions by elimination for

$$\begin{cases} 2x - 3y = 4 \\ 4x + 5y = 3. \end{cases}$$

We will eliminate the variable y. Notice that the coefficients of y are already alternating but we need a common multiple. Between 3 and 5, the least common multiple is $3(5) = 15$. Thus, we will multiply the first equation by 5 and the second by 3. This gives

$$\begin{cases} 10x - 15y = 20 \\ 12x + 15y = 9. \end{cases}$$

If we add the equations, we get $22x = 29 \Rightarrow x = \frac{29}{22}$. Plugging this x value into either equation will give the y-value. Alternatively, we could start the elimination process over again by canceling out the x this time. To do so, we will multiply the first equation by -2 and the second by nothing. This gives

$$\begin{cases} -4x + 6y = -8 \\ 4x + 5y = 3. \end{cases}$$

If we add the equations, we get $11y = -5 \Rightarrow y = -\frac{5}{11}$. Therefore, $\boxed{\left(\frac{29}{22}, -\frac{5}{11}\right)}$ is the solution of the system.

Example B.2. Solve

$$\begin{cases} x - 2y - z = 8 \\ 2x - 3y + z = 23 \\ 4x - 5y + 5z = 53. \end{cases}$$

First, we will eliminate x from the system, which means eliminating x from the first pair of equations and from the second pair of equations. With the first pair, multiply the first equation by -2 : $-2x+4y+2z = -16$. When we add the first 2 equation together, we get $y + 3z = 7$. Next, we eliminate x from the second pair of equations. We will multiply the second equation by -2, which gives $-4x + 6y - 2z = -46$. After we add this equation with the third, we get $3y + 9z = 21$. We now have to solve the 2×2 system

$$\begin{cases} y + 3z = 7 \\ 3y + 9z = 21. \end{cases}$$

But upon further inspection, we notice that the first equation is 3 times the second equation. Therefore, the equations are dependent and the system has infinitely many solutions.

In order to get the general form of the solution, notice that we have $y = 7 - 3z$ and $x = 2y + z + 8 = 2(7 - 3z) + z + 8 = -5z + 22$. Therefore, the general solution has the form $\boxed{(-5z + 22, 7 - 3z, z)}$.

Appendix C

Further Topics on Orthogonality

C.1 Angles Between Vectors

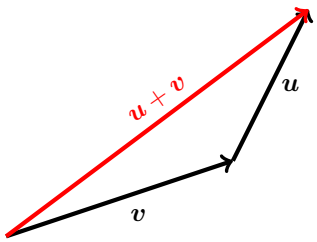
C.1 Inner Product Spaces

Theorem C.1 (Cauchy-Schwarz Inequality)**.** *For all* $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$*,*

$$|\boldsymbol{u} \cdot \boldsymbol{v}| \leq \|\boldsymbol{u}\|\|\boldsymbol{v}\|.$$

Theorem C.2 (The Triangle Inequality)**.** *For all* $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$*,*

$$\|\boldsymbol{u} + \boldsymbol{v}\| \leq \|\boldsymbol{u}\| + \|\boldsymbol{v}\|.$$



Theorem C.3 (Chebyshev's Inequality)**.** *Let* $\boldsymbol{x} \in \mathbb{R}^n$ *and* $a > 0$*. Suppose there are at least* k *entries of* $\boldsymbol{x}$ *which satisfy the inequality* $|x_i| \geq a$*. Then*

$$\frac{\|\boldsymbol{x}\|^2}{a^2} \geq k.$$

In the case that $k = n$, Chebyshev's inequality $\dfrac{\|\boldsymbol{x}\|}{a^2} \geq n$ tells us nothing. After all, if $|x_i| \geq a$ for all i, then

$$\|\boldsymbol{x}\| = \sqrt{x_1^2 + \ldots + x_n^2} \geq \sqrt{a^2 + \ldots a^2} = \sqrt{na^2} = a\sqrt{n}$$

and

$$\frac{\|\boldsymbol{x}\|^2}{a^2} \geq \frac{(\sqrt{n}a)^2}{a^2} = n \geq n,$$

which is certainly a trivial observation. A nontrivial observation is the case when $a > \|\boldsymbol{x}\|$. Note that

$$k \leq \frac{\|\boldsymbol{x}\|^2}{a^2} < \frac{\|\boldsymbol{x}\|^2}{\|\boldsymbol{x}\|^2} = 1.$$

As k is a positive integer less than 1, we have a contradiction. Therefore, no single entry of a vector can be larger than the norm of the vector. Intuitively this is clear, but Chebyshev's Inequality provides the proof of this observation.

Theorem C.4 (The Law of Cosines)**.** *Let* $\boldsymbol{u}, \boldsymbol{y} \in \mathbb{R}^n$*. Then*

$$\boldsymbol{u} \cdot \boldsymbol{y} = \|\boldsymbol{u}\|\|\boldsymbol{y}\| \cos\theta,$$

where θ *is the angle between the two line segments from the origin to the points identified with* $\boldsymbol{u}$ *and* $\boldsymbol{y}$*.*

If $\angle(\boldsymbol{u}, \boldsymbol{v})$ denotes the angle between vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, then

$$\angle(\boldsymbol{u}, \boldsymbol{v}) = \cos^{-1}\left(\frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|}\right). \tag{C.1.1}$$

From the previous theorem, if $\angle(\boldsymbol{u}, \boldsymbol{v}) = 90°$, then the inner product $\boldsymbol{u} \cdot \boldsymbol{v} = 0$. In this case, this is why we say the vectors are **perpdicular** (or **orthogonal**), denoted $\boldsymbol{u} \perp \boldsymbol{v}$. We say two vectors $\boldsymbol{u}$, $\boldsymbol{v}$ are **parallel** if $\angle(\boldsymbol{u}, \boldsymbol{v}) = 0°$, denoted $\boldsymbol{u} \parallel \boldsymbol{v}$.

Example C.5. Compute $\angle(\boldsymbol{u}, \boldsymbol{v})$ where $\boldsymbol{u} = (6, -1), \boldsymbol{v} = (1, 4) \in \mathbb{R}^2$.

$$\theta = \cos^{-1}\left(\frac{6(1) - 1(4)}{\sqrt{6^2 + (-1)^2}\sqrt{1^2 + 4^2}}\right) = \cos^{-1}\left(\frac{6 - 4}{\sqrt{36 + 1}\sqrt{1 + 16}}\right) = \cos^{-1}\left(\frac{2}{\sqrt{37}\sqrt{17}}\right) \approx \boxed{85.43°}$$

Theorem C.6. *For all vector $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ and scalars $s, t \in \mathbb{R}$,*

$$\angle(s\boldsymbol{u}, t\boldsymbol{v}) = \angle(\boldsymbol{u}, \boldsymbol{v}).$$

Example C.7. Consider two points on the surface of a sphere. One can measure the distance between these two points (like cities on Earth) by the shortest arc across the surface between these two points. Of course, if this same sphere were to increase its radius, then the distance between these two points would increase proportional to the increase of radius. Conversely, if the radius shrunk, the distance between the two points would also shrink. On the other hand, if we consider the vectors that connect the center of the sphere to these two points, respectively, note that the angle between the two vectors is fixed no matter how large or small the radius becomes. As such, the angle between two vectors, or the points' *spherical distance*, can be a potential way of resolving the scaling bias associated to distance between vectors.

C.2 Idempotent Matrices

Definition C.8. Let $P : \mathbb{R}^n \to \mathbb{R}^n$ be a linear transformation. We say that P is a **projection** if $P \circ P = P$.

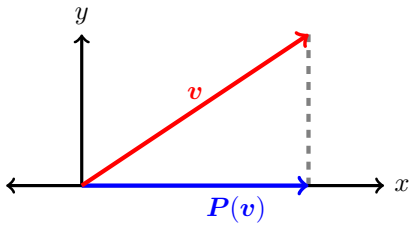
Let A be the standard matrix of P. Then the property that $P \circ P = P$ translate to mean that $A^2 = A$. Any matrix (necessarily a square) which satisfies this identity is called an **idempotent** matrix and is necessarily the standard matrix of a projection.

Let $\boldsymbol{x} \in \mathbb{R}^n$. Then $\boldsymbol{y} = P(\boldsymbol{x})$ is an arbitrary element of the range of P. By definition, we have that

$$P(\boldsymbol{y}) = P(P(\boldsymbol{x})) = P \circ P(\boldsymbol{x}) = P(\boldsymbol{x}) = \boldsymbol{y},$$

that is, a projection is exactly a linear transformation that fixes its image. Essentially this means that while some of the coordinates of $\boldsymbol{x}$ are unaltered the other coordinates are forgotten.

Geometrically, a projection is a map from the ambient space which projects onto some subspace (the range of the projection). In the process of projecting into the subspace, some information about the vectors is removed so that they can fit inside the smaller space. The image of a vector $\boldsymbol{v}$ is the shadow cast by $\boldsymbol{v}$ onto the subspace associated with the projection P.



Example C.9. For example, the matrices

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

are easily seen to be idempotent matrices and hence correspond to projections. The first matrix is the projection in $\mathbb{R}^2$ onto the x-axis where the y-coordinate is forgotten and replaced with 0. Any point already on the x-axis has the form $(x, 0)$ and is unaffected by the projection.

The second matrix is a projection in $\mathbb{R}^3$ onto the y-axis, where the x- and z-coordinates are discarded. Likewise, the third matrix is a projection in $\mathbb{R}^3$ onto the xy-plane, where the z-coordinate is the only information forgotten. In either case, elements already in the subspace are not altered by the projection.

Example C.10. The matrices

$$\begin{bmatrix} 2 & 2 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} -8 & 4 & 1 \\ -18 & 9 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

are also idempotent matrices. (Convince yourself of this). The first matrix corresponds to projection of $\mathbb{R}^2$ onto the line spanned by $(2, -1)$, that is, the line $y = -\frac{1}{2}x$. The second matrix corresponds to projection of $\mathbb{R}^3$ onto the plane spanned by $(4, 9, 0)$ and $(1, 2, 1)$, that is, the plane $9x - 4y - z = 0$.

If A is idempotent, then multiplication by A is projection onto $\mathrm{Col}(A)$.

Definition C.11. Let $\boldsymbol{u} = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix}, \boldsymbol{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^n$. Then the **outer product** (or **matrix product**) of $\boldsymbol{u}$ and $\boldsymbol{v}$, denoted $\boldsymbol{u} \otimes \boldsymbol{v}$, is the $n \times n$ matrix of the form $\begin{bmatrix} u_i v_j \end{bmatrix}$.

More compactly, we have that $\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^\top$ (or $\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^*$ for complex vectors). This highly resembles the definition of the inner product $\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u}^\top \boldsymbol{v}$ (or $\boldsymbol{u}^* \boldsymbol{v}$ for complex vectors). Despite this similarity, the outer product is a matrix and the inner product is a scalar. Neither of them is a vector. Of course, these two products are interconnected, justifying the complementary names, by the following formula:

$$(\boldsymbol{u} \otimes \boldsymbol{v})\boldsymbol{w} = (\boldsymbol{v} \cdot \boldsymbol{w})\boldsymbol{u}.$$

Proof.

$$(\boldsymbol{u} \otimes \boldsymbol{v})\boldsymbol{w} = (\boldsymbol{u}\boldsymbol{v}^\top)\boldsymbol{w} = \boldsymbol{u}(\boldsymbol{v}^\top \boldsymbol{w}) = \boldsymbol{u}(\boldsymbol{v} \cdot \boldsymbol{w}) = (\boldsymbol{v} \cdot \boldsymbol{w})\boldsymbol{u}. \qquad \square$$

Essentially, the adjectives inner versus outer can be a mnemonic to describe the location of the $^\top$ (or *).

Example C.12. Let $\boldsymbol{u} = \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$ and $\boldsymbol{v} = \begin{bmatrix} 2 \\ -2 \\ 3 \\ -6 \end{bmatrix}$. Then $\boldsymbol{u} \otimes \boldsymbol{v} = \begin{bmatrix} 2 & -2 & 3 & -6 \\ -4 & 4 & -6 & 12 \\ 0 & 0 & 0 & 0 \end{bmatrix}$.

Theorem C.13. *Let $\boldsymbol{u}$ be a unit vector. Then $A = \boldsymbol{u} \otimes \boldsymbol{u}$ is an idempotent matrix, and the matrix transformation $\boldsymbol{x} \mapsto A\boldsymbol{x}$ is a projection onto the subspace* $\mathrm{Span}\{\boldsymbol{u}\}$.

Example C.14. Consider the vector $\boldsymbol{v} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Its normalization is $\boldsymbol{u} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2 \\ \sqrt{2}/2 \end{bmatrix}$. Then the matrix $A = \boldsymbol{u} \otimes \boldsymbol{u} = \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix}$ is an idempotent matrix since

$$A^2 = \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix} \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix} = \begin{bmatrix} 1/4+1/4 & 1/4+1/4 \\ 1/4+1/4 & 1/4+1/4 \end{bmatrix} = A.$$

The mapping $\boldsymbol{x} \mapsto A\boldsymbol{x}$ is the projection of a vector in $\mathbb{R}^2$ onto the line $y = x$.

C.2 Orthogonal Decomposition

C.1 Orthogonal Complements

Theorem C.1. *Let W be a subspace of $\mathbb{R}^n$. Let $W^\perp = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{w} \cdot \boldsymbol{x} = 0,\ \forall \boldsymbol{w} \in W\}$, called the **orthogonal complement** of W. Then $W^\perp$ is also a subspace of $\mathbb{R}^n$.*

Example C.2. Let W be the z-axis in $\mathbb{R}^3$. Then $W^\perp$ is the xy-plane.

Theorem C.3. *Let A be an $m \times n$ matrix. Then*

$$(\operatorname{Row} A)^\perp = \operatorname{Nul} A.^{\text{i}}$$

Example C.4. Let W be the subspace of $\mathbb{R}^6$ spanned by the vectors

$$\boldsymbol{w}_1 = (1, 3, -2, 0, 2, 0),\ \boldsymbol{w}_2 = (2, 6, -5, -2, 4, -3),\ \boldsymbol{w}_3 = (0, 0, 5, 10, 0, 15),\ \boldsymbol{w}_4 = (2, 6, 0, 8, 4, 18).$$

Construct a base for $W^\perp$.

We can arrange these vectors into a matrix A:

$$A = \begin{bmatrix} 1 & 3 & -2 & 0 & 2 & 0 \\ 2 & 6 & -5 & -2 & 4 & -3 \\ 0 & 0 & 5 & 10 & 0 & 15 \\ 2 & 6 & 0 & 9 & 4 & 18 \end{bmatrix} \sim \begin{bmatrix} 1 & 3 & 0 & 4 & 2 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

such that $W = \operatorname{Row}(A)$. By the previous result, $W^\perp = \operatorname{Row}(A)^\perp = \operatorname{Nul}(A)$. As we know how to find a basis for a null space given the echelon form of the matrix,

$$W^\perp = \operatorname{Span}\{(-3, 1, 0, 0, 0, 0),\ (-4, 0, -2, 1, 0, 0),\ (-2, 0, 0, 0, 1, 0)\}.$$

Despite our reluctance to use any fields other than $\mathbb{R}$ and $\mathbb{C}$, the entirety of this section[ii] is transferable to any field using the dot product or Hermitian product.

[i] For real vector spaces, recall that $\operatorname{Row}(A) = \operatorname{Col}(A^\top)$. With the perspective, Theorem C.3 tells us that $\operatorname{Nul}(A)^\perp = \operatorname{Col}(A^\top)$. Hence, the orthogonal complement $\perp$ and transpose $\top$ are, in same way, dual notions of each other.

[ii] There is one important partial exception to this claim, namely the Pythagorean Theorem. We say this as a partial exception because while the algebraic statement and proof still remain valid (note that $\|\boldsymbol{u}\|^2 = \boldsymbol{u} \cdot \boldsymbol{u}$ and is valid for any vector space which we desire to introduce the dot product), the geometric interpretations about the norm (if it is defined via the dot product) do not necessarily transfer to general vector spaces.

C.2 Orthogonal Decompositions

Theorem C.5 (The Orthogonal Decomposition Theorem)**.** *Let $\boldsymbol{v} \in \mathbb{R}^n$ and $W \leq \mathbb{R}^n$. Then there exists unique vectors $\boldsymbol{v}_{\parallel} \in W$ and $\boldsymbol{v}_{\perp} \in W^{\perp}$ such that*

$$\boldsymbol{v} = \boldsymbol{v}_{\parallel} + \boldsymbol{v}_{\perp}.$$

Example C.6. Let $\boldsymbol{u}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, $\boldsymbol{u}_2 = \begin{bmatrix} -5 \\ 4 \\ -1 \end{bmatrix}$, and $\boldsymbol{y} = \begin{bmatrix} -9 \\ 20 \\ -1 \end{bmatrix}$. Let $W = \text{Span}\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$. Since $\boldsymbol{u}_1 \cdot \boldsymbol{u}_2 = 0$, $\{\boldsymbol{u}_1, \boldsymbol{u}_2\}$ is an orthogonal basis for W. Using the orthogonal decomposition of $\boldsymbol{y}$ onto W, we can express $\boldsymbol{y}$ as a sum of a vector from W and a vector from $W^{\perp}$.

Now,

$$\boldsymbol{w}_1 = \widehat{\boldsymbol{y}} = \text{proj}_W \boldsymbol{y} = \frac{\boldsymbol{u}_1 \cdot \boldsymbol{y}}{\boldsymbol{u}_1 \cdot \boldsymbol{u}_1}\boldsymbol{u}_1 + \frac{\boldsymbol{u}_2 \cdot \boldsymbol{y}}{\boldsymbol{u}_2 \cdot \boldsymbol{u}_2}\boldsymbol{u}_2 = \frac{28}{14}\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \frac{126}{42}\begin{bmatrix} -5 \\ 4 \\ -1 \end{bmatrix} = \begin{bmatrix} -13 \\ 16 \\ 3 \end{bmatrix}.$$

Thus,

$$\boldsymbol{w}_2 = \boldsymbol{y} - \widehat{\boldsymbol{y}} = \begin{bmatrix} -9 \\ 20 \\ -1 \end{bmatrix} - \begin{bmatrix} -13 \\ 16 \\ 3 \end{bmatrix} = \begin{bmatrix} 4 \\ 4 \\ -4 \end{bmatrix}.$$

Note that $\boldsymbol{w}_2 \cdot \boldsymbol{u}_1 = \boldsymbol{w}_2 \cdot \boldsymbol{u}_2 = 0$, that is, $\boldsymbol{w}_2 \in W^{\perp}$. Therefore,

$$\boldsymbol{y} = \boldsymbol{w}_1 + \boldsymbol{w}_2 = \begin{bmatrix} -13 \\ 16 \\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ 4 \\ -4 \end{bmatrix} = \begin{bmatrix} -9 \\ 20 \\ -1 \end{bmatrix}.$$

C.3 Best Approximation

Theorem C.7 (The Best Approximation Theorem)**.** *Let $W \leq \mathbb{R}^n$, let $\boldsymbol{y} \in \mathbb{R}^n$, and let $\widehat{\boldsymbol{y}} = \text{proj}_W \boldsymbol{y}$. Then $\widehat{\boldsymbol{y}}$ is the closest point in W to $\boldsymbol{y}$, that is,*

$$\|\boldsymbol{y} - \widehat{\boldsymbol{y}}\| \leq \|\boldsymbol{y} - \boldsymbol{w}\|$$

for all $\boldsymbol{w} \in W$.

The vector $\widehat{\boldsymbol{y}}$ is called the **best approximation** to $\boldsymbol{y}$ in W.

Example C.8. The distance from a point $\boldsymbol{y} \in \mathbb{R}^n$ to a subspace W is defined as the distance from $\boldsymbol{y}$ to the nearest point in W, the best approximation. Continuing with Example C.6, the vector $\widehat{\boldsymbol{y}} = \begin{bmatrix} -13 \\ 16 \\ 3 \end{bmatrix}$ is the closest vector in W to the vector $\boldsymbol{y}$. Hence, $\text{dist}(W, \boldsymbol{y}) = \|\boldsymbol{y} - \widehat{\boldsymbol{y}}\| = 4\sqrt{1^2 + 1^2 + (-1)^2} = \boxed{4\sqrt{3}}$.

Let H be a flat in $\mathbb{R}^n$ passing through the vector $\boldsymbol{h}$. Can we compute the distance between H and some vector $\boldsymbol{y}$? Modifying the Best Approximation Theorem, we can because we can translate H by the vector $-\boldsymbol{h}$ so that it becomes a subspace, say W. If we also translate $\boldsymbol{y}$ by $-\boldsymbol{h}$, then we get

$$\operatorname{dist}(H, \boldsymbol{y}) = \operatorname{dist}(W, \boldsymbol{y} - \boldsymbol{h}) = \|(\boldsymbol{y} - \boldsymbol{h}) - \widehat{(\boldsymbol{y} - \boldsymbol{h})}\| = \|(\boldsymbol{y} - \widehat{\boldsymbol{y}}) - (\boldsymbol{h} - \widehat{\boldsymbol{h}})\|,$$

since vector translation is an isometry and orthogonal projections are linear transformations.

It can be shown that $\operatorname{proj}_W : \mathbb{R}^n \to \mathbb{R}^n$ for some subspace $W \leq \mathbb{R}^n$ is a linear transformation. Therefore, there must be some matrix that represent proj_W. Let $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r\}$ denote an orthonormal basis for W. Let $Q = \begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \ldots & \boldsymbol{u}_r \end{bmatrix}$ be the $n \times r$ matrix whose columns vectors are this orthonormal basis. Then $\operatorname{proj}_W(\boldsymbol{y}) = QQ^\top(\boldsymbol{y})$, that is, $QQ^\top$ is the standard matrix representation for proj_W. Note that $QQ^\top$ can be thought of as a generalization of the outer product of vectors.

Python Projects

1. In this project, you will write Python functions to implement the formula for vector angles and verify its accuracy, such as explained in Section C.1. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/angle.py

   (a) Write a `lambda` function to implement the angle formula

   $$\angle(\boldsymbol{u},\boldsymbol{v}) = \cos^{-1}\left(\frac{\boldsymbol{u}\cdot\boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|}\right).$$

   It will take in two vectors and output the angle between them in radians. Use this function to verify the result in Example C.5.

   (b) The radius of Earth is approximately $r = 6367.5$ km. Coordinates on Earth are measured by two angles, latitude α, and longitude β. (Note that the equator is latitude of $0°$ and the prime meridian is longitude of $0°$.) Using spherical coordinates, latitude and longitude can be converted into a 3-vector by the formula

   $$\begin{bmatrix} r\cos\alpha\sin\beta \\ r\cos\alpha\cos\beta \\ r\sin\alpha \end{bmatrix}.$$

   (For example, the North Pole, which is latitude $90°$ and longitude $0°$, is $r(0,0,1)$, and the intersection of the equator with the prime meridian is $r(0,1,0)$.) The following `lambda` function converts latitude and longitude into a 3-vector using this formula:

```
convert_coord = lambda coords, radius : radius*np.array([np.cos(coords[0]*np.pi
    /180)*np.sin(coords[1]*np.pi/180), np.cos(coords[0]*np.pi/180)*np.cos(coords
    [1]*np.pi/180), np.sin(coords[0]*np.pi/180)])
```

   The input `coords` is a 2-vector with latitude first, longitude second. The input `radius` is the radius of the sphere. For example, if the cities of Beijing and Palo Alto have coordinates $(39.914°, 116.392°)$ and $(37.429°, -122.138°)$, respectively, then their 3-dimensional coordinates are

   $$\boldsymbol{u} = (4374.89306899, -2170.95445797, 4085.6240192)$$

   and

   $$\boldsymbol{v} = (-4281.66943679, -2689.8449669, 3870.02546801),$$

   respectively. Therefore, spherical distance between them is

   $$6367.5\cos^{-1}\left(\frac{\boldsymbol{u}\cdot\boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|}\right) \text{ km}.$$

   This is in contrast to the linear distance between them of $\|\boldsymbol{u}-\boldsymbol{v}\| = 8674.8$ km.
   Find the spherical distance between Cedar City, Utah and Pusan, South Korea, whose coordinates are given as

   $$(37.6775°, -113.0619°) \quad \text{and} \quad (35.2100°, 129.0689°),$$

   respectively.

2. Consider the affine set containing the particular vector $\boldsymbol{x}_0 = (8,2,0)$ and for which

   $$\{(1,-1,0),(2,0,-3),(0,2,1)\}$$

   is a set of normal vectors, such as explained in Section 2.9. Then this affine set is the solution set of the linear system associated to the augmented matrix

   $$\left[\begin{array}{ccc|c} 1 & -1 & 0 & \boldsymbol{n}_1\cdot\boldsymbol{x}_0 \\ 2 & 0 & -3 & \boldsymbol{n}_2\cdot\boldsymbol{x}_0 \\ 0 & 2 & 1 & \boldsymbol{n}_3\cdot\boldsymbol{x}_0 \end{array}\right] = \left[\begin{array}{ccc|c} 1 & -1 & 0 & 6 \\ 2 & 0 & -3 & 16 \\ 0 & 2 & 1 & 4 \end{array}\right].$$

Write a Python function to create a linear system, as an augmented matrix, given a list of normal vectors $\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m$ and a particular solution $\boldsymbol{x}_0$. It should take as input an array of vectors, for the normal vectors, and a single vector, for the particular solution, and should output an augmented matrix as two parts, the coefficient matrix and the augmented column. Test this function on the above example. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/normal_vectors.py

3. Write a Python function to determine whether a set of vectors is orthogonal or not, such as explained in Section 2.9.2. It should take as input an array of vectors and should output a boolean, namely, `True` or `False`. Test this function on Example 2.9.6, as well as on the linked data set below. Your code must be able to determine whether the set contains $\mathbf{0}$ or not. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/normal_vectors.py

4. In this project, you will write a function to implement the Gram-Schmidt algorithm, such as explained in Section 3.6.1. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/GramSchmidt.py

   (a) Write a function to implement the Gram-Schmit algorithm, that is, a function which inputs a set of vectors and outputs an orthonormal set of vectors with the same span. Verify this function on Example 3.6.2.

   (b) Consider a set of vectors $A = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$. Recall that if this set is orthogonal then it is necessarily linearly independent too. Since the Gram-Schmidt algorithms orthogonalizes a set, say it turns A into $Q = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_r\}$, Q is necessarily a linearly independent set such that $\text{Span}(A) = \text{Span}(Q)$. What if A was not already independent? For vectors where a dependency relation is obtained, that is, suppose $\boldsymbol{x}_k \in \text{Span}\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{k-1}\}$, the Gram-Schmidt algorithm will turn them into the zero vector, that is,

$$\boldsymbol{v}_k = \boldsymbol{x}_k - \frac{\boldsymbol{v}_1 \cdot \boldsymbol{x}_k}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1}\boldsymbol{v}_1 - \ldots - \frac{\boldsymbol{v}_{k-1} \cdot \boldsymbol{x}_k}{\boldsymbol{v}_{k-1} \cdot \boldsymbol{v}_{k-1}}\boldsymbol{v}_{k-1} = \mathbf{0}.$$

   Such vectors are omitted from Q. Therefore, A is linearly independent if and only if $p = k$. Modify the Gram Schmidt process into a function to determine whether a set of vector is linearly independent or not. How does this compare to the algorithm in Python Project 2.**??**?

Appendix D

Error-Detecting and Correcting Codes

Digital communications are often transmitted as a sequence of binary numbers (1's and 0's), called a *bit*. Unfortunately, during transmission sometimes "noise" can interfere with the original message and alter the bits in the message. As the significance of each bit is important, a signal error in a bit can make the transmitted message useless or dangerous. As such, it will be important to be able to detect and, if possible, correct erroneous transmission.

In this chapter all our arithmetic will be modulo 2, which means that if n is an integer, then $n \equiv 0 \pmod 2$ if and only if $n = 2k$ is even and $n \equiv 1 \pmod 2$ if and only if $n = 2k+1$ is odd. When working with modular arithmetic, we need only consider the two boolean quantities 0 and 1. Let $\mathbb{Z}_2 = \{0, 1\}$. We note that

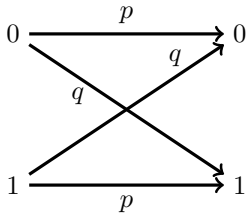
$+$	0	1
0	0	1
1	1	0

$\times$	0	1
0	0	0
1	0	1

With regard to regular arithmetic we have the important difference that $1 + 1 \equiv 0 \pmod 2$.

D.1 Binary Symmetric Channels

A **binary symmetric channel** T is a model that consists of a transmitter capable of sending a binary signal together with a receiver. Let p be the probability that a bit is transmitted correctly. Let $q = 1 - p$ then be the probability that a bit is transmitted incorrectly. This model is an example of a **Bernoulli trial** (or **binomial**[i] **trial**) from Probability Theory, which which is a random, independent experiment with two possible outcomes: "success" and "failure."



Let X be the random variable which counts the number of failures. Then by standard probability reasoning, the probability that k errors will occur if n bits are transmitted through T is

$$\mathcal{P}(X = k) = \binom{n}{k} p^{n-k} q^k.$$

[i]Recall that the **binomial coefficient** is given as $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Example D.1. Suppose that $p = 0.995$ and a 500-bit message is sent. The probability that the message is error-free ($k = 0$) is

$$\mathcal{P}(X = 0) = \binom{500}{0}(0.995)^{500}(0.005)^0 \approx 0.08157186$$

Thus, the probability that at least one error will be in the transmission is then $\mathcal{P}(X > 0) = 1 - \mathcal{P}(X = 0) \approx 0.91842814$. Continuing, the probability that exactly one error will occur is

$$\mathcal{P}(X = 1) = \binom{500}{1}(0.995)^{499}(0.005)^1 \approx 0.20495443$$

The probability that exactly two errors will occur is

$$\mathcal{P}(X = 2) = \binom{500}{2}(0.995)^{498}(0.005)^2 \approx 0.25696547$$

In particular, the probability of the message having one or two errors is $\mathcal{P}(1 \leq X \leq 2) = \mathcal{P}(X = 1) + \mathcal{P}(X = 2) \approx 0.46191990$. Furthermore, the probability of having more than two errors in the transmission is $\mathcal{P}(X > 2) = 1 - \mathcal{P}(X = 0) - \mathcal{P}(X = 1) - \mathcal{P}(X = 2) \approx 0.45650824$. This hopefully illustrates the fact that even with a low probability of failure, we must expect frequent errors when transmitting binary messages.

While you do not need to be able to calculate the probability yourself for this section, it is important to understand that errors can be surprisingly common based on the probability results here, even when the chance of error is very low. With enough transmissions, some errors will eventually arise and we need a strategy to address them.

A sequence of m-bits (read left to right) $b_1 b_2 \dots b_m$ can be identified naturally with the k-tuple $(b_1, b_2, \dots, b_m) \in \mathbb{Z}_2^m$.

Definition D.2. An **(n, m)-block code** C is a subset of $\mathbb{Z}_2^n$ paired with two functions $E : \mathbb{Z}_2^m \to \mathbb{Z}_2^n$ and $D : \mathbb{Z}_2^n \to \mathbb{Z}_2^m$, called the **encoding function** and **decoding function** respectively, such that $D \circ E = Id_{\mathbb{Z}_2^m}$ and $E(\mathbb{Z}_2^m) = C$. Necessarily, E must be injective. We call the elements of C **codewords**.

Below illustrates the typical model of communication systems.

$$\boxed{\text{Message } w \in \mathbb{Z}_2^m} \xmapsto{E} \boxed{\text{Associated code } E(w) = c \in \mathbb{Z}_2^n} \xmapsto{T} \boxed{\text{Transmitted code } T(c) \in \mathbb{Z}_2^n} \xmapsto{D} \boxed{\text{Decoded message } D(y) \in \mathbb{Z}_2^m}$$

Ideally, we want $D \circ T \circ E = Id_{\mathbb{Z}_2^m}$, but it will not be possible since T is not a function as $T(c)$ is determined by chance. A **k-error detecting code** is a code which reports an error message when $D \circ T \circ E \neq Id_{\mathbb{Z}_2^m}$ for k or fewer errors. An **ℓ-error correcting code** is a code for which $D \circ T \circ E = Id_{\mathbb{Z}_2^m}$ even in the presence of ℓ or fewer errors.

Example D.3. A simple method to detect an error in a codeword transmitted through the binary symmetric channel is to attach a **check bit** as the last bit in the message, that is, if $\boldsymbol{w} = b_1 b_2 \dots b_m \in \mathbb{Z}_2^m$ is the original codeword, then $\boldsymbol{x} = E(\boldsymbol{w}) = b_1 b_2 \dots b_m c \in \mathbb{Z}_2^{m+1}$ is the encoded message where $c = b_1 + b_2 + \dots + b_m \pmod 2$. This is called the **parity-check code**.

Imagine there is an error in transmission, say b_i is replaced with $\neg b_i$ (its negation), then the receiver has the word $T(\boldsymbol{x}) = b_1 b_2 \dots b_{i-1}(\neg b_i) b_{i+1} \dots b_m c$. Note that $b_1 + b_2 + \dots b_{i-1} + \neg b_i + b_{i+1} + \dots + b_m = \neg c \not\equiv c \pmod 2$, that is, the check bit will not match. This means that an error occurred in transmission, that is, the code detected a single error. If no error is found, then the check bit can be discarded and the original message is recovered. Therefore, this code can detect if a single error occurs from the channel.

For example, if we want to transmit the codeword $\boldsymbol{w} = 1101 \in \mathbb{Z}_2^4$, then $E(\boldsymbol{w}) = 1101 - 1 = \boldsymbol{x}$, since $1 + 1 + 0 + 1 = 3 \equiv 1 \pmod 2$. If one of the message bits alters, say $T(1101 - 1) = 1111 - 1$, then the check bit 1 will not match the check $1 + 1 + 1 + 1 \equiv 0 \not\equiv 1 \pmod 2$. Hence, an error is detected. Similarly, if the check bit itself alters during transmission, say $T(1101 - 1) = 1101 - 0$, we will also detect an transmission error. Therefore, the code can detect the change in a single bit.

Unfortunately, this **parity checking code** cannot detect the presence of two errors. For example, if $T(0000-0000) = 0000-0011$, then the code would accept the transmitted codeword as correct. This parity checking code and its variation are probably the most used type of code.

Example D.4. Consider $\mathbb{Z}_2^m$ for encoding. Then we can construct the **triple repetition code** C with $E : \mathbb{Z}_2^m \to \mathbb{Z}_2^{3m}$ given by $E(b_1, \ldots, b_m) = (b_1, \ldots, b_m, b_1, \ldots, b_m, b_1, \ldots, b_m)$ and $D : \mathbb{Z}_2^{3m} \to \mathbb{Z}_2^m$ given by $D(c_1, \ldots, c_{3m}) = (c_1, \ldots, c_m)$. For example,

$$E(0110) = 0110 - 0110 - 0110.$$

is the encoding of the original message 0110. If $T(\boldsymbol{x}) = 0110 - 0110 - 0110$ was received on the other side of the channel correctly, then $D(0110 - 0110 - 0110) = 0110$, the original four bits.

This $(3m, m)$-block code is, in fact, error correcting, because every bit is essentially sent three times and if there is an error with a bit transmission the code can correct this by outputting the most common value of the three transmission. For example, if $T(\boldsymbol{x}) = 0110 - 0010 - 0110$ is received, an inspection of the first bit of each block shows that all three are 0, which indicates this bit was likely transmitted correctly. Conversely, an inspection of the second bit of each block gives $1-0-1$. Since two of the bits are 1 and the other is 0, the most likely outcome is that this bit in the second block was transmitted incorrectly. It should instead be a 1. A similar inspection of the remaining two bits in each block indicates that all other bits were transmitted correctly. Therefore, the original codeword from the sender was $\boldsymbol{x} = 0110 - 0110 - 0110$, the same codeword as above. Hence, $D(\boldsymbol{x}) = 0110$ was the intended message to be sent.

This error-correcting code still has limits. First, it requires sending three times as much data than the original message. Also, the code cannot detect nor correct if two or more errors occur for the "same" bit. For example, if $T(\boldsymbol{x}) = 0110 - 0010 - 0010$ is received, an inspection of the second bit of each block gives $1 - 0 - 0$. We would "correct" this bit as 0, that is, we would incorrectly believe $0010 - 0010 - 0010$ is the intended codeword instead of $\boldsymbol{x} = 0110 - 0110 - 0110$.

A binary sequence can naturally be viewed as a column vector. One example of this is the parity-check from the ASCII code. Let $\boldsymbol{x} \in \mathbb{Z}_2^7$. Then the check bit at the beginning of the ASCII codeword is equal to the *inner product* (or *dot product*) of $\boldsymbol{x}$ and the all 1's vector $\mathbf{1}$:

$$\mathbf{1} \cdot \boldsymbol{x} = \mathbf{1}^T \boldsymbol{x} = 1 \cdot x_1 + 1 \cdot x_2 + \ldots + 1 \cdot x_7 = x_1 + x_2 + \ldots + x_7.$$

D.2 The Hamming Norm

Definition D.1. Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}_2^n$. The **Hamming distance** between $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\mathbb{Z}_2^n$, denoted $d(\boldsymbol{x}, \boldsymbol{y})$, is the number of bits in which $\boldsymbol{x}$ and $\boldsymbol{y}$ differ. The **Hamming norm** of $\boldsymbol{x}$, denoted $\|\boldsymbol{x}\|$, is the total number of 1's in $\boldsymbol{x}$.

The **minimum distance** of a code C, denoted $d_{\min}$ is the minimum of all Hamming distances $d(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are distinct codewords in C.

Example D.2. Let $\boldsymbol{x} = 10101$, $\boldsymbol{y} = 11010$, and $\boldsymbol{z} = 00011$ be the set of codewords for a code C. Then

$$d(\boldsymbol{x}, \boldsymbol{y}) = 4, \quad d(\boldsymbol{x}, \boldsymbol{z}) = 3, \quad d(\boldsymbol{y}, \boldsymbol{z}) = 3.$$

Thus, the minimum distance of C is 3. Also,

$$\|\boldsymbol{x}\| = 3, \quad \|\boldsymbol{y}\| = 3, \quad \|\boldsymbol{z}\| = 2.$$

Proposition D.3. *Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}_2^n$. Then $\|\boldsymbol{x} + \boldsymbol{y}\| = d(\boldsymbol{x}, \boldsymbol{y})$. In particular, $\|\boldsymbol{x}\| = d(\boldsymbol{x}, \mathbf{0})$.*

Proof. Note that $\|\boldsymbol{x} + \boldsymbol{y}\|$ counts the number of 1's in $\boldsymbol{x} + \boldsymbol{y}$, which is exactly the number of terms where $\boldsymbol{x}$ and $\boldsymbol{y}$ differ since $0 + 0 = 1 + 1 = 0$ and $1 + 0 = 0 + 1 = 1$. In particular, $\|\boldsymbol{x}\| = \|\boldsymbol{x} + \mathbf{0}\| = d(\boldsymbol{x}, \mathbf{0})$. □

Suppose that $\boldsymbol{x} = 1101$ and $\boldsymbol{y} = 1100$ are codewords in some code. If we transmit 1101 and an error occurs in the rightmost bit, then 1100 will be received. Since 1100 is a codeword, the decoder will decode 1100 as the transmitted message. This code is clearly not very appropriate for error detection. The problem is that $d(\boldsymbol{x}, \boldsymbol{y}) = 1$.

If $\boldsymbol{x} = 1100$ and $\boldsymbol{y} = 1010$ are codewords, then $d(\boldsymbol{x}, \boldsymbol{y}) = 2$. If $\boldsymbol{x}$ is transmitted and a single error occurs, then $\boldsymbol{y}$ can never be received. In particular, if $d(\boldsymbol{x}, \boldsymbol{y}) = k + 1$ and T transmits $\boldsymbol{x}$ with k errors, then the decoder D cannot confuse $T(\boldsymbol{x})$ with $\boldsymbol{y}$. This proves the following result.

Theorem D.4. *Let C be a code with minimum distance $k + 1$. Then C can error detect up to k errors.*

Given that the Hamming metric actually gives us a distance function, it makes sense to ask what the closest codeword to a binary message is. In fact, the decoder D can accomplish error correction by always mapping a binary message to its closest codeword. Of course, this would be undefined if the binary message was the midpoint between two codewords. For example, if $d(\boldsymbol{x}, \boldsymbol{y}) = 2\ell+1$ (or 2ℓ) and $T(\boldsymbol{x})$ has less than $\ell+1$ errors, then the decoder can correct $T(\boldsymbol{x})$ by mapping it back to $\boldsymbol{x}$. On the other hand, if $d(\boldsymbol{x}, \boldsymbol{y}) = 2\ell + 1$ (or 2ℓ) and $T(\boldsymbol{x})$ has greater than $\ell + 1$ errors, then the closest codeword is $\boldsymbol{y}$ instead of $\boldsymbol{x}$. We summarize this in the next theorem.

Theorem D.5. *Let C be a code with minimum distance $2\ell + 1$. Then C can error correct up to ℓ errors.*

Example D.6. The following table shows the four codewords of a code and their respective Hamming distances between each other. We can clearly see that $d_{\min} = 3$. Therefore, code has 2-error detection and 1-error correction.

	00000	00111	11100	11011
00000	0	3	3	4
00111	3	0	4	3
11100	3	4	0	3
11011	4	3	3	0

D.3 Linear Codes

Let H be an $m \times n$ binary matrix. Recall that the *null space* of H is the set of all vectors in $\mathbb{Z}_2^n$ whose product with H on the left is the zero vector, that is,

$$\text{Nul}(H) = \{\boldsymbol{y} \mid H\boldsymbol{y} = \boldsymbol{0}\}.$$

The null space of H forms a code, called the **linear code** associated to H.

One of the great advantages of linear codes is the simplicity of computing the minimum distance.

Theorem D.1. *Let C be a linear code. Then $d_{\min}$ is the minimum of all the norms of nonzero codewords in C, that is,*

$$d_{\min} = \min\{\|\boldsymbol{x}\| \mid \boldsymbol{x} \in C, \boldsymbol{x} \neq \boldsymbol{0}\}.$$

Proof. By definition, $d_{\min} = \min\{d(\boldsymbol{x}, \boldsymbol{y}) \mid \boldsymbol{x}, \boldsymbol{y} \in C, \boldsymbol{x} \neq \boldsymbol{y}\}$. By Proposition D.3, $d(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} + \boldsymbol{y}\|$. Note also that $\boldsymbol{x} \neq \boldsymbol{y}$ if and only if $\boldsymbol{x} + \boldsymbol{y} \neq 0$. Since C is a linear code, if $\boldsymbol{x}, \boldsymbol{y} \in C$ then $\boldsymbol{x} + \boldsymbol{y} \in C$. Therefore, the two sets are equal as they span the same elements. □

Example D.2. Let $H = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$. Let $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5)^\top \in \text{Nul}(H)$. Then $\boldsymbol{x}$ is a solution to the homogeneous linear system below:

$$\begin{cases} x_2 + x_4 = 0 \\ x_1 + x_2 + x_3 + x_4 = 0 \\ x_3 + x_4 + x_5 = 0 \end{cases} \sim \begin{cases} x_1 + x_4 + x_5 = 0 \\ x_2 + x_4 = 0 \\ x_3 + x_4 + x_5 = 0 \end{cases}$$

Gaussian Elimination can be used to row reduce the matrix/linear system, performing row operations in $\mathbb{Z}_2$, as displayed above. Solving the system gives the general solution

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_4 + x_5 \\ x_4 \\ x_4 + x_5 \\ x_4 \\ x_5 \end{bmatrix} = x_4 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} + x_5 \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Therefore, the null space of H is given as

$$C = \text{Span}\{11110, 10101\} = \{00000, 11110, 10101, 01011\},$$

which is a $(5, 2)$-block group code. With $d_{\min} = 3$, the code can 2-error detect and 1-error correct.

Example D.3. Let C be the linear code given by the matrix $H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$.

It is very easy for the decoder to check if a transmitted message is a codeword without computing the null space of H since H is the set of vectors $\boldsymbol{x}$ such that $H\boldsymbol{x} = \boldsymbol{0}$. Suppose that $\boldsymbol{x} = 010\ 011$ is received. Then

$$H\boldsymbol{x} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0+0+0+0+1+1 \\ 0+1+0+0+1+1 \\ 0+0+0+0+0+1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \neq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Therefore, the code has detected an error. Now the decoder must decide to request the message be sent again or correct it itself.

D.4 Parity-Check and Generator Matrices

Definition D.1. Let A be an $(n-m) \times m$ binary matrix. Then the matrix $H = (A \mid I_{n-m})$ is an $(n-m) \times n$ matrix called a **canonical parity-check matrix**. Associated to this matrix H is the $n \times m$

matrix $G = \left[\begin{array}{c} I_m \\ \hline A \end{array}\right]$, called the **standard generator matrix**.

These two matrices have a special relationship, specifically

$$HG = \left[\begin{array}{c|c} A & I_{n-m} \end{array}\right] \left[\begin{array}{c} I_m \\ \hline A \end{array}\right] = AI_m + I_{n-m}A = A + A = 0.$$

In particular, let $\boldsymbol{y} = G\boldsymbol{x}$ for some $\boldsymbol{x} \in \mathbb{Z}_2^m$. Then

$$H\boldsymbol{y} = H(G\boldsymbol{x}) = (HG)\boldsymbol{x} = 0\boldsymbol{x} = \mathbf{0}.$$

Thus, $\mathrm{Col}(G) = \mathrm{Nul}(H) = C$.

Example D.2. Suppose that we have the following eight words to encode (all of $\mathbb{Z}_2^3$):

$$000, 001, 010, 011, 100, 101, 110, 111.$$

Using $A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, we construct the canonical parity-check matrix H and its generator G:

$$H = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}, \quad G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

We can then encode each $\boldsymbol{x} \in \mathbb{Z}_2^3$ by multiplying it by G, that is, $G\boldsymbol{x}$ is its associated codeword:

$$\left\{\begin{array}{l} G(000) = 000\ 000,\ G(100) = 100\ 011,\ G(010) = 010\ 110,\ G(110) = 110\ 101, \\ G(001) = 001\ 101,\ G(101) = 101\ 110,\ G(011) = 011\ 011,\ G(111) = 111\ 000 \end{array}\right\}$$

Notice that the first three bits of each codeword is just the decoded message.

The efficacy of linear codes comes from balance between linear independence and dependence in the column vectors of H. For example, to form an (n, m)-block code, H must be a $p \times n$ matrix with $\mathrm{nullity}(H) = m$. The nullity determines the size of code C, and as a consequence the message set $\mathbb{Z}_2^m$. For parity-check matrices, the rank of H, which is $\mathrm{rank}(H) = n - m$, will be the size of the augmented identity matrix I_{n-m}. In particular, we have that $p = n - m$. But still, how does one choose the matrix A? More importantly, how large does one set n? The larger n is, the more room there is possible between codewords, thus increasing the minimum distance. This allows for larger levels of error detection and correction. Thus, a large n allows better codes, but a large n does not guarantee better detection/correction. This depends on the choice of A.

As i ranges from 1 to n, the set of the $\boldsymbol{e}_i$'s form the standard basis on $\mathbb{Z}_2^n$. Also, the product $H\boldsymbol{e}_i$ is equal to the ith column of the matrix H. Let $\boldsymbol{x} \in \mathbb{Z}_2^n$. Then there exists some index set I such that $\boldsymbol{x} = \sum_{i \in I} \boldsymbol{e}_i$. Thus, $H\boldsymbol{x} = \sum_{i \in I} H\boldsymbol{e}_i$, that is, $H\boldsymbol{x}$ is a sum of columns of H. If $H\boldsymbol{x} = \mathbf{0}$, then the corresponding sum of

columns is zero. Let $\boldsymbol{x} \in \text{Nul}(H)$ such that $\boldsymbol{x}$ has minimum nonzero weight. Of course, $w(\boldsymbol{x}) = |I|$. Then $\sum_{i \in I} H\boldsymbol{e}_i$ is the smallest nontrivial linear combination of column vectors of H which combine to zero. We call this a **minimum dependency relation** on the columns of H. Then $d_{\min} = w(\boldsymbol{x}) = |I|$, the size of a minimum dependency relation. We have then shown the following.

Theorem D.3. *Let H be a matrix with $\mathbb{Z}_2$-entries. Let d be the size of a minimum dependency relation on the columns of H. Then the linear code $\text{Nul}(H)$ can detect $d-1$ errors and correct $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors.*

Corollary D.4. *Let H be an $m \times n$ binary matrix. Then the null space of H is a single error-detecting code if and only if no column of H consists entirely of zeros. Furthermore, the null space of H is a single error-correcting code if and only if H does not contain any zero columns and no two columns of H are identical.*

Proof. Let d denote the size of a minimum dependency relation. For the first equivalence, note that both statements imply that $d > 1$. For the second equivalence, note that both statements imply that $d > 2$. □

Example D.5. The linear code associated with matrix H_1, below, can detect 1 error since it has no column of zeros but cannot correct the error since the first two columns are identical. The linear code associated with matrix H_2, also below, cannot detect an error since it has a column of zeros. Finally, the linear code associated with matrix H_3 can both correct and detect a single error.

$$H_1 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}, \quad H_2 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}, \quad H_3 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

D.5 Decoding

Although a code CAN correct an error, how can this be done efficiently? Consider the linear code C associated with the binary matrix H. If $\boldsymbol{x} \in \mathbb{Z}_2^n$ was received and has no errors, then no correction is necessary. As mentioned before, we can detect errors by computing $H\boldsymbol{x}$, called the **syndrome** of $\boldsymbol{x}$. If the syndrome is zero, then $\boldsymbol{x}$ has no errors. If H is a parity-check matrix, then decoding just means finding the information bits and discarding the parity bits.

In the presence of errors, let $\boldsymbol{\mu}, \boldsymbol{\varepsilon} \in \mathbb{Z}_2^n$ such that $\boldsymbol{\mu}$ was the original codeword, that is, $T(\boldsymbol{\mu}) = \boldsymbol{x}$, and $\boldsymbol{\varepsilon}$ is the error added to $\boldsymbol{\mu}$ from noise, that is, $\boldsymbol{x} = \boldsymbol{\mu} + \boldsymbol{\varepsilon}$. Then

$$H\boldsymbol{x} = H(\boldsymbol{\mu} + \boldsymbol{\varepsilon}) = H\boldsymbol{\mu} + H\boldsymbol{\varepsilon} = H\boldsymbol{\varepsilon},$$

where the last equality follows since $\boldsymbol{\mu}$ is a codeword, that is, $\boldsymbol{\mu} \in \text{Nul}(H)$. Therefore, $\boldsymbol{x}$ has the same syndrome as its error. If $\boldsymbol{\varepsilon} = \boldsymbol{e}_i$, then $H\boldsymbol{x} = H\boldsymbol{\varepsilon} = H\boldsymbol{e}_i$, which the the ith column of H. Therefore, the error of $\boldsymbol{x}$ can be found in the ith bit. Therefore, using syndromes, we can both detect and correct a single error.

For multiple errors, the detection process remains the same, use the syndrome. For error correction, we will solve the linear system associated with the augmented matrix $(H \mid H\boldsymbol{x})$. This will express $H\boldsymbol{x}$ as a linear combination of the columns of H. Since the columns are linearly dependent, there will be multiple solutions. We assume, of course, that the numbers of errors is few and choose the vector in this solution set of minimum weight. If there is a tie, then we cannot correct the error and request the original message be re-transmitted.

Example D.1. Consider the linear code associated with $H = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$. Suppose that the codes $\boldsymbol{x}$ = 001-111, $\boldsymbol{y}$ = 111-110, $\boldsymbol{z}$ = 010-111, and $\boldsymbol{w}$ = 111-111 are received. We compute their syndromes below:

$$H\boldsymbol{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad H\boldsymbol{y} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad H\boldsymbol{z} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad H\boldsymbol{w} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

Considering the syndromes, the first message was sent without errors and is $\boxed{\textbf{001}\text{-}111}$. For the second message, the syndrome is the third column of H and thus $\boldsymbol{y}$ has an error in the third bit. Thus, the original message was $\boxed{\textbf{110}\text{-}110}$. For the third message, the syndrome is the fourth column of H. Thus, the error is in the fourth bit and the sent message was $\boxed{\textbf{010}\text{-}011}$.

The last one is more difficult. The syndrome does not correspond to a column of H and hence represents more than one error in transmission. It could be the sum of the first two columns, the sum of the third and sixth columns, or the sum of the fourth and fifth columns, which all have the same length of 2! This is because $d_{\min} = 3$ and C can only correct a single error not two. But C can detect two errors, so the receiver should request the message be sent again.

Python Projects

1. In this project, you will write Python functions to implement a parity-check error detection code, such as explained in Example D.3. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/parity_check.py

   (a) Write a Python function `encode_paritycheck(message)` which takes one input parameter: `message` $\boldsymbol{w} \in \mathbb{Z}_2^m$. It will append the check-bit at the end of the array and output the encoded message $\boldsymbol{x} \in \mathbb{Z}_2^{m+1}$. Verify this function using Example D.3.

   (b) Write a Python function `detect_paritycheck(codeword)` which takes one input parameter: `codeword` $\boldsymbol{x} \in \mathbb{Z}_2^{m+1}$, a codeword encoded by the previous function and sent through the channel. The function will check the last bit and return `True` if an error is detected. Otherwise, return `False`.

   (c) Write a `lambda` function `decode_paritycheck(codeword)` which takes one input parameter: `codeword` $\boldsymbol{x} \in \mathbb{Z}_2^{m+1}$, a codeword encoded by the previous function and sent through the channel. It will remove the check-bit at the end of the array and outputs the decoded message $\boldsymbol{w} \in \mathbb{Z}_2^m$.

2. In this project, you will write Python functions to implement triple-repetition error correction, such as explained in Example D.4. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/triple_rep.py

   (a) Write a Python function `encode_triplerep(message)` which takes one input parameter: `message` $\boldsymbol{w} \in \mathbb{Z}_2^m$. It will concatenate three copies of message together and output the encoded message $\boldsymbol{x} \in \mathbb{Z}_2^{3m}$.

   (b) Write a Python function `decode_triplerep(codeword)` which takes one input parameter: `codeword`, a codeword $\boldsymbol{x} \in \mathbb{Z}_2^{3m}$ encoded by the previous function and sent through the channel. It will correct any errors and remove the repetition, restoring the original `message` $\boldsymbol{w} \in \mathbb{Z}_2^m$, which will be the output of the function.

3. In this project, you will write functions to implement compute Hamming distance, Hamming norms, and to decode message associated to a specific (5,2)-block code, such as explained in Section D.2. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/Hamming.py

   (a) Write two Python functions to compute the Hamming distance between two Boolean vectors and to compute the Hamming norm of a Boolean vector. Verify it on Example D.2.

   (b) Using the code in Example D.6, write a Python function to detect/correct transmitted Boolean 5-vectors, that is, write $D : \mathbb{Z}_2^5 \to C$ the decoding function associated to this code using the Hamming distance. Verify the function on the messages 01011, 11100, 10001, and 11000.

4. Write a Python function to detect (not correct) errors associated to the code $C = \text{Nul}(A)$, where A is the 3×6 matrix in Example D.3, such as explained in Section D.3. Verify the function on the messages 111000, 110110, and 111101. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/linear_detection.py

5. In this project, you will write Python functions to create canonical parity check and generator matrices and to encode linear codes, such as explained in Section D.4. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/linear_encoding.py

   (a) Write a function to create the canonical-parity check matrix H and its associated generator matrix G given a matrix A. Verify your function on the example above. Example D.2.

   (b) Write a function to encode messages from $\mathbb{Z}_2^m$ using the generator matrix of a linear code. Verify it on the linear code associated to H_3 from Example D.5 and the four elements of

$$\mathbb{Z}_2^2 = \{00, 01, 10, 11\}.$$

6. Let C be the $(10, 5)$-linear code associated to the matrix

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Write a Python program to decode transmitted messages using the canonical parity check matrix of a linear code, such as explained in Section D.5. The following five messages were received through the binary channel:

10010-11100, 01101-00100, 11000-00011, 11001-01010, 11101-11000

using the $(10, 5)$-linear code above. Decode these messages, detect any errors, and correct any errors if possible. Python code is started for you at:
https://github.com/emisseldine/MisselBooks/tree/main/LinearAlgebra/projects/linear_detection.py
You will need `linear_code(A)` from Exercise D.4 to verify your code here.

Appendix E

Solutions to Select Exercises

Chapter 1 : The Algebra and Geometry of Vectors

1.1 Vectors as Data

1. 75 2. 420 3. 16

4. A patient with few positive metrics, like 0 age, 0 weight, 0 height. Bizarre! Maybe an embryo?

♠ 5. A bill that requires only a few distinct materials.

♠ 6. Data where nearly all the observations occur in a small number of categories.

♠ 7. A business with very few customers and vendors.

♠ 8. A profile where nearly all the features are absent.

♠ 9. A location which is nearly always 0 F°. Freezing!

♠ 10. An audio signal that is nearly all silence.

11. A photo that is nearly all black/blank.

1.2 Vector Operations

1. $\begin{bmatrix} 3 \\ -9 \end{bmatrix}$ 2. $\begin{bmatrix} 1 \\ -8 \end{bmatrix}$ 3. $\begin{bmatrix} 9 \\ 20 \\ 27 \end{bmatrix}$ ♠ 4. $\begin{bmatrix} -9 \\ -8 \\ 5 \end{bmatrix}$ 5. $\begin{bmatrix} 8 \\ 10 \\ 16 \end{bmatrix}$ 6. $\begin{bmatrix} 4 \\ 14 \\ 6 \end{bmatrix}$

7. $\begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}$ 8. $\begin{bmatrix} 14 \\ 32 \\ 28 \end{bmatrix}$ 9. $\begin{bmatrix} \frac{9\pi+16}{3} \\ -7\pi+4 \\ \frac{9\pi+140}{15} \end{bmatrix}$ 10. $\begin{bmatrix} 23 \\ 19 \\ 19 \end{bmatrix}$

1.3 Inner Products

1. 5 ♠ 2. 48 3. 18 ♠ 4. -4 5. -40 ♠ 6. 7 ♠ 7. 36

♠ 8. 22 9. 342

10. 748 ft-lbs
11. 802 ft-lbs
12. 0 ft-lbs
13. 1311 ft-lb
14. 480 ft-lb
15. 2918.311 ft-lbs
16. 3467.466 ft-lbs
17. 5196.152 ft-lbs
18. 9243.910 ft-lbs

1.4 Norms

1. $\sqrt{35}$ ♠ 2. $5\sqrt{3}$ 3. $\sqrt{14}$ ♠ 4. $3\sqrt{2}$ 5. $\sqrt{298}$ 6. $\sqrt{17}$

7. $\begin{bmatrix} \frac{3}{5} \\ \frac{4}{5} \end{bmatrix}$ 8. $\begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \\ \frac{2}{3} \end{bmatrix}$ 9. $\begin{bmatrix} \frac{2}{\sqrt{14}} \\ \frac{3}{\sqrt{14}} \\ \frac{1}{\sqrt{14}} \end{bmatrix}$

10. $\begin{bmatrix} \frac{4}{5} \\ 0 \\ \frac{3}{5} \end{bmatrix}$ 11. $\begin{bmatrix} \frac{1}{\sqrt{21}} \\ \frac{4}{\sqrt{21}} \\ -\frac{2}{\sqrt{21}} \end{bmatrix}$ 12. $\begin{bmatrix} \frac{3}{2\sqrt{65}} \\ \frac{7}{2\sqrt{65}} \\ \frac{12}{2\sqrt{65}} \\ \frac{8}{2\sqrt{65}} \end{bmatrix}$

13. $\sqrt{83}$ ♠ 14. $3\sqrt{2}$ ♠ 15. $3\sqrt{5}$ 16. $\sqrt{14}$

17. $\frac{4}{3}$, $\sqrt{\frac{14}{3}}$, $\frac{1}{3}\sqrt{26}$ 18. $\frac{5}{4}$, $\frac{1}{2}\sqrt{33}$, $\frac{1}{4}\sqrt{107}$

1.5 Clustering

♠ 1. The mean of a collection of nonnegative numbers is itself nonnegative (as sums and positive products of nonnegative numbers are nonnegative). As each $\boldsymbol{z}_j$ is a mean of vectors, each of its entries is the mean of the corresponding entries in the vectors $\boldsymbol{x}_i$. Therefore, each $\boldsymbol{z}_j$ has only nonnegative entries.

♠ 2. Each entry of $\boldsymbol{z}_j$ will be an average number of occurences of that category for its group.

♠ 3. Relative frequency vectors are nonnegative, and by Exercise **??** we see that each $\boldsymbol{z}_j$ is likewise nonnegative. The sum of $\boldsymbol{z}_j$ will be the sum of averages of each entry from each vector $\boldsymbol{x}_i$ in its group. Using properties of summation, this is equivalent to the average of the vector sums. As each vector sums to 1, their average is likewise 1. Thus, the sum of $\boldsymbol{z}_j$ is 1. Therefore, $\boldsymbol{z}_j$ is a relative frequency vector.

♠ 4. Each entry of $\boldsymbol{z}_j$ is will be average presence of the feature measured by the Boolean entry.

♠ 5. Suppose we have a collection of patient profile where it is known that some patients have a certain disorder, it is known that some patients do not have the disorder, and it is not known whether the remaining patients have the disorder or not. Suppose we cluster them into two groups. We require all the patients known to not have the disorder into group 0 and all patients known to have the disorder into group 1. All the remaining patients are clustered according to the prescribed clustering method. If the collection of known patients is large enough, then the clustering of the unknown patients could be a test to determine the presence of the disorder or not.

♠ 6. $f(x) = \|\boldsymbol{x}-\boldsymbol{z}_2\|^2-\|\boldsymbol{x}-\boldsymbol{z}_1\|^2 = (\boldsymbol{x}-\boldsymbol{z}_2)\cdot(\boldsymbol{x}-\boldsymbol{z}_2)-(\boldsymbol{x}-\boldsymbol{z}_1)\cdot(\boldsymbol{x}-\boldsymbol{z}_1) = (\boldsymbol{x}\cdot\boldsymbol{x}-\boldsymbol{x}\cdot\boldsymbol{z}_2-\boldsymbol{z}_2\cdot\boldsymbol{x}+\boldsymbol{z}_2\cdot\boldsymbol{z}_2)-(\boldsymbol{x}\cdot\boldsymbol{x}-\boldsymbol{x}\cdot\boldsymbol{z}_1-\boldsymbol{z}_1\cdot\boldsymbol{x}+\boldsymbol{z}_1\cdot\boldsymbol{z}_1) = (\boldsymbol{x}\cdot\boldsymbol{x}-\boldsymbol{x}\cdot\boldsymbol{x})-(2\boldsymbol{z}_2)\cdot\boldsymbol{x}+(2\boldsymbol{z}_1)\cdot\boldsymbol{x}+(\boldsymbol{z}_2\cdot\boldsymbol{z}_2-\boldsymbol{z}_1\cdot\boldsymbol{z}_1) = 2(\boldsymbol{z}_1-\boldsymbol{z}_2)\cdot\boldsymbol{x}+(\boldsymbol{z}_2\cdot\boldsymbol{z}_2-\boldsymbol{z}_1\cdot\boldsymbol{z}_1)$. Therefore, let $\boldsymbol{a} = 2(\boldsymbol{z}_1 - \boldsymbol{z}_2)$ and $b = \boldsymbol{z}_2 \cdot \boldsymbol{z}_2 - \boldsymbol{z}_1 \cdot \boldsymbol{z}_1 = \|\boldsymbol{z}_2\|^2 - \|\boldsymbol{z}_1\|^2$.

1.6 The k-means Algorithm

1. $\boldsymbol{c} = [1,3,3,2,1,2,3,1,1,2]$

$$Z = \left\{ \begin{bmatrix} 7/2 \\ -3/2 \\ 1/2 \end{bmatrix}, \begin{bmatrix} 0 \\ 3 \\ -2/3 \end{bmatrix}, \begin{bmatrix} 1/3 \\ 2/3 \\ 7/3 \end{bmatrix} \right\}$$

♠ 2. $\boldsymbol{c} = [1,3,3,3,1,2,3,1,1,2]$

$$Z = \left\{ \begin{bmatrix} 7/2 \\ -3/2 \\ 1/2 \end{bmatrix}, \begin{bmatrix} 1 \\ 5/2 \\ -5/2 \end{bmatrix}, \begin{bmatrix} -1/4 \\ 3/2 \\ 5/2 \end{bmatrix} \right\}$$

♠ 3. $\boldsymbol{c} = [1,3,3,3,1,3,3,1,1,2]$

$$Z = \left\{ \begin{bmatrix} 7/2 \\ -3/2 \\ 1/2 \end{bmatrix}, \begin{bmatrix} 3 \\ 4 \\ -5 \end{bmatrix}, \begin{bmatrix} -2/5 \\ 7/5 \\ 2 \end{bmatrix} \right\}$$

4. $\boldsymbol{c} = [1,3,3,3,1,3,3,1,1,2]$

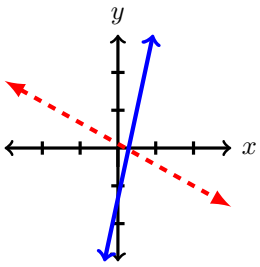
$$Z = \left\{ \begin{bmatrix} 7/2 \\ -3/2 \\ 1/2 \end{bmatrix}, \begin{bmatrix} 3 \\ 4 \\ -5 \end{bmatrix}, \begin{bmatrix} -2/5 \\ 7/5 \\ 2 \end{bmatrix} \right\}$$

All solutions determined by the k-means Algorithm depend on the initial randomization. Answers may vary on the remaining exercises.

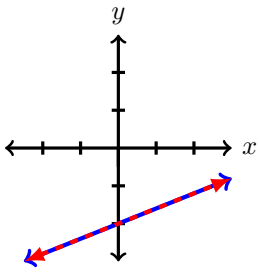
Chapter 2 : Linear Systems

2.1 Linear Systems

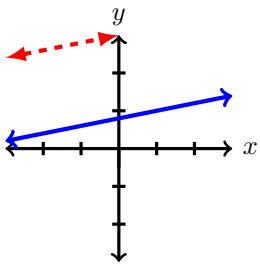
1. yes
2. yes
3. no
4. no
5. no
6. no
7. yes
8. no
♠ 9. no
♠ 10. no
♠ 11. yes
♠ 12. no
13. no
14. no
15. no
16. yes
17. no
18. no solution, inconsistent
19. infinitely many solutions, consistent
20. no solution, inconsistent
21. unique solution, consistent
22. consistent, unique solution



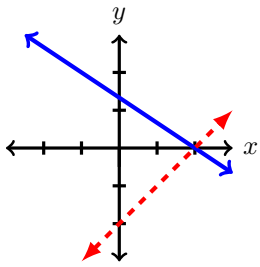
23. consistent, multiple solutions



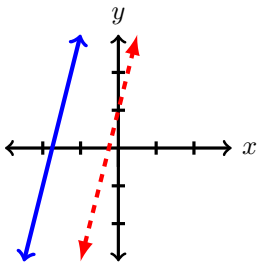
24. inconsistent



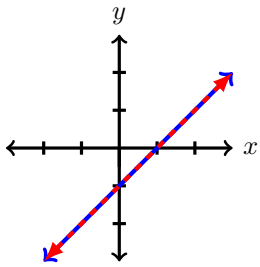
25. consistent, unique solution



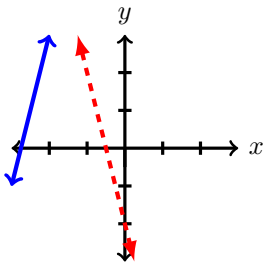
26. inconsistent



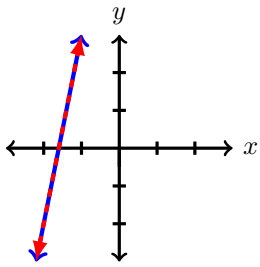
27. consistent, multiple solutions



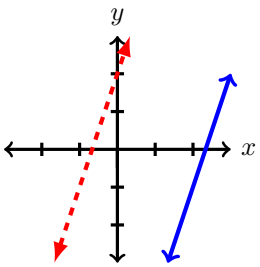
28. consistent, unique solution



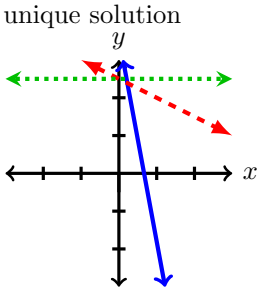
29. consistent, multiple solutions



30. inconsistent



31. consistent, unique solution



32. homogeneous, $(0,0)$

33. nonhomogeneous

♠ 34. homogeneous, $(0,0)$

♠ 35. nonhomogeneous

36. nonhomogeneous

37. nonhomogeneous

38. nonhomogeneous

39. nonhomogeneous

♠ 40. homogeneous, $(0,0,0,0)$

41. nonhomogeneous

42. homogeneous, $(0,0,0,0,0)$

43. $\begin{cases} x + y - z = 5 \\ 3x - y + z = 11 \\ -x + 2y - 4z = 0 \end{cases}$

2.2 Augmented Matrices

1. $\left[\begin{array}{cc|c} \boxed{1} & 0 & 0 \\ 0 & \boxed{1} & 0 \end{array}\right]$
free variables: none

2. $\left[\begin{array}{cccc|c} \boxed{1} & 2 & 1 & 4 & -1 \\ 0 & \boxed{1} & 3 & 5 & 0 \end{array}\right]$
free variables: x_3, x_4

3. $\left[\begin{array}{cccc|c} 0 & \boxed{2} & 1 & 3 & 0 \\ 0 & 0 & \boxed{1} & 2 & 0 \\ 0 & 0 & 0 & \boxed{4} & 0 \end{array}\right]$
free variables: x_1

4. $\left[\begin{array}{cccc|c} \boxed{1} & 4 & 2 & 6 & 1 \\ 0 & \boxed{2} & 2 & 1 & 2 \\ 0 & 0 & 0 & \boxed{2} & 3 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$
free variables: x_3

5. $\left[\begin{array}{ccccc|c} \boxed{1} & 3 & 5 & 0 & 1 & 1 \\ 0 & \boxed{3} & 4 & 1 & 2 & 2 \\ 0 & 0 & 0 & \boxed{6} & 1 & 3 \\ 0 & 0 & 0 & 0 & \boxed{2} & 4 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{array}\right]$
free variables: x_3

6. $\left[\begin{array}{ccc} \boxed{1} & 0 & 0 \\ 0 & \boxed{1} & 0 \\ 0 & 0 & \boxed{1} \end{array}\right]$
RREF, rank 3

7. $\left[\begin{array}{ccc} \boxed{1} & 2 & 0 & -3 \\ 0 & 0 & \boxed{1} & 5 \end{array}\right]$
RREF, rank 2

8. $\left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]$
RREF, rank 0

9. $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
RREF, rank 0

10. $\begin{bmatrix} \boxed{3} & -2 & 4 & 5 \end{bmatrix}$
echelon form, rank 1

11. $\begin{bmatrix} \boxed{1} & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$
RREF, rank 1

12. $\begin{bmatrix} \boxed{1} & 1 & 0 \\ 0 & \boxed{1} & 0 \\ 0 & 0 & \boxed{1} \end{bmatrix}$
echelon form, rank 3

13. $\begin{bmatrix} 0 & \boxed{1} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
RREF, rank 1

14. $\begin{bmatrix} \boxed{1} & 2 & 4 & -3 \\ 0 & 0 & \boxed{1} & 5 \end{bmatrix}$
echelon form, rank 2

15. $\begin{bmatrix} \boxed{1} & 0 & 0 & 0 \\ 0 & 0 & \boxed{2} & 0 \\ 0 & 0 & 0 & \boxed{1} \end{bmatrix}$
echelon form, rank 3

16. $\begin{bmatrix} \boxed{1} & 6 & -5 & 0 \\ 0 & 0 & 0 & \boxed{2} \\ 0 & 0 & 0 & 0 \end{bmatrix}$
echelon form, rank 2

17. $\left[\begin{array}{cc|c} \boxed{5} & 6 & 3 \\ 0 & \boxed{4} & 12 \end{array}\right]$
echelon form, rank 2

18. $\left[\begin{array}{ccc|c} \boxed{2} & 4 & 3 & 6 \\ 0 & \boxed{2} & 6 & 2 \\ 0 & 0 & \boxed{1} & 4 \end{array}\right]$
echelon form, rank 3

19. NOT echelon form

♠ 20. $\left[\begin{array}{ccccc} \boxed{1} & 2 & 3 & 4 & 5 \\ 0 & \boxed{6} & 7 & 8 & 9 \\ 0 & 0 & 0 & \boxed{3} & 4 \end{array}\right]$
echelon form, rank 3

♠ 21. $\left[\begin{array}{ccccc} \boxed{1} & 2 & 3 & 4 & 5 \\ 0 & \boxed{1} & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 & 2 \end{array}\right]$
echelon form, rank 2

♠ 22. NOT echelon form

♠ 23. $\left[\begin{array}{cccc|c} \boxed{1} & 0 & 3 & 0 & 5 \\ 0 & \boxed{1} & 7 & 0 & 9 \\ 0 & 0 & 0 & \boxed{1} & 2 \end{array}\right]$
RREF, rank 3

♠ 24. $\left[\begin{array}{cccc|c} \boxed{1} & 0 & 3 & 4 & 5 \\ 0 & \boxed{1} & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 & 2 \end{array}\right]$
RREF, rank 2

♠ 25. NOT echelon form

26. NOT echelon form

27. $\left[\begin{array}{ccc|c} \boxed{1} & -\frac{1}{5} & \frac{2}{5} & -\frac{2}{5} \\ 0 & \boxed{2} & 1 & \frac{7}{3} \\ 0 & 0 & 0 & 1 \end{array}\right]$
echelon form, rank 2

28. $\left[\begin{array}{ccc|c} \boxed{1} & 0 & \frac{1}{2} & 0 \\ 0 & \boxed{1} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]$
RREF, rank 2

29. NOT echelon form

30. $\left[\begin{array}{cccc|c} \boxed{1} & 0 & 0 & 3 & 5 \\ 0 & \boxed{7} & 0 & 0 & 4 \\ 0 & 0 & 0 & \boxed{1} & 6 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$

echelon form, rank 3

31. $\left[\begin{array}{cccc|c} \boxed{1} & 0 & 2 & 1 & 5 \\ 0 & \boxed{4} & 3 & 2 & 6 \\ 0 & 0 & 0 & \boxed{3} & 2 \end{array}\right]$

32. $\left[\begin{array}{ccc|c} 9 & 4 & 2 & 6 \\ 3 & 5 & 2 & 7 \\ 3 & 17 & 8 & 24 \end{array}\right]$

♠ 33. $\left[\begin{array}{cc|c} 3 & -1 & 0 \\ 0 & -2 & 5 \\ 1 & 7 & 13 \end{array}\right]$

♠ 34. $\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ -2 & -6 & 1 & 12 \end{array}\right]$

35. $\left[\begin{array}{cccc|c} 10 & -7 & 2 & -4 & 10 \\ 3 & 4 & -3 & 1 & 20 \\ 1 & 0 & 4 & 0 & 73 \\ 0 & 6 & -3 & 2 & 10 \end{array}\right]$

36. $\left[\begin{array}{cccc|c} 12 & -2 & 3 & \frac{3}{2} & 13 \\ 1 & 3 & -2 & 0 & 9 \\ 0 & 1 & 17 & -20 & 12 \end{array}\right]$

37. $\begin{cases} x_1 - 2x_2 - 3x_3 = -4 \\ x_2 + 2x_3 = 1 \\ -2x_1 - 2x_3 = 8 \end{cases}$

♠ 38. $\begin{cases} 3x - 2y - z = 4 \\ x + 3z = -3 \\ 0 = 2 \end{cases}$

♠ 39. $\begin{cases} 4x - y - 2z + 5w = 12 \\ -3x + w = 5 \\ x = 0 \end{cases}$

40. $\begin{cases} 12x + z - 9w = 3 \\ 9y + 21z + 25w = -9 \\ 14x - 7y + 36w = 67 \\ 3x + 4y - 7z + 19w = 2 \\ -18x + 2z + 7w = 1 \end{cases}$

41. $\begin{cases} 2x + 2y = 5 \\ 3x + 3y - 3z = 3 \\ -5y + 4z = -1 \end{cases}$

♠ 42. $\begin{cases} x - y + 3z = 6 \\ 2y + 5z = 10 \\ 2x + 7y - z = 0 \end{cases}$

♠ 43. $\begin{cases} 3x - y = 0 \\ y = -\frac{5}{2} \\ x + 7y = 13 \end{cases}$

♠ 44. $\begin{cases} x + y + z = 1 \\ -4y + 3z = 14 \end{cases}$

45. $\left[\begin{array}{ccc|c} 1 & 3 & 2 & 3 \\ 0 & 1 & 7 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right]$

46. $\left[\begin{array}{cccc|c} 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 5 \\ 4 & 4 & 6 & 8 & 17 \end{array}\right]$

47. $\left[\begin{array}{cccc|c} 1 & 2 & 3 & -1 & 2 \\ 0 & 1 & 3 & 5 & 5 \\ 0 & 0 & 0 & 1 & 1 \end{array}\right]$

48. $\left[\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 1 & 2 & 2 & 3 \\ 9 & 4 & 7 & 2 \end{array}\right]$

49. $\left[\begin{array}{ccc|c} 0 & 2 & 7 & 6 \\ 1 & 2 & 0 & \frac{1}{2} \\ 3 & 6 & 4 & 4 \end{array}\right]$

50. $\left[\begin{array}{ccc|c} 1 & 2 & 3 & 6 \\ 0 & -1 & -2 & -11 \\ 0 & 0 & 0 & 16 \end{array}\right]$

51. Interchange Rows 1 and 2.

52. Interchange Rows 1 and 3,
replace Row 1 with Row 1 − Row 2,
scale Row 1 by $\frac{1}{2}$.

53. Replace Row 2 with Row 2 − 3Row 1,
scale Row 2 by $\frac{1}{2}$

54. Replace Row 2 with Row 2 − 3Row 1,
replace Row 3 with Row 3 − 2Row 1,
replace Row 4 with Row 4 + 2Row 1

55. Replace Row 1 with Row 1 − 6Row 3,
replace Row 2 with Row 2 − 2Row 3,
replace Row 4 with Row 4 − 3Row 3

56. $(2, 4, 1)$

2.3 Reduction of Linear Systems

1. $\left[\begin{array}{cc} \boxed{1} & 2 \\ 0 & \boxed{13} \end{array}\right] \sim \left[\begin{array}{cc} \boxed{1} & 0 \\ 0 & \boxed{1} \end{array}\right]$,

replace Row 2 with Row 2 + 4Row 1
scale Row 2 by $\frac{1}{13}$,
replace Row 1 with Row 1 − 2Row 2

2. $\left[\begin{array}{cc} \boxed{3} & 1 \\ 0 & \boxed{\text{-4}} \end{array}\right] \sim \left[\begin{array}{cc} \boxed{1} & 0 \\ 0 & \boxed{1} \end{array}\right]$,

interchange Row 1 and Row 2
replace Row 2 with Row 2 − 2Row 1
scale Row 2 by $-\frac{1}{4}$,
replace Row 1 with Row 1 − Row 2

scale Row 1 by $\frac{1}{3}$

♠ 3. $\begin{bmatrix} \boxed{1} & 2 & 3 \\ 0 & \boxed{1} & 3 \\ 0 & 0 & \boxed{1} \end{bmatrix} \sim \begin{bmatrix} \boxed{1} & 0 & 0 \\ 0 & \boxed{1} & 0 \\ 0 & 0 & \boxed{1} \end{bmatrix}$,

replace Row 2 with Row 2 − 2Row 1,
replace Row 3 with Row 3 − 2Row 1,
scale Row 2 by −1, scale Row 3 by −1
replace Row 2 with Row 2 − 3Row 3,
replace Row 1 with Row 1 − 3Row 3,
replace Row 1 with Row 1 − 2Row 2

♠ 4. $\begin{bmatrix} \boxed{1} & 2 & 2 & 1 \\ 0 & \boxed{-3} & -6 & -4 \\ 0 & 0 & 0 & 0 \end{bmatrix} \sim \begin{bmatrix} \boxed{1} & 0 & -2 & -\frac{5}{3} \\ 0 & \boxed{1} & 2 & \frac{4}{3} \\ 0 & 0 & 0 & 0 \end{bmatrix}$,

replace Row 2 with Row 2 − 2Row 1,
replace Row 3 with Row 3 − Row 1,
replace Row 3 with Row 3 − Row 2,
scale Row 2 by $-\frac{1}{3}$,
replace Row 1 with Row 1 − 2Row 2

♠ 5. $\begin{bmatrix} \boxed{1} & 5 & 6 & -5 \\ 0 & \boxed{1} & 3 & 3 \\ 0 & 0 & \boxed{25} & 61 \\ 0 & 0 & 0 & \boxed{\frac{712}{25}} \end{bmatrix} \sim \begin{bmatrix} \boxed{1} & 0 & 0 & 0 \\ 0 & \boxed{1} & 0 & 0 \\ 0 & 0 & \boxed{1} & 0 \\ 0 & 0 & 0 & \boxed{1} \end{bmatrix}$,

replace Row 2 with Row 2 − 3Row 1,
replace Row 4 with Row 4 + 3Row 1,
interchange Rows 2 and 3,
replace Row 3 with Row 3 + 15Row 2,
replace Row 4 with Row 4 − 20Row 2,
replace Row 4 with Row 4 + $\frac{42}{25}$Row 3
scale Row 4 by $\frac{25}{712}$,
replace Row 1 with Row 1 + 5Row 4,
replace Row 2 with Row 2 − 3Row 4,
replace Row 3 with Row 3 − 61Row 4,
scale Row 3 by $\frac{1}{25}$,
replace Row 1 with Row 1 − 6Row 3,
replace Row 2 with Row 2 − 3Row 3,
replace Row 1 with Row 1 − 5Row 2

6. $\left[\begin{array}{ccc|c} \boxed{1} & 2 & 0 & 9 \\ 0 & \boxed{2} & 4 & 6 \\ 0 & 0 & \boxed{5} & 10 \end{array}\right] \sim \left[\begin{array}{ccc|c} \boxed{1} & 0 & 0 & 11 \\ 0 & \boxed{1} & 0 & -1 \\ 0 & 0 & \boxed{1} & 2 \end{array}\right]$,

scale Row 2 by $\frac{1}{2}$,
scale Row 3 by $\frac{1}{5}$,
replace Row 2 with Row 2 − 2Row 3
replace Row 1 with Row 1 − 2Row 2

7. $\left[\begin{array}{ccc|c} \boxed{8} & 16 & 0 & 24 \\ 0 & \boxed{9} & 18 & 9 \\ 0 & 0 & \boxed{4} & 8 \end{array}\right] \sim \left[\begin{array}{ccc|c} \boxed{1} & 0 & 0 & 9 \\ 0 & \boxed{1} & 0 & -3 \\ 0 & 0 & \boxed{1} & 2 \end{array}\right]$,

interchange Rows 2 and 3,
scale Row 1 by $\frac{1}{8}$,
scale Row 2 by $\frac{1}{9}$,
scale Row 3 by $\frac{1}{4}$,
replace Row 2 with Row 2 − 2Row 3
replace Row 1 with Row 1 − 2Row 2

8. $\left[\begin{array}{cccc|c} \boxed{3} & 0 & 6 & 9 & 18 \\ 0 & \boxed{2} & 4 & 0 & 8 \\ 0 & 0 & 0 & \boxed{3} & 6 \end{array}\right] \sim \left[\begin{array}{cccc|c} \boxed{1} & 0 & 2 & 0 & 9 \\ 0 & \boxed{1} & 2 & 0 & -3 \\ 0 & 0 & 0 & \boxed{1} & 2 \end{array}\right]$,

scale Row 1 by $\frac{1}{3}$,
scale Row 2 by $\frac{1}{2}$,
scale Row 3 by $\frac{1}{3}$,
replace Row 1 with Row 1 − 3Row 3

9. $\left[\begin{array}{ccc|c} 1 & -2 & 3 & 1 \\ 2 & -2 & 5 & 6 \\ 2 & 3 & -4 & 2 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & -1 & 3 & 1 \\ 0 & 5 & -10 & 0 \\ 0 & 0 & -1 & 4 \end{array}\right]$,

$\boldsymbol{x} = (5, -8, -4)$

10. $\left[\begin{array}{ccc|c} 0 & 2 & 6 & 4 \\ 1 & 3 & 2 & 5 \\ 3 & 3 & 3 & 3 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 3 & 2 & 5 \\ 0 & 2 & 6 & 4 \\ 0 & 0 & 15 & 0 \end{array}\right]$,

$\boldsymbol{x} = (-1, 2, 0)$

11. $\left[\begin{array}{ccc|c} 3 & 2 & 1 & 1 \\ 3 & 8 & 3 & 5 \\ 0 & 3 & 1 & 6 \end{array}\right] \sim \left[\begin{array}{ccc|c} 3 & 2 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 0 & 4 \end{array}\right]$,
inconsistent

12. $\left[\begin{array}{ccc|c} 2 & 3 & -1 & 7 \\ 4 & -2 & 3 & 5 \\ 1 & -1 & 2 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|c} 2 & 3 & -1 & 7 \\ 0 & -8 & 5 & -9 \\ 0 & 0 & \frac{15}{16} & \frac{5}{16} \end{array}\right]$,
$\boldsymbol{x} = \left(\frac{5}{3}, \frac{4}{3}, \frac{1}{3}\right)$

13. $\left[\begin{array}{ccc|c} 2 & -1 & 2 & 3 \\ 1 & 2 & -1 & 1 \\ 3 & 1 & 2 & 9 \end{array}\right] \sim \left[\begin{array}{ccc|c} 2 & -1 & 1 & 3 \\ 0 & \frac{5}{2} & -\frac{3}{2} & -\frac{1}{2} \\ 0 & 0 & \frac{4}{5} & \frac{8}{5} \end{array}\right]$,
$\boldsymbol{x} = (1, 1, 2)$

14. $\left[\begin{array}{ccc|c} 1 & 2 & 2 & 2 \\ 0 & -4 & 8 & 8 \\ 2 & 2 & 6 & 6 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 6 & 6 \\ 0 & 1 & -2 & -2 \\ 0 & 0 & 1 & 1 \end{array}\right]$,
inconsistent

♠ 15. $\left[\begin{array}{cc|c} 1 & 1 & 8 \\ 1 & -1 & 4 \end{array}\right] \sim \left[\begin{array}{cc|c} 1 & 0 & 6 \\ 0 & 1 & 2 \end{array}\right]$,
$\boldsymbol{x} = (6, 2)$

♠ 16. $\left[\begin{array}{ccc|c} 1 & -1 & 0 & 6 \\ 2 & 0 & -3 & 16 \\ 0 & 2 & 1 & 4 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 8 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{array}\right]$,
$\boldsymbol{x} = (8, 2, 0)$

17. $\left[\begin{array}{ccc|c} 0 & 3 & 6 & 3 \\ 1 & 2 & -3 & 2 \\ 4 & 0 & -3 & 0 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right]$,
$\boldsymbol{x} = (0, 1, 0)$

18. $\left[\begin{array}{ccc|c} 0 & 3 & -2 & 0 \\ 1 & 1 & 3 & 12 \\ 6 & 2 & 1 & 13 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array}\right]$,
$\boldsymbol{x} = (1, 2, 3)$

19. $\left[\begin{array}{ccc|c} 1 & 4 & 7 & 3 \\ 2 & 5 & 8 & 3 \\ 3 & 6 & 9 & 3 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & -1 & -1 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right]$,
$\boldsymbol{x} = (t - 1, -2t + 1, t)$

♠ 20. $\left[\begin{array}{ccc|c} 2 & -2 & -2 & 2 \\ 2 & 3 & 1 & 2 \\ 3 & 2 & 0 & 0 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & -2/5 & 1 \\ 0 & 1 & 3/5 & 0 \\ 0 & 0 & 0 & -15 \end{array}\right]$,
inconsistent

21. $\left[\begin{array}{ccc|c} 7 & 8 & 9 & 10 \\ 4 & 5 & 6 & 10 \\ 1 & 2 & 3 & 10 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & -1 & -10 \\ 0 & 1 & 2 & 10 \\ 0 & 0 & 0 & 0 \end{array}\right]$,
$\boldsymbol{x} = (10 + t, 10 - 2t, t)$

22. $\left[\begin{array}{ccc|c} 2 & 1 & -1 & -10 \\ 1 & -3 & -2 & -4 \\ 3 & 1 & 1 & 15 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & -5 \\ 0 & 0 & 1 & 11 \end{array}\right]$,
$\boldsymbol{x} = (3, -5, 11)$

23. $\left[\begin{array}{ccc|c} 2 & 1 & 4 & 199 \\ 2 & 10 & 15 & 984 \\ -1 & 0 & 10 & 58 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 42 \\ 0 & 1 & 0 & 75 \\ 0 & 0 & 1 & 10 \end{array}\right]$,
$\boldsymbol{x} = (42, 75, 10)$

24. $\left[\begin{array}{ccc|c} 5 & 2 & 6 & 11 \\ 0 & 1 & 2 & 5 \\ 3 & 2 & 1 & 7 \\ 2 & 1 & 4 & 6 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & -1/8 \\ 0 & 1 & 0 & 13/4 \\ 0 & 0 & 1 & 7/8 \\ 0 & 0 & 0 & -3/2 \end{array}\right]$,
inconsistent

25. $\left[\begin{array}{cccc|c} 1 & 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 0 & 1 \\ 1 & 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 4 & 0 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -2 \\ 0 & 0 & 0 & 1 & -1 \end{array}\right]$
$\boldsymbol{x} = (3, 1, -2, -1)$

26. $\left[\begin{array}{cccc|c} 2 & -3 & 6 & 2 & 5 \\ -2 & 3 & -3 & -3 & -4 \\ 4 & 6 & 9 & 5 & 9 \\ -2 & 3 & 3 & -4 & 1 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & -\frac{3}{2} & 0 & 0 & -\frac{9}{2} \\ 0 & 0 & 1 & 0 & \frac{4}{3} \\ 0 & 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$
$\boldsymbol{x} = \left(-\frac{9}{2} + \frac{3}{2}t, t, \frac{4}{3}, 3\right)$

27. $\left[\begin{array}{ccc|c} 2 & 4 & -2 & 10 \\ 16 & -23 & 5 & 33 \\ 4 & 2 & -6 & 2 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array}\right]$,
$\boldsymbol{x} = (4, 2, 3)$

2.4 Applications of Linear Systems

♠ 1. The cost for one loaf of bread is \$2.39, for one dozen eggs is \$2.79, and one apple is \$0.79.

♠ 2. $f(x) = -\frac{1}{6}x^3 + \frac{19}{6}x - 2$

3. $f(x) = -3x^5 + 7x^4 - x^3 + 2x^2 - x + 5$

4. $f(x) = \dfrac{x^2 - 3x + 2}{2x^2 - x + 1}$

2.5 Vector Equations

1. $x_1 \begin{bmatrix} 1 \\ 8 \end{bmatrix} + x_2 \begin{bmatrix} 7 \\ -3 \end{bmatrix} = \begin{bmatrix} 16 \\ 6 \end{bmatrix}$

♠ 2. $x_1 \begin{bmatrix} 3 \\ -1 \end{bmatrix} + x_2 \begin{bmatrix} 7 \\ -2 \end{bmatrix} + x_3 \begin{bmatrix} -5 \\ 6 \end{bmatrix} = \begin{bmatrix} -5 \\ 4 \end{bmatrix}$

♠ 3. $x_1 \begin{bmatrix} -2 \\ 1 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 6 \\ -2 \\ -19 \end{bmatrix} + x_3 \begin{bmatrix} -11 \\ 5 \\ 31 \end{bmatrix} = \begin{bmatrix} 6 \\ -3 \\ -17 \end{bmatrix}$

4. $x_1 \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} + x_3 \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$

5. $x_1 \begin{bmatrix} 1 \\ 7 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 3 \\ 6 \\ 1 \end{bmatrix} + x_3 \begin{bmatrix} 0 \\ -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 7 \\ 8 \\ 3 \end{bmatrix}$

6. $x_1 \begin{bmatrix} 2 \\ 1 \\ -3 \end{bmatrix} + x_2 \begin{bmatrix} 4 \\ -6 \\ 5 \end{bmatrix} + x_3 \begin{bmatrix} -1 \\ 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 \\ 12 \\ 9 \end{bmatrix}$

7. $\begin{cases} 9x_1 + 7x_2 + x_3 = 7 \\ 3x_1 + 3x_2 - x_3 = 5 \end{cases}$

8. $\begin{cases} 7x_1 - 3x_2 + 6x_3 + 9x_4 = 17 \\ -3x_1 - x_2 + 3x_3 + 3x_4 = 4 \\ 4x_1 - x_2 - 8x_3 + x_4 = -12 \\ 2x_1 + 5x_3 + 3x_4 = 8 \end{cases}$

9. yes, $\boldsymbol{b} = \boldsymbol{a}_1 + \boldsymbol{a}_2$

♠ 10. yes, $\boldsymbol{b} = 3\boldsymbol{a}_1 + 2\boldsymbol{a}_2$

♠ 11. no, the corresponding linear system is inconsistent

♠ 12. yes, $\boldsymbol{b} = 3\boldsymbol{a}_1 + 4\boldsymbol{a}_2$

13. yes, $\boldsymbol{a}_1 + 2\boldsymbol{a}_2 + 3\boldsymbol{a}_3 + 4\boldsymbol{a}_4$

14. no

♠ 15. yes, $\boldsymbol{b} = -\boldsymbol{a}_1 + \boldsymbol{a}_2 + \boldsymbol{a}_3$, for example. There are, in fact, infinitely many possible linear combinations. A dependency relation occurs whenever $x_1 = 2 - 3x_3$ and $x_2 = 3 - 2x_3$.

16. no

2.6 Matrix Equations

1. $\begin{bmatrix} 30 \\ 5 \\ 9 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 11 \\ 6 \\ -2 \end{bmatrix}$

3. $\begin{bmatrix} 128 \\ 111 \end{bmatrix}$

♠ 4. $\begin{cases} x + 2y = 2 \\ 3x + 4y = 0 \\ -x - 2y = 2 \\ 4x + 5y = -9 \end{cases}$

5. $\begin{cases} 21x + 27y + 17z - 32w = 12 \\ 4x + 42y + 6z + 19w = -6 \\ 103x - 72y + 17z - 8w = 15 \\ 2x + 11y - 10z + 13w = 1 \end{cases}$

6. $\begin{bmatrix} 1 & -2 & -3 \\ 0 & 1 & 2 \\ -2 & 0 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -4 \\ 1 \\ 8 \end{bmatrix}$

7. $\begin{bmatrix} 3 & 0 & 1 & 2 \\ -1 & 4 & 5 & -2 \\ 6 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 17 \\ 15 \end{bmatrix}$

8. $\begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$

♠ 9. $\begin{bmatrix} -7 \\ 9 \\ 2 \end{bmatrix}$

♠ 10. inconsistent

11. $\begin{bmatrix} 2 \\ -1 \\ 3 \\ -4 \end{bmatrix}$

♠ 12. $T(1,0,0,1) = (5,1)$,
$T(1,2,3,4) = (12,10)$

♠ 13. $\boldsymbol{x} = (2/3, 1/3, 0, 0)$
(Answers may vary)

14. $T(1,0,-1,3,0) = (7,-5)$,
$T(1,2,3,4,5) = (38,2)$

15. $\boldsymbol{x} = (1,0,-2,2,3)$
(Answers may vary)

♠ 16. Each entry x_i would be the total value if we converted all notes into currency i.

2.7 Linear Independence

1. linearly independent

♠ 2. linearly dependent;
$2\boldsymbol{v}_1 - \boldsymbol{v}_2 + 0\boldsymbol{v}_3 = \boldsymbol{0}$

3. linearly independent

♠ 4. linearly independent

5. linearly dependent;
$2\boldsymbol{v}_1 - 3\boldsymbol{v}_2 - \boldsymbol{v}_3 = \boldsymbol{0}$

2.8 Solution Sets of Linear Systems

1. rank $A = 3$, nullity $A = 0$,
$\text{Col}(A) = \text{Span}\left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 15 \\ 9 \\ 0 \end{bmatrix}, \begin{bmatrix} 8 \\ 6 \\ 2 \end{bmatrix} \right\}$,
$\text{Nul}(A) = \text{Span}\,\{\}$

2. rank $A = 3$, nullity $A = 0$,
$\text{Col}(A) = \text{Span}\left\{ \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}, \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 5 \\ 4 \end{bmatrix} \right\}$,
$\text{Nul}(A) = \text{Span}\,\{\}$

3. rank $A = 3$, nullity $A = 1$,
$\text{Col}(A) = \text{Span}\left\{ \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \right\}$,

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix} 3 \\ 2 \\ -5 \\ 4 \end{bmatrix}\right\}$$

4. $\text{rank}\, A = 3$, $\text{nullity}\, A = 1$,

$$\text{Col}(A) = \text{Span}\left\{\begin{bmatrix} 5 \\ 4 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 \\ 4 \\ 5 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \\ 6 \end{bmatrix}\right\},$$

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix} 7 \\ -13 \\ 8 \\ 1 \end{bmatrix}\right\}$$

♠ 5. $\text{rank}\, A = 2$, $\text{nullity}\, A = 3$,

$$\text{Col}(A) = \text{Span}\left\{\begin{bmatrix} 8 \\ -2 \end{bmatrix}, \begin{bmatrix} -3 \\ 1 \end{bmatrix}\right\},$$

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix} 2 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -3 \\ -3 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -4 \\ -5 \\ 0 \\ 0 \\ 1 \end{bmatrix}\right\}$$

♠ 6. $\text{rank}\, A = 2$, $\text{nullity}\, A = 2$,

$$\text{Col}(A) = \text{Span}\left\{\begin{bmatrix} -1 \\ 7 \\ -3 \end{bmatrix}, \begin{bmatrix} 0 \\ -8 \\ 2 \end{bmatrix}\right\},$$

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix} 2 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}\right\}$$

7. $\text{rank}\, A = 2$, $\text{nullity}\, A = 2$,

$$\text{Col}(A) = \text{Span}\left\{\begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 4 \end{bmatrix}\right\},$$

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix} 3 \\ -4 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 17 \\ -13 \\ 0 \\ 1 \end{bmatrix}\right\}$$

8. $\text{rank}\, A = 3$, $\text{nullity}\, A = 1$,

$$\text{Col}(A) = \text{Span}\left\{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 4 \\ 4 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}\right\},$$

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix} 0 \\ -2 \\ 1 \\ 0 \end{bmatrix}\right\}$$

9. $\text{rank}\, A = 3$, $\text{nullity}\, A = 3$,

$$\text{Col}(A) = \text{Span}\left\{\begin{bmatrix} 5 \\ 0 \\ 9 \end{bmatrix}, \begin{bmatrix} 0 \\ 4 \\ 0 \end{bmatrix}, \begin{bmatrix} 6 \\ 4 \\ 9 \end{bmatrix}\right\},$$

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 10 \\ 7 \\ 0 \\ 0 \\ -10 \\ 2 \end{bmatrix}\right\}$$

10. $\text{rank}\, A = 4$, $\text{nullity}\, A = 2$,

$$\text{Col}(A) = \text{Span}\left\{\begin{bmatrix} 6 \\ 5 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 5 \\ 4 \\ 4 \\ 1 \end{bmatrix}, \begin{bmatrix} 4 \\ 3 \\ 6 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 2 \\ 2 \end{bmatrix}\right\},$$

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 23 \\ -50 \\ 18 \\ 0 \\ 19 \\ 2 \end{bmatrix}\right\}$$

11. rank $A = 2$, nullity $A = 3$,

$$\text{Col}(A) = \text{Span}\left\{\begin{bmatrix}1\\2\\3\\2\end{bmatrix}, \begin{bmatrix}2\\4\\6\\-2\end{bmatrix}\right\},$$

$$\text{Nul}(A) = \text{Span}\left\{\begin{bmatrix}7\\1\\3\\0\\0\end{bmatrix}, \begin{bmatrix}-8\\-2\\0\\3\\0\end{bmatrix}, \begin{bmatrix}13\\1\\0\\0\\3\end{bmatrix}\right\}$$

12. $\boldsymbol{x} = \begin{bmatrix}1\\-\frac{3}{4}\\0\end{bmatrix} + x_3\begin{bmatrix}-1\\0\\1\end{bmatrix}$

♠ 13. $\boldsymbol{x} = \begin{bmatrix}1\\2\\0\end{bmatrix} + t\begin{bmatrix}2\\3\\1\end{bmatrix}$

♠ 14. $\boldsymbol{x} = \begin{bmatrix}1\\2\\3\\0\end{bmatrix} + t\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

15. $\boldsymbol{x} = \begin{bmatrix}5\\0\\0\\-8\end{bmatrix} + s\begin{bmatrix}3\\1\\0\\0\end{bmatrix} + t\begin{bmatrix}-4\\0\\1\\0\end{bmatrix}$

2.9 Orthogonality

1. orthogonal

2. no, but $(6,1) \perp (8,-24)$

3. orthogonal

4. no, but $(2,8) \perp (4,-1)$

5. no, but $(1,2,-1) \perp (3,-1,1)$

6. no, but $(-2,-1,2) \perp (3,2,4)$

7. no

8. no

♠ 9. $3x_1 - 2x_2 = -1$

♠ 10. $x_1 - 2x_2 + 2x_3 - x_4 = 0$

♠ 11. $3x_1 + 6x_2 - 3x_3 - 2x_4 + 5x_5 = 23$

12. $x_1+3x_2+5x_3+7x_4+9x_5 = 6$

♠ 13. $\boldsymbol{y} = -2\begin{bmatrix}1\\3\end{bmatrix}$

♠ 14. $\boldsymbol{y} = 5\begin{bmatrix}1\\2\\1\end{bmatrix} - 6\begin{bmatrix}1\\0\\-1\end{bmatrix}$

♠ 15. $\boldsymbol{y} = \begin{bmatrix}5\\4\\-2\end{bmatrix} + 6\begin{bmatrix}2\\-2\\1\end{bmatrix}$

♠ 16. $\boldsymbol{y} = \begin{bmatrix}3\\4\\1\\0\end{bmatrix} - 2\begin{bmatrix}3\\-1\\-5\\1\end{bmatrix} + 3\begin{bmatrix}-1\\1\\-1\\-1\end{bmatrix}$

♠ 17. $\boldsymbol{y} = 2\begin{bmatrix}1\\6\\-6\\0\\1\end{bmatrix} - 3\begin{bmatrix}7\\-2\\0\\5\\5\end{bmatrix}$

Chapter 3 : The Algebra and Geometry of Matrices

3.1 Matrix Operations

1. $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 2 & -1 & 8 \\ 13 & -11 & 0 \\ 8 & -2 & -5 \end{bmatrix}$

♠ 3. $\begin{bmatrix} 4 & -7 & -6 \\ 31 & 3 & -6 \\ -2 & -8 & 5 \end{bmatrix}$

♠ 4. $\begin{bmatrix} 14 & 23 & 21 & -16 \\ 9 & -21 & -28 & 17 \\ -2 & -17 & -19 & 12 \end{bmatrix}$

♠ 5. $\begin{bmatrix} 1 & 4 & 1 \\ 2 & -3 & -2 \\ 3 & 0 & -1 \end{bmatrix}$

♠ 6. $\begin{bmatrix} 3 & 1 & 3 \\ 0 & 7 & 3 \\ -1 & 8 & 2 \\ 2 & -3 & -4 \end{bmatrix}$

♠ 7. $\begin{bmatrix} 2 & -5 & 0 & 1 \\ 3 & 0 & -2 & 2 \\ -2 & 7 & 4 & 3 \end{bmatrix}$

♠ 8. -3

♠ 9. -8

♠ 10. $\begin{bmatrix} 3 & -29 & 10 \\ 42 & 39 & -31 \\ 14 & 18 & -12 \\ 28 & -9 & -7 \end{bmatrix}$

♠ 11. $\begin{bmatrix} 9 & -6 & 6 \\ 0 & 6 & 12 \\ -6 & 6 & -3 \end{bmatrix}$

12. $\begin{bmatrix} 3 & 3 \\ 2 & 8 \\ 7 & 14 \end{bmatrix}$

13. $\begin{bmatrix} 1 & 7 \\ 26 & 27 \end{bmatrix}$

14. $\begin{bmatrix} -7 & -2 \\ 2 & -4 \\ 6 & -8 \end{bmatrix}$

15. $\begin{bmatrix} 19 & -14 \\ 29 & -54 \end{bmatrix}$

16. $7, \begin{bmatrix} 8 & 0 & 3 \\ -7 & -2 & 2 \\ 2 & 4 & 1 \end{bmatrix}, 7$

17. $2, \begin{bmatrix} 1/8 & 9/8 & 1 & 1/4 \\ -7/8 & -1/2 & -12/15 & 8/15 \\ 2/5 & 1/4 & 2 & -3/8 \\ -12/13 & 1/4 & 1/5 & 3/8 \end{bmatrix}, 2$

18. $6, \begin{bmatrix} 5 & 3 & 5 & 2 \\ 1 & 0 & 2 & 3 \\ 6 & 3 & 1 & 0 \\ 2 & 4 & 1 & 0 \end{bmatrix} \pmod 7, 6$

19. $3+9i, \begin{bmatrix} 1+3i & 2+4i & -1-2i \\ -5+5i & -8i & 3+6i \\ 2-2i & 6-i & 2-4i \end{bmatrix}, 3-9i$

3.2 Matrix Applications

♠ 1. $\begin{bmatrix} \mathbf{0} & I_{n-2} & \mathbf{0} \end{bmatrix}$

♠ 2. $\begin{bmatrix} \mathbf{0} & \boldsymbol{e}_1 & \mathbf{0} & \boldsymbol{e}_2 & \mathbf{0} & \boldsymbol{e}_3 & \dots & \mathbf{0} & \boldsymbol{e}_{\frac{n}{2}} \end{bmatrix}$

♠ 3. $\begin{bmatrix} \boldsymbol{e}_1 + \frac{1}{2}\boldsymbol{e}_2 & \frac{1}{2}\boldsymbol{e}_2 + \boldsymbol{e}_3 + \frac{1}{2}\boldsymbol{e}_4 & \frac{1}{2}\boldsymbol{e}_4 + \boldsymbol{e}_5 + \frac{1}{2}\boldsymbol{e}_6 & \dots & \frac{1}{2}\boldsymbol{e}_{2n-4} + \boldsymbol{e}_{2n-3} + \frac{1}{2}\boldsymbol{e}_{2n-2} & \frac{1}{2}\boldsymbol{e}_{2n-2} + \boldsymbol{e}_{2n-1} \end{bmatrix}$

3.3 Nonsingular Matrices

1. $\frac{1}{2}\begin{bmatrix} -3 \\ 2 \end{bmatrix}$ ♠ 2. $\begin{bmatrix} 29 \\ -25 \\ 7 \end{bmatrix}$ ♠ 3. $\begin{bmatrix} 18 \\ 15 \\ -11 \\ 8 \end{bmatrix}$

4. $\begin{bmatrix} -1 \\ 1 \end{bmatrix}$ 5. $\begin{bmatrix} 5 \\ -3 \end{bmatrix}$ 6. $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

7. singular

8. singular

9. $\begin{bmatrix} -2 & 7 & -1 \\ -2 & 6 & -1 \\ -3 & 10 & -1 \end{bmatrix}$ ♠ 10. $\frac{1}{6}\begin{bmatrix} 3 & 3 & -3 \\ -6 & -12 & 24 \\ 5 & 7 & -9 \end{bmatrix}$ 11. $\begin{bmatrix} -\frac{25}{8} & 4 & -\frac{3}{8} \\ -2 & 2 & 0 \\ -\frac{3}{8} & 0 & -\frac{1}{8} \end{bmatrix}$

3.4 Linear Transformations

1. $T((x_1, y_1, z_1) + (x_2, y_2, z_2)) = T(x_1 + x_2, y_1 + y_2, z_1 + z_2) = \begin{bmatrix} (x_1 + x_2) + (y_1 + y_2) - 2(z_1 + z_2) \\ -(y_1 + y_2) + (z_1 + z_2) \end{bmatrix} =$
$\begin{bmatrix} x_1 + x_2 + y_1 + y_2 - 2z_1 - 2z_2 \\ -y_1 - y_2 + z_1 + z_2 \end{bmatrix} = \begin{bmatrix} x_1 + y_1 - 2z_1 \\ -y_1 + z_1 \end{bmatrix} + \begin{bmatrix} x_2 + y_2 - 2z_2 \\ -y_2 + z_2 \end{bmatrix} = T(x_1, y_1, z_1) + T(x_2, y_2, z_2)$
and $T(c(x, y, z)) = T(cx, xy, cz) = \begin{bmatrix} (cx) + (cy) - 2(cz) \\ -(cx) + (cz) \end{bmatrix} = \begin{bmatrix} c(x + y - 2z) \\ c(-y + z) \end{bmatrix} = c\begin{bmatrix} x + y - 2z \\ -y + z \end{bmatrix} =$
$cT(x, y, z)$. Therefore, T is linear.

♠ 2. $T(1, 2, 3) = (-3, 1)$, $T(1, 0, -2) = (5, -2)$

♠ 3. yes, $T(2, 1, 0) = (3, -1)$

4. $T(1, 3, 2) = (7, 6, -6)$

5. yes, $T(1, -1, 1) = (1, 6, 2)$

6. (Answers may vary). Note that $T(2, 0, 0) = (0, 8, 0) \neq (0, 12, 0) = 2(0, 6, 0) = 2T(1, 0, 0)$. Thus, scalar multiplication is NOT preserved. Thus, T is not linear.

7. $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ 8. $\begin{bmatrix} 2 & 2 \\ 1 & 3 \end{bmatrix}$ ♠ 9. $\begin{bmatrix} 2 & -1 \\ 0 & -4 \end{bmatrix}$ 10. $\begin{bmatrix} 1 & 1 & 0 \\ 2 & 0 & 3 \\ 1 & 2 & 2 \end{bmatrix}$ ♠ 11. $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

12. $\begin{bmatrix} 1 & 2 & 1 \\ 3 & -5 & 14 \\ -5 & 3 & -18 \\ -7 & 19 & -40 \end{bmatrix}$

13. $[S \circ T] = [S][T] = \begin{bmatrix} 2 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 6 \\ 1 & 5 \end{bmatrix}$

♠ 14. $[S \circ T] = [S][T] = \begin{bmatrix} 0 & -2 & 1 \\ 1 & 0 & -2 \end{bmatrix} \begin{bmatrix} 0 & 2 \\ 3 & 0 \\ 2 & -3 \end{bmatrix} = \begin{bmatrix} -4 & -3 \\ -4 & 8 \end{bmatrix}$

3.5.3 Linear Transformations on $\mathbb{R}^2$

1. $\begin{bmatrix} 0 & -4 \\ -1 & 0 \end{bmatrix}$

♠ 2. $\begin{bmatrix} -2 & 0 \\ 0 & 3 \end{bmatrix}$

♠ 3. $\frac{\sqrt{2}}{4} \begin{bmatrix} -2 & 2 \\ 1 & 1 \end{bmatrix}$

♠ 4. $\begin{bmatrix} 0 & -1 \\ 2 & 0 \end{bmatrix}$

5. $-\frac{1}{2} \begin{bmatrix} 2 & 2\sqrt{3} \\ \sqrt{3} & -1 \end{bmatrix}$

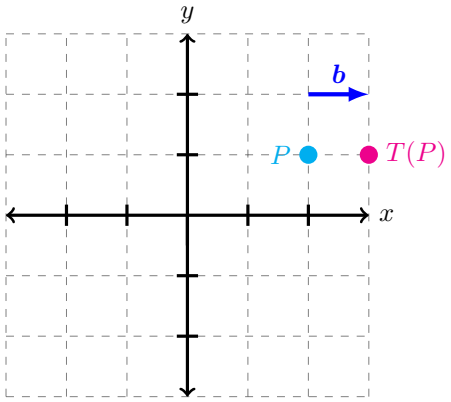
6. $\begin{bmatrix} 3 & 0 \\ 0 & -3 \end{bmatrix}$

7. $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$

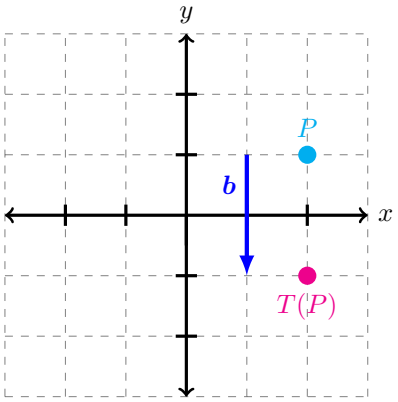
8. $\begin{bmatrix} -2 & 0 \\ 0 & -1 \end{bmatrix}$

9. $\begin{bmatrix} 2 & -2 \\ 1 & 0 \end{bmatrix}$

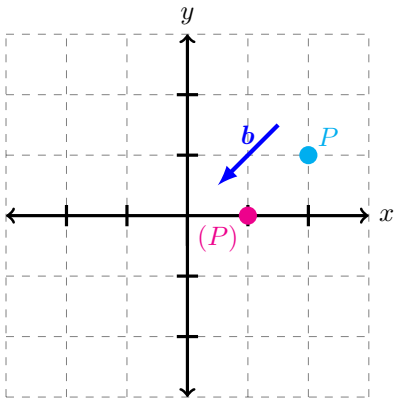
♠ 10. $(3, 1)$



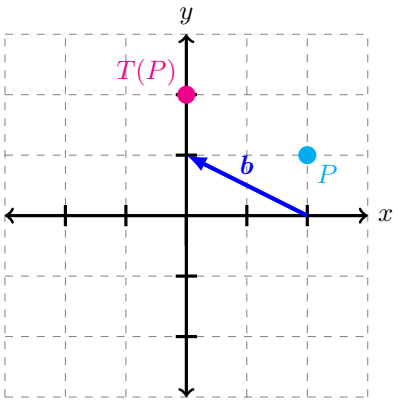
♠ 11. $(2, -1)$



♠ 12. $(1, 0)$



♠ 13. $(0, 2)$



14. $(1, -5, 5)$

15. $\begin{bmatrix} -4 \\ -4 \end{bmatrix}$

16. $\begin{bmatrix} 4 \\ -5 \\ 8 \end{bmatrix}$

17. $\begin{bmatrix} 7 \\ 10 \\ -6 \end{bmatrix}$

18. $\begin{bmatrix} 11 \\ 42 \\ 44 \end{bmatrix}$

19. $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

♠ 20. $\begin{bmatrix} -4 \\ 20 \\ 14 \end{bmatrix}$

♠ 21. $\begin{bmatrix} -7 \\ 14 \\ -6 \end{bmatrix}$

22. $\begin{bmatrix} 23 \\ 10 \\ 33 \end{bmatrix}$

23. $\begin{bmatrix} -4 \\ 4 \\ 2 \end{bmatrix}$

3.6 The Gram-Schmidt Algorithm

♠ 1. $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} \sqrt{2} & 2\sqrt{2} \\ 0 & 2\sqrt{2} \end{bmatrix}$

♠ 3. $\begin{bmatrix} \frac{2}{3} & \frac{2}{3} \\ -\frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & -\frac{2}{3} \end{bmatrix} \begin{bmatrix} 6 & -3 \\ 0 & 6 \end{bmatrix}$

♠ 4. $\begin{bmatrix} 1/\sqrt{11} & 6/\sqrt{209} \\ 0 & 11/\sqrt{209} \\ 3/\sqrt{11} & -4/\sqrt{209} \\ 1/\sqrt{11} & 6/\sqrt{209} \end{bmatrix} \begin{bmatrix} \sqrt{11} & \frac{16}{\sqrt{11}} \\ 0 & \frac{19}{\sqrt{209}} \end{bmatrix}$

♠ 5. $\begin{bmatrix} \frac{1}{3} & \frac{2}{3} & -\frac{2}{3} \\ -\frac{2}{3} & \frac{2}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{1}{3} & \frac{2}{3} \end{bmatrix} \begin{bmatrix} 3 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$

3.7 Linear Dynamical Systems

1. $\begin{bmatrix} 13 \\ 22 \end{bmatrix}, \begin{bmatrix} 79 \\ 136 \end{bmatrix}, \begin{bmatrix} 487 \\ 838 \end{bmatrix}$ ♠ 2. $\begin{bmatrix} 8 \\ 16 \end{bmatrix}, \begin{bmatrix} 40 \\ 80 \end{bmatrix}, \begin{bmatrix} 200 \\ 400 \end{bmatrix}$ ♠ 3. $\begin{bmatrix} 26 \\ 5 \\ 27 \end{bmatrix}, \begin{bmatrix} 276 \\ 161 \\ 118 \end{bmatrix}, \begin{bmatrix} 2134 \\ 495 \\ 1165 \end{bmatrix}$

♠ 4. $\begin{bmatrix} -54 \\ 62 \\ 27 \\ 1 \end{bmatrix}, \begin{bmatrix} -15 \\ -303 \\ 70 \\ -116 \end{bmatrix}, \begin{bmatrix} -367 \\ 191 \\ 16 \\ -1334 \end{bmatrix}$ 5. $\begin{bmatrix} 4 \\ 6 \\ -12 \\ 20 \\ 30 \end{bmatrix}, \begin{bmatrix} 14 \\ 18 \\ 48 \\ 100 \\ 180 \end{bmatrix}, \begin{bmatrix} 46 \\ 54 \\ -192 \\ 500 \\ 1080 \end{bmatrix}$ 6. $\begin{bmatrix} 46 \\ 1 \\ -15 \\ -7 \\ 26 \end{bmatrix}, \begin{bmatrix} 264 \\ -1 \\ -157 \\ 169 \\ 22 \end{bmatrix}, \begin{bmatrix} 862 \\ 595 \\ 771 \\ 493 \\ 380 \end{bmatrix}$

3.8 Applications of Linear Dynamical Systems

♠ 1. $A = \begin{bmatrix} 0.9 & 0.0 \\ 0.1 & 0.95 \end{bmatrix}$, $\boldsymbol{x}_0 = \begin{bmatrix} 330 \\ 10^{-5} \end{bmatrix}$

2. 332.3 million, 4.298 million

Chapter 4 : Least-Squares

4.1 The Least-Squares Problem

1. $\frac{1}{106}\begin{bmatrix} 116 \\ -399 \end{bmatrix}$ ♠ 2. $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ 3. $\frac{9}{14}\begin{bmatrix} -1 \\ 7 \end{bmatrix}$ 4. $\frac{1}{3}\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 5. $\frac{1}{35}\begin{bmatrix} 53 \\ 49 \end{bmatrix}$

♠ 6. $\begin{bmatrix} -2 \\ 3 \end{bmatrix}$ ♠ 7. $\frac{1}{3}\begin{bmatrix} 1 \\ -1 \\ -2 \end{bmatrix}$ 8. $\begin{bmatrix} -2 \\ 0 \\ 1 \end{bmatrix}$ 9. $\begin{bmatrix} 0.7 \\ 1.2 \end{bmatrix}$ 10. $\begin{bmatrix} 3.5 \\ 1.4 \end{bmatrix}$

4.2 Data Fitting

♠ 1. $f(x) = \frac{3}{2}x + \frac{5}{6}$

♠ 2. $f(x) = x^2 + \frac{2}{5}x + \frac{3}{10}$

3. $f(x) = -7.706 + 39.07x + 51.03x^2 - 34.05x^3 - 5.08x^4 + 2.71x^5$

4. $f(x) = \frac{0.4001 - 0.6961x + 0.1529x^2}{1 - 0.06324x - 0.07445x^2}$

4.3 Regression

♠ 1. $f(x) = x^2 + \frac{2}{5}x + \frac{3}{10}$, $\rho = 0.992831448793946$

2. $y = -15.13691844x_1^2 - 2.68474092x_2^2 - 1.85095432a_3x_1x_2 + 224.02918964a_4x_1 - 4.2959796a_5x_2 - 27.75275964$, $\rho = 0.7542234751727548$

3. $y = 2.35x - 7$, $\widehat{y} = 216.25$
4. $y = 13x - 45.7$, $\widehat{y} = 97.7$

4.4 Validation

1. $\approx 29\%$

4.5 Feature Engineering

1. $\approx 29.6\%$, the new model performs worse, the new model examines the interaction of the home being a condo in the different ZIP codes.
2. $\rho = 0.751$; (Answers may vary) c. Interpret the coefficient on area. Does it make sense? A 1000 square foot increase in area leads to a predicted increase of \$149,350. This makes sense because a 1000 square foot increase is a big leap, and larger houses tend to be considered more valuable. Increasing bedrooms is predicted to decrease sale price with all else held equal, that condos tend to sell for less money, and that the correlation between selling price and predicted selling price is moderately strong, meaning the model is somewhat accurate but not necessarily trustworthy.

4.6 Classification

1. `[ 0.42810996 0.64605445 -2.73590876] False True True`
2. `setosa`
3. `versicolor`

4.7 Multi-Objective Least Squares

♠ 1. $\begin{bmatrix} 2 \\ \frac{1}{3} \end{bmatrix}$

♠ 2. $\begin{bmatrix} 0.66918127 \\ -0.01092971 \end{bmatrix}$

4.8 Regularization

♠ 1. $f(x) = 2.501x^5 - 4.385x^4 - 32.74x^3 + 47.38x^2 + 36.47x - 5.539$

♠ 2. $f(x) = \dfrac{0.1534x^2 - 0.6837x + 0.3858}{-0.07681x^2 - 0.05834x + 1}$

4.9 Constrained Least Squares

♠ 1. $\begin{bmatrix} -\frac{3}{4} \\ -\frac{9}{4} \end{bmatrix}$

2. $\begin{bmatrix} \frac{51}{185} \\ \frac{17}{185} \end{bmatrix}$

Chapter 5 : Eigenvalues

5.1 Eigenvalues and Eigenvectors

1. yes, $\lambda = 5$
2. no
3. no
4. no
5. no
6. yes, $\lambda = 3$

♠ 7. yes, $\lambda = 2$

♠ 8. no

♠ 9. yes, $\lambda = -1$

♠ 10. yes, $\lambda = 1$

♠ 11. yes, $\lambda = 0$

♠ 12. no

13. $c\begin{bmatrix} 1 \\ 0 \end{bmatrix}$

♠ 14. $c\begin{bmatrix} 1 \\ 0 \end{bmatrix}$

♠ 15. $c\begin{bmatrix} 3 \\ 1 \end{bmatrix}$

♠ 16. $c\begin{bmatrix} 1 \\ 1 \end{bmatrix}$

17. $c\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$

18. $\lambda = -1$, mult $= 1$, $\left\{\begin{bmatrix} -2 \\ 1 \end{bmatrix}\right\}$;
$\lambda = 4$, mult $= 1$, $\left\{\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right\}$

♠ 19. $\lambda = 5$, mult $= 1$, $\left\{\begin{bmatrix} 2 \\ -4 \\ 1 \end{bmatrix}\right\}$;
$\lambda = -3$, mult $= 1$, $\left\{\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}\right\}$

♠ 20. $\lambda = -3$, mult $= 1$, $\left\{\begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}\right\}$;
$\lambda = 2$, mult $= 2$, $\left\{\begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ 2 \\ 0 \end{bmatrix}\right\}$

5.2 Diagonalization

1. $\begin{bmatrix} -1 & -4 \\ 1 & 5 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}\begin{bmatrix} -5 & -4 \\ 1 & 1 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}$

3. $\begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} 5 & -2 \\ -2 & 1 \end{bmatrix}$

4. $\begin{bmatrix} 1 & 1 \\ 1 & -2 \end{bmatrix}\begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix}\begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & -\frac{1}{3} \end{bmatrix}$

5. $\begin{bmatrix} 1 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}\begin{bmatrix} 1 & 1/2 & 0 \\ 0 & -1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

♠ 6. $\begin{bmatrix} 1 & 2 & 2 \\ 0 & 1 & 1 \\ -1 & 0 & 1 \end{bmatrix}\begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}\begin{bmatrix} 1 & -2 & 0 \\ -1 & 3 & -1 \\ 1 & -2 & 1 \end{bmatrix}$

7. $\begin{bmatrix} -1 & -1 & -2 \\ 2 & -6 & -3 \\ 1 & 13 & 2 \end{bmatrix}\begin{bmatrix} -4 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}\begin{bmatrix} -\frac{9}{28} & \frac{2}{7} & \frac{3}{28} \\ \frac{1}{12} & 0 & \frac{1}{12} \\ -\frac{8}{21} & -\frac{1}{7} & -\frac{2}{21} \end{bmatrix}$

♠ 8. $\begin{bmatrix} 2 & 0 & 2 & 5 \\ 1 & 0 & 1 & 3 \\ 4 & 1 & 0 & 12 \\ 5 & 0 & 6 & 15 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 3 & 0 & 0 & -1 \\ 0 & -24 & 1 & 4 \\ 0 & -5 & 0 & 1 \\ -1 & 2 & 0 & 0 \end{bmatrix}$

9. $\begin{bmatrix} 4 & 2 & -1 \\ 1 & -3 & 0 \\ 2 & 1 & 1 \end{bmatrix}\begin{bmatrix} 18 & 0 & 0 \\ 0 & -17 & 0 \\ 0 & 0 & -2 \end{bmatrix}\left(\frac{1}{21}\begin{bmatrix} 3 & 3 & 3 \\ 1 & -6 & 1 \\ -7 & 0 & 14 \end{bmatrix}\right)$

10. $\begin{bmatrix} -1 & -4 & 0 \\ 0 & -3 & 1 \\ 1 & 2 & 0 \end{bmatrix}\begin{bmatrix} 0 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -4 \end{bmatrix}\left(\frac{1}{2}\begin{bmatrix} 2 & 0 & 4 \\ -1 & 0 & -1 \\ -3 & 2 & -3 \end{bmatrix}\right)$

11. $\begin{bmatrix} 2 & 3 & 2 \\ 3 & 4 & 3 \\ 1 & 1 & 2 \end{bmatrix}\begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 8 \end{bmatrix}\begin{bmatrix} -5 & 3 & 1 \\ 3 & -2 & 0 \\ 1 & -1 & 1 \end{bmatrix}$

12. $\begin{bmatrix} -1 & -1 & -6 \\ 0 & 0 & -2 \\ 2 & 1 & 3 \end{bmatrix}\begin{bmatrix} 0 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -4 \end{bmatrix}\left(\frac{1}{2}\begin{bmatrix} 2 & -3 & 2 \\ -4 & 9 & -2 \\ 0 & -1 & 0 \end{bmatrix}\right)$

13. $\begin{bmatrix} -2 & 0 & -1 \\ -1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}\begin{bmatrix} 2 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & -2 \end{bmatrix}\begin{bmatrix} -1 & 0 & -1 \\ -1 & 1 & -1 \\ 1 & 0 & 2 \end{bmatrix}$

14. similar; note that A and B are both similar to $\begin{bmatrix} 0 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -4 \end{bmatrix}$.

5.3 Orthogonal Diagonalization

1. $\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

2. $\begin{bmatrix} \frac{5}{\sqrt{26}} & -\frac{1}{\sqrt{26}} \\ \frac{1}{\sqrt{26}} & \frac{5}{\sqrt{26}} \end{bmatrix}\begin{bmatrix} 18 & 0 \\ 0 & -8 \end{bmatrix}\begin{bmatrix} \frac{5}{\sqrt{26}} & \frac{1}{\sqrt{26}} \\ -\frac{1}{\sqrt{26}} & \frac{5}{\sqrt{26}} \end{bmatrix}$

3. $\begin{bmatrix} \frac{5}{\sqrt{50}} & 0 & -\frac{5}{\sqrt{50}} \\ \frac{3}{\sqrt{50}} & -\frac{4}{5} & \frac{3}{\sqrt{50}} \\ \frac{4}{\sqrt{50}} & \frac{3}{4} & \frac{4}{\sqrt{50}} \end{bmatrix}\begin{bmatrix} 9 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & -1 \end{bmatrix}\begin{bmatrix} \frac{5}{\sqrt{50}} & \frac{3}{\sqrt{50}} & \frac{4}{\sqrt{50}} \\ 0 & -\frac{4}{5} & \frac{3}{5} \\ -\frac{5}{\sqrt{50}} & \frac{3}{\sqrt{50}} & \frac{4}{\sqrt{50}} \end{bmatrix}$

4. $\begin{bmatrix} -\frac{\sqrt{2}}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2+\sqrt{2} & 0 \\ 0 & 0 & 2-\sqrt{2} \end{bmatrix}\begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ \frac{1}{2} & \frac{\sqrt{2}}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{\sqrt{2}}{2} & \frac{1}{2} \end{bmatrix}$

♠ 5. $\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & 0 & -\frac{2}{\sqrt{6}} \end{bmatrix}\begin{bmatrix} 30 & 0 & 0 \\ 0 & -12 & 0 \\ 0 & 0 & 6 \end{bmatrix}\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & -\frac{2}{\sqrt{6}} \end{bmatrix}$

♠ 6. $\begin{bmatrix} \frac{1}{2} & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{12}} \\ -\frac{1}{2} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{12}} \\ \frac{1}{2} & 0 & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{12}} \\ -\frac{1}{2} & 0 & 0 & \frac{3}{\sqrt{12}} \end{bmatrix}\begin{bmatrix} -4 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 8 \end{bmatrix}\begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{12}} & -\frac{1}{\sqrt{12}} & \frac{1}{\sqrt{12}} & \frac{3}{\sqrt{12}} \end{bmatrix}$

7. $3\begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} + 1\begin{bmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$

8. $18\begin{bmatrix} \frac{25}{26} & \frac{5}{26} \\ \frac{5}{26} & \frac{1}{26} \end{bmatrix} - 8\begin{bmatrix} \frac{1}{26} & -\frac{5}{26} \\ -\frac{5}{26} & \frac{25}{26} \end{bmatrix}$

9. $9\begin{bmatrix} \frac{1}{2} & \frac{3}{10} & \frac{2}{5} \\ \frac{3}{10} & \frac{9}{50} & \frac{12}{50} \\ \frac{2}{5} & \frac{12}{50} & \frac{8}{25} \end{bmatrix} + 4\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{16}{25} & -\frac{12}{25} \\ 0 & -\frac{12}{25} & \frac{9}{25} \end{bmatrix} - 1\begin{bmatrix} \frac{1}{2} & -\frac{3}{10} & -\frac{2}{5} \\ -\frac{3}{10} & \frac{9}{50} & \frac{12}{50} \\ -\frac{2}{5} & \frac{12}{50} & \frac{8}{25} \end{bmatrix}$

10. $2\begin{bmatrix} \frac{1}{2} & 0 & -\frac{1}{2} \\ 0 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} + (2+\sqrt{2})\begin{bmatrix} \frac{1}{4} & \frac{\sqrt{2}}{4} & \frac{1}{4} \\ \frac{\sqrt{2}}{4} & \frac{1}{2} & \frac{\sqrt{2}}{4} \\ \frac{1}{4} & \frac{\sqrt{2}}{4} & \frac{1}{4} \end{bmatrix} + (2-\sqrt{2})\begin{bmatrix} \frac{1}{4} & -\frac{\sqrt{2}}{4} & \frac{1}{4} \\ -\frac{\sqrt{2}}{4} & \frac{1}{2} & -\frac{\sqrt{2}}{4} \\ \frac{1}{4} & -\frac{\sqrt{2}}{4} & \frac{1}{4} \end{bmatrix}$

♠ 11. $30\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} - 12\begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & 0 \\ -\frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 \end{bmatrix} + 6\begin{bmatrix} \frac{1}{6} & \frac{1}{6} & -\frac{1}{3} \\ \frac{1}{6} & \frac{1}{6} & -\frac{1}{3} \\ -\frac{1}{3} & -\frac{1}{3} & \frac{2}{3} \end{bmatrix}$

♠ 12. $-4\begin{bmatrix} \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} \end{bmatrix} + 8\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + 8\begin{bmatrix} \frac{1}{6} & -\frac{1}{6} & -\frac{1}{3} & 0 \\ -\frac{1}{6} & \frac{1}{6} & \frac{1}{3} & 0 \\ -\frac{1}{3} & \frac{1}{3} & \frac{2}{3} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + 8\begin{bmatrix} \frac{1}{12} & -\frac{1}{12} & \frac{1}{12} & \frac{1}{4} \\ -\frac{1}{12} & \frac{1}{12} & -\frac{1}{12} & -\frac{1}{4} \\ \frac{1}{12} & -\frac{1}{12} & \frac{1}{12} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & \frac{3}{4} \end{bmatrix}$

$$= -4\begin{bmatrix} \frac{1}{4} & -\frac{1}{4}\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} \end{bmatrix} + 8\begin{bmatrix} \frac{3}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{3}{4} & \frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & \frac{1}{4} & \frac{3}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

5.4 Singular Value Decomposition

1. $\begin{bmatrix} 1 \end{bmatrix}\begin{bmatrix} 10 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} \frac{1}{10} & \frac{1}{10} & \frac{3}{10} & \frac{5}{10} & \frac{4}{5} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 \\ -\frac{3}{\sqrt{22}} & -\frac{3}{\sqrt{22}} & \sqrt{\frac{2}{11}} & 0 & 0 \\ -\frac{5}{6\sqrt{11}} & -\frac{5}{6\sqrt{11}} & -\frac{5}{2\sqrt{11}} & \frac{\sqrt{11}}{6} & 0 \\ -\frac{2}{15} & -\frac{2}{15} & -\frac{3}{5} & -\frac{2}{3} & \frac{3}{5} \end{bmatrix}$

♠ 2. $\begin{bmatrix} -1 \end{bmatrix}\begin{bmatrix} 5 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} -\frac{1}{5} & -\frac{2}{5} & -\frac{2}{5} & -\frac{4}{5} \\ -\frac{2}{5} & \frac{13}{15} & -\frac{2}{15} & -\frac{4}{15} \\ -\frac{2}{5} & -\frac{2}{15} & \frac{13}{15} & -\frac{2}{15} \\ -\frac{4}{5} & -\frac{4}{15} & -\frac{4}{15} & \frac{7}{15} \end{bmatrix}$

♠ 3. $\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

♠ 4. $\begin{bmatrix} \frac{7}{\sqrt{74}} & \frac{1}{\sqrt{26}} & \frac{12}{\sqrt{481}} \\ \frac{3}{\sqrt{74}} & \frac{3}{\sqrt{26}} & \frac{-16}{\sqrt{481}} \\ -\frac{4}{\sqrt{74}} & \frac{4}{\sqrt{26}} & \frac{9}{\sqrt{481}} \end{bmatrix}\begin{bmatrix} \sqrt{37} & 0 \\ 0 & \sqrt{13} \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

5. $\begin{bmatrix} \frac{1}{\sqrt{5}} & -\frac{2}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} \end{bmatrix}\begin{bmatrix} \sqrt{6} & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} \frac{1}{\sqrt{30}} & \frac{2}{\sqrt{30}} & \frac{5}{\sqrt{30}} \\ -\frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} & 0 \\ -\frac{1}{\sqrt{6}} & -\frac{2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \end{bmatrix}$ 6. $\begin{bmatrix} \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & 0 & \sqrt{\frac{2}{3}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} \end{bmatrix}\begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{6}} & \sqrt{\frac{2}{3}} & -\frac{1}{\sqrt{6}} \end{bmatrix}$

5.5 Dimension Reduction

1. $\begin{bmatrix} 1 & 1 & 3 & 5 & 8 \end{bmatrix}$

♠ 2. $\begin{bmatrix} 1 & 2 & 2 & 4 \end{bmatrix}$

♠ 3. $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

♠ 4. $\begin{bmatrix} \frac{7}{2} & \frac{7}{2} \\ \frac{3}{2} & \frac{3}{2} \\ -2 & -2 \end{bmatrix}$

5. $\begin{bmatrix} 1 & 2 & 5 \\ 2 & 4 & 10 \end{bmatrix}$

6. $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$