

Math for Programmers

3D graphics, machine learning,
and simulations with Python

Paul Orland



MEAP



MANNING



MEAP Edition
Manning Early Access Program
Math for Programmers
3D graphics, machine learning, and simulations with Python
Version 3

Copyright 2019 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP of *Math for Programmers*. I've been a math enthusiast my whole life, and only accidentally stumbled into software engineering as a career. I first taught myself how to code on a TI-84 graphing calculator, writing programs to do my high school math homework for me. Ever since then I've been excited by how complementary the disciplines of math and programming can be. I look forward to sharing what I've learned with you!

Now more than ever, knowing some math can accelerate your career as a developer. As evidence, look at the recent prevalence of the job title "Data Scientist," and what people with this title get paid. I joke with my coworkers about what a "Data Scientist" really is, but the best answer is probably someone who knows statistics, linear algebra, and calculus, and how to turn them into code. Beyond data analysis, these fields of math are useful in graphics, game design, simulation, optimization, and many other software development domains.

In this book, we'll start by exploring *vectors*, which are the mathematical tool for representing multidimensional data. Computer graphics in 2D and 3D are built with vectors, and you'll learn how to render your own 3D animations using *matrix transformations*. Part 1 culminates by showing you how these geometric lessons extend to higher dimensions within the framework of *linear algebra*.

Part 2 focuses on *calculus*, which is the study of continuous change. You'll learn that many of the laws of physics can be expressed in terms of calculus equations called *differential equations*. By solving these in Python, you can create realistic simulations of the physical world.

With a working knowledge of calculus and linear algebra, you'll be ready to learn some of the math behind *machine learning* in Part 3. Machine learning algorithms are often used to draw conclusions about vector data, and the "learning" is often accomplished using an operation from calculus called the *gradient*.

With a title as broad as *Math for Programmers*, there's more content than I'll be able to cover. I look forward to hearing your feedback on what I have included as well as what I haven't. Please post your questions and comments in the [forum](#), and I will take them seriously to make this book as useful as possible. Happy reading!

—Paul Orland

brief contents

1 Learning Math in Code

PART 1: VECTORS AND GRAPHICS

2 Drawing with 2D Vectors

3 Ascending to the 3D World

4 Transforming Vectors and Graphics

5 Computing Transformations with Matrices

6 Generalizing to Higher Dimensions

7 Solving Systems of Linear Equations

PART 2: CALCULUS AND PHYSICAL SIMULATION

8 Measuring motion with calculus

9 Working with symbolic expressions

10 Approximating functions

11 Modeling motion in 2D and 3D

12 Doing calculus with multi-dimensional functions

13 Solving differential equations

14 Building physical simulations

PART 3: MACHINE LEARNING APPLICATIONS

15 Quantifying uncertainty

16 Exploring and optimizing functions

17 Fitting functions to data

18 Classifying data

19 Training neural networks

20 Reducing dimensionality

Appendix: Loading and Rendering 3D Models with OpenGL and PyGame

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/math-for-programmers>

Licensed to Paul Olsztyn <polsztyn@gmail.com>

1

Learning Math in Code

This chapter covers

- Making money by implementing mathematical ideas in code
- Avoiding common pitfalls in math learning
- Thinking like a programmer to understand math
- Using Python as a powerful and extensible calculator

Math is like baseball or poetry or fine wine. Some people are so fascinated by math that they devote their whole lives to it, while others feel like they just don't "get it." You've probably already been forced into one camp or another by twelve years of compulsory math education in school.

What if we learned about fine wine in school like we learn math? I don't think I'd like wine at all if I got lectured on grape varietals and fermentation techniques for an hour a day. Maybe in such a world, I'd need to consume three or four glasses for homework, as assigned by the teacher. Sometimes this would be a delicious and educational experience, but sometimes I wouldn't feel like getting loaded on a school night. My experience in math class went something like this, and it turned me off of the subject for a while. Like wine, mathematics is an acquired taste, and a daily grind of lectures and assignments is no way to refine one's palate.

If you miss this, it's easy to think you're either "cut out" for math or you aren't. If you already believe in yourself and you're excited to start learning, that's great! Otherwise, this chapter is designed for you. Feeling intimidated by math is so common, it has a name: "math anxiety." I hope to dispel any anxiety you might have, and show you that math can be a

stimulating experience rather than a frightening one. All you need are the right tools and the right mindset.

The main tool for learning in this book is Python programming. I'm guessing when you learned math in high school, you saw it written on the blackboard and not written in computer code. This is a shame, because a high-level programming language is a far more powerful than a blackboard, and far more versatile than whatever overpriced graphing calculator you may have used. An advantage of meeting math in code is that the ideas have to be precise enough for a computer to understand, there's never any hand-waving about what new symbols mean.

As with learning any new subject, the best way to set yourself up for success is to *want* to learn it. There are plenty of good reasons. You could be intrigued by the beauty of mathematical concepts or enjoy the "brain-teaser" feel of math problems. Maybe there's an app or game you've been dreaming of building that needs some math to make it work. For now, I'll try to motivate you with an even lower common denominator: solving mathematical problems with software can make you filthy rich.

1.1 Solving lucrative problems with math and software

A classic criticism you hear in high school math class is "when am I ever going to use this stuff in real life?" Our teachers told us that math would help us succeed professionally and make money. I think they were right about this, even though their examples were off. For instance, I don't calculate my compounding bank interest by hand (and neither does my bank). Maybe if I became a construction site surveyor, as my trigonometry teacher suggested, I'd be using sines and cosines every day to earn my paycheck.

It turns out the "real world" applications from high school textbooks aren't that useful. Still, there are real applications of math out there, and some of them are mind-bogglingly lucrative. Many of them are solved by translating the right mathematical idea into usable software. I'll share some of my favorite examples.

1.1.1 Predicting financial market movements

We've all heard legends of stock traders making millions of dollars by buying and selling the right stocks at the right time. Based on the movies I've seen, I always pictured a trader as a middle-aged man in a suit yelling at his broker over a cell phone while driving around in a sports car. Maybe this stereotype was spot-on at one point, but the situation is different today. Holed up in back-offices of skyscrapers all over Manhattan are thousands of people called *quants*. Quants, otherwise known as quantitative analysts, design mathematical algorithms that automatically trade stocks and earn a profit. They don't wear suits and they don't spend any time yelling on their cell phones, but I'm sure many of them still own very nice sports cars.

So how does a quant write a program that automatically makes money? The best answers to this question are closely-guarded trade secrets, but you can be sure they involve a lot of math. We can look at a toy example to get a sense of how an automated trading strategy might work.

Stocks represents an ownership stakes in companies. When the market perceives a company is doing well, the price goes up: buying the stock becomes more costly and selling it becomes more rewarding. Stock prices change erratically and in real time. A graph of a stock price over a day of trading might look something like this.

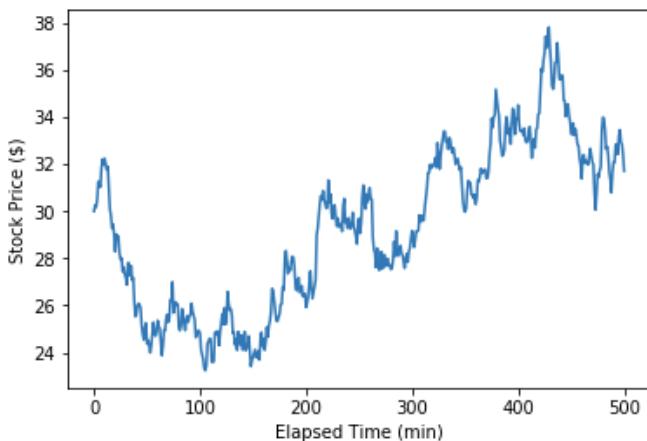


Figure 1.1 Typical graph of a stock price over time.

If you bought a thousand shares of this stock for \$24 around minute 100 and sold them for \$38 at minute 400, you would make off with \$14,000 for the day. Not bad! The challenge is that you'd have to know in advance that the stock was going up, and that minutes 100 and 400 were the best times to buy and sell, respectively. It may not be possible to predict the exact lowest highest price points, but maybe you can find relatively good times to buy and sell throughout the day. Let's look at a way to do this mathematically.

First we could measure whether the stock is going up or down by finding a line of "best fit", that approximately follows the direction the price is moving. This process is called *linear regression*, and we'll cover it in part 3 of the book. Based on the variability of data, we can calculate two more lines above and below the "best fit" line that show the region in which the price is wobbling up and down. Overlaid on the price graph, we see they follow the trend nicely.

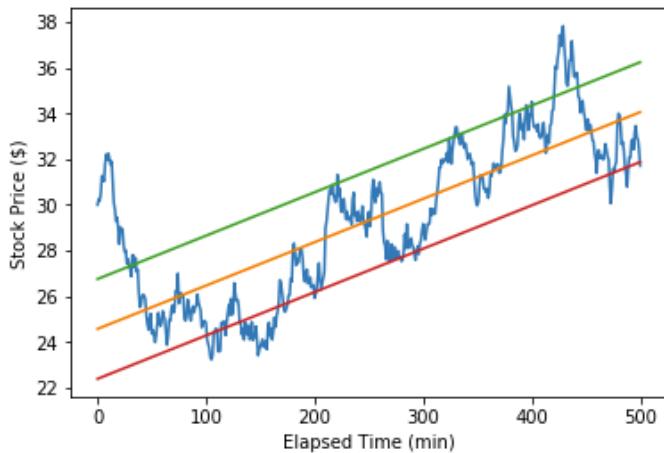


Figure 1.2 Using linear regression to identify the trend in a changing stock price.

With a mathematical understanding of the price movement, we could write software to automatically buy when the price is going through a low fluctuation and automatically sell when the price goes back up. Specifically, our program could connect to the stock exchange over the network and buy 100 shares whenever the price crosses the bottom line and sell 100 shares whenever the price crosses the top line. One profitable trade is shown below: entering at around \$27.80 and selling at around \$32.60 makes you \$480 in an hour.

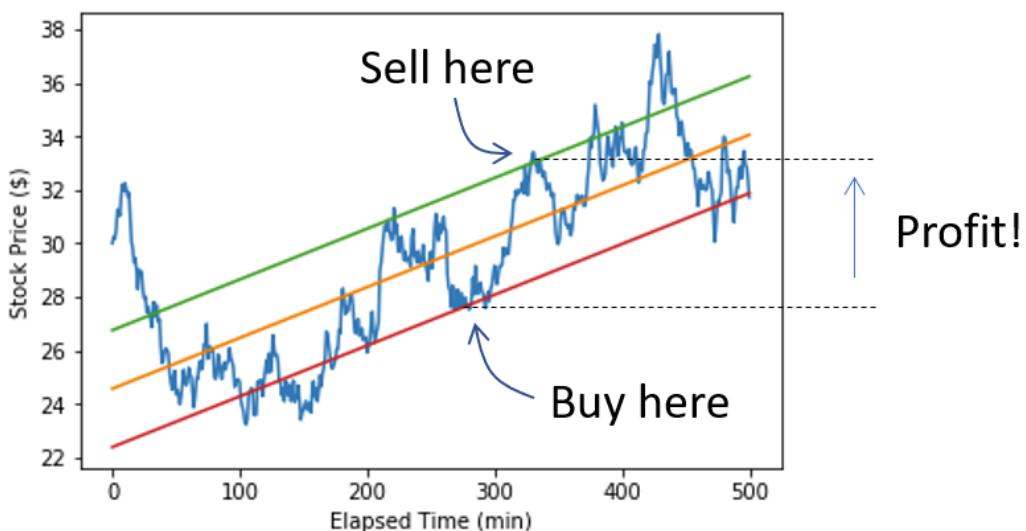


Figure 1.3 Buying and selling according to a rule to make a profit.

I'm sweeping a bunch of the complexities under the rug here, but the core idea works. At this very moment, some unknowable number of programs are building and updating models measuring the predicted trend of stocks and other financial instruments. If you write such a program, you can enjoy some leisure time while it makes money for you!

1.1.2 Finding a good deal

Maybe you don't have deep enough pockets to consider risky stock trading. Math can still help you make and save money in other transactions, like buying a used car. New cars are easy to understand commodities; if two dealers are selling the same car you obviously want to choose the cheaper of the two. Used cars have more numbers associated to them: an asking price as well as a mileage and a model year. You can even use the duration that a particular used car has been on the market to assess its quality -- the longer the duration, the more suspicious you might be.

In mathematics, objects you can describe with ordered lists of numbers are called *vectors*, and there is a whole field called *linear algebra* dedicated to studying them. A used car might correspond to a *four-dimensional* vector, meaning a four-tuple of numbers:

(2015, 41429, 22.27, 16980).

These numbers are the model year, mileage, days on the market, and asking price. A friend of mine runs a site called CarGraph.com that aggregates data on used cars for sale. At the time of writing, it shows 101 Toyota Priuses for sale, and it gives some or all of these four pieces of data for each one. The site also lives up to its name and visually presents the data in a graph. It's hard to visualize four-dimensional objects, but if you choose two of the dimensions like price and mileage, you can plot them as points on a scatter plot.

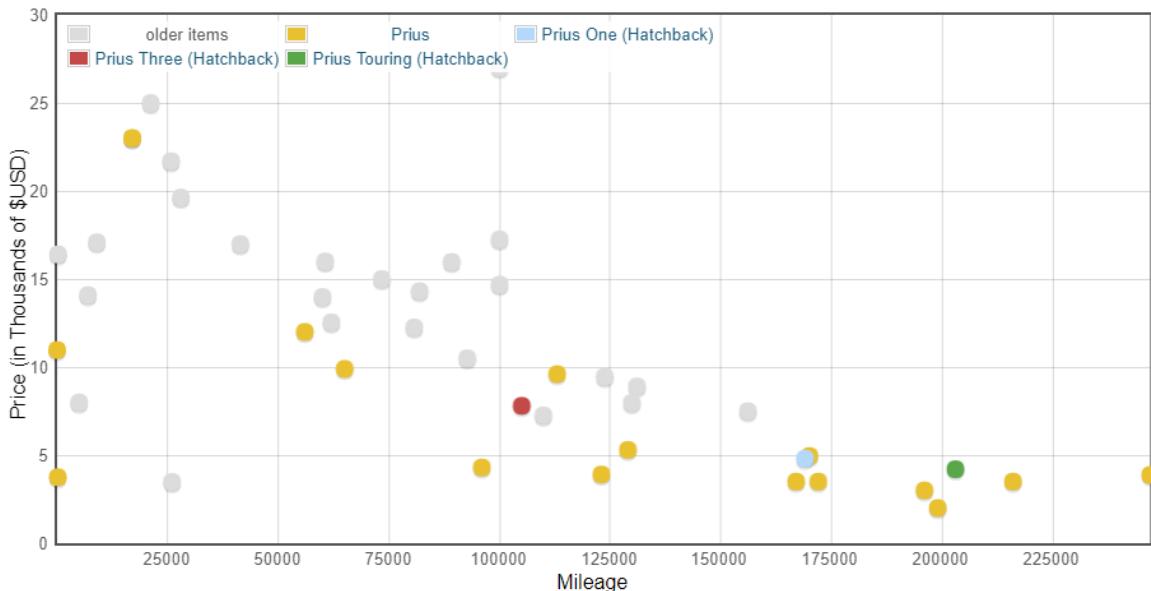


Figure 1.4 A graph of price vs. mileage for used priuses from CarGraph.com

We might be interested in drawing a trend line here too -- every point on this graph represents someone's opinion of a fair price, so the trend line would aggregate these opinions together into a more reliable price at any mileage. In this case, I decided to fit to an *exponential* decline curve rather than a line, and I omitted some of the nearly-new cars selling for below retail price.

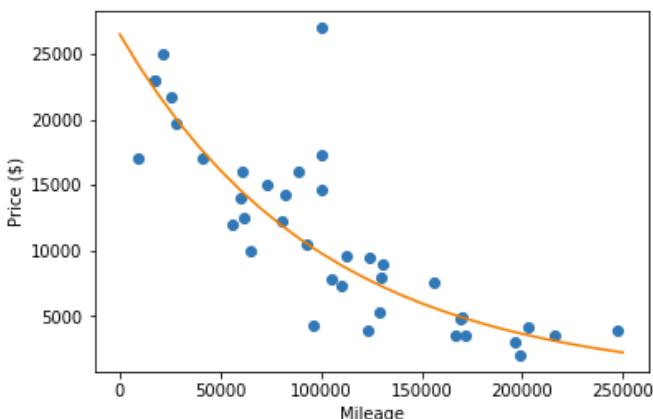


Figure 1.5 Fitting an exponential decline curve to price vs. mileage data

The equation for the curve of best fit is

$$\text{price} = \$26,500 \cdot (0.99999017^{\text{mileage}})$$

Figure 1.6 An equation for the best fit curve.

That is, the best fit price is \$26,500 times 0.99999017 raised to the power of the mileage. Plugging values in to the equation, I find that if my budget is \$10,000, then I should suspect to be buying a Prius with about 97,000 miles on it. If I believe the curve indicates a *fair* price, then cars below the line should typically be good deals.

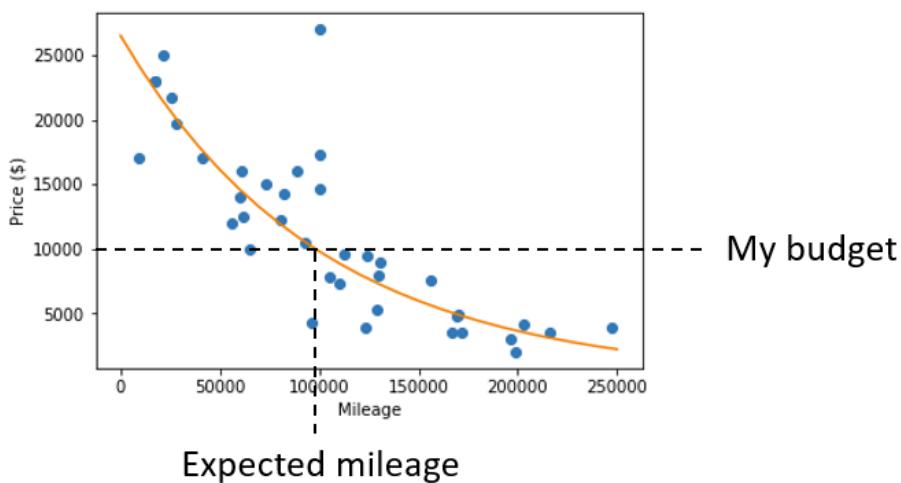


Figure 1.7 Finding the mileage I should expect on a used Prius for my \$10,000 budget.

But we can learn more from this equation than how to find a good deal -- it tells a story about how cars depreciate. The first number in the equation is \$26,500, which is the exponential function's understanding of the price at zero mileage. This is an impressively close match to the retail price of a new Prius. If we used a line of best fit, it would have implied a Prius loses a fixed amount of value with each mile driven. This exponential function says instead that it loses a fixed *percentage* of its value with each mile. After driving one mile, a Prius is only worth 0.99999017 or 99.999017% of its original price, according to this equation. After 50,000 miles, its price is multiplied by a factor of $(0.99999017)^{50,000} = .612$. That tells us that it's worth about 61% of what it was originally.

To make the graph above, I implemented the `price(mileage)` function in Python. Calculating `price(0) - price(50000)` and `price(50000) - price(100000)` tell me that the first and second 50,000 miles driven cost about \$10,000 and \$6,300 respectively. If we had instead used a *line* of best fit, it would have implied that the car depreciated at fixed rate of \$0.10 per mile. That would imply that every 50,000 miles have the same fixed cost of \$5000.

Conventional wisdom says that the first miles you drive on a new car are the most expensive, and only the exponential function agrees with this.

Remember, this is only a *two-dimensional* analysis. We only built a mathematical model to relate two of the four numerical dimensions describing each car. In part 1 we'll learn more about vectors of various dimensions, and learn how to manipulate higher-dimensional data. Different kinds of functions like linear functions and exponential functions will be covered in part 2, and we'll compare them by their different rates of change. Finally in part 3 we'll look at how to build mathematical models that incorporate *all* the dimensions of a data set to give us an accurate picture.

1.1.3 Building 3D graphics and animations

Many of the most famous and financially successful software projects in the world have dealt with multi-dimensional data, specifically *three-dimensional* or *3D* data. Here I'm thinking of 3D animated movies and 3D video games which gross in the billions of dollars. Pixar's 3D animation software has helped them rake in over \$13 billion at box offices. Activision's *Call of Duty* franchise of 3D action games has earned over \$16 billion, and Rockstar's *Grand Theft Auto V* alone brought in \$6 billion.

Every one of these acclaimed projects is based on an understanding of how to do computations with 3D vectors, or triples of numbers of the form (x,y,z) . A triple of numbers is sufficient to locate a point in 3D space relative to a reference point called the origin. Each of the three numbers tells you how far to go in one of three perpendicular directions:

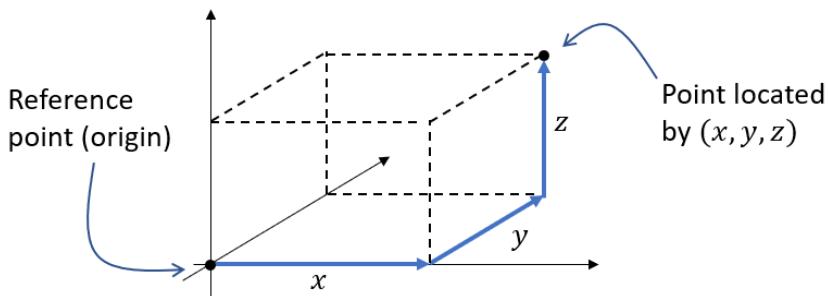


Figure 1.8 Labeling a point in 3D with a vector of three numbers: x , y , and z .

Any 3D object, from a clownfish in *Finding Nemo* to an aircraft carrier in *Call of Duty*, can be defined for a computer as a collection of 3D vectors that make it up. In code, each of these objects looks like a list of triples of `float` values. With three triples of floats, we have three points in space which can define a triangle:

```
triangle = [(2.3,1.1,0.9), (4.5,3.3,2.0), (1.0,3.5,3.9)]
```

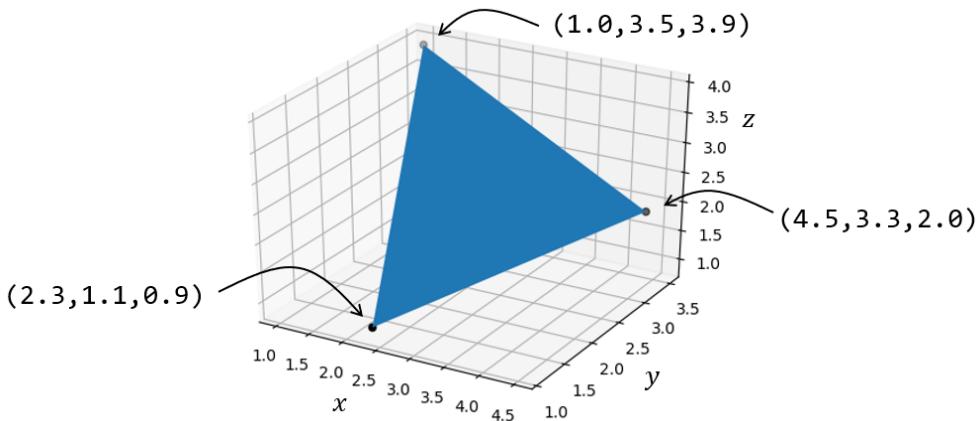


Figure 1.9 Building a triangle in 3D using a triple of float values for each of its corners.

Combining many triangles, you can define the surface of a 3D object. Using more, smaller triangles, you can even make the result look smooth. Here are six renderings of a 3D sphere, using an increasing number of smaller and smaller triangles.

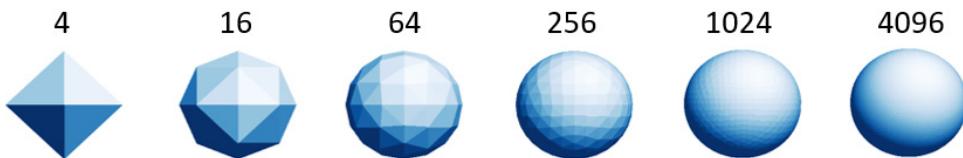


Figure 1.10 3D spheres built out of the specified number of triangles:

In chapters 3 and 4, you'll learn how to use 3D vector math to turn 3D models into shaded 2D images like the ones on this page. You need to make your 3D models smooth to make them realistic in a game or movie, and you also need them to move and change in realistic ways. This means that your objects should obey the laws of physics, which are also expressed in terms of 3D vectors.

Suppose you were a programmer on *Grand Theft Auto V* and wanted to enable some basic use case like shooting a bazooka at a helicopter. A projectile coming out of a bazooka starts at the protagonist's location and then its position changes over time. We can use numeric subscripts to label the various positions it has over its flight, starting with (x_0, y_0, z_0) . As time elapses, the projectile arrives at a new positions labeled by vectors (x_1, y_1, z_1) , (x_2, y_2, z_2) , and so on. The rates of change for the x, y, and z values are decided by the direction and speed of the bazooka. Moreover, the rates can change over time -- the projectile should increase its z position at a decreasing rate because of the continuous downward pull of gravity.

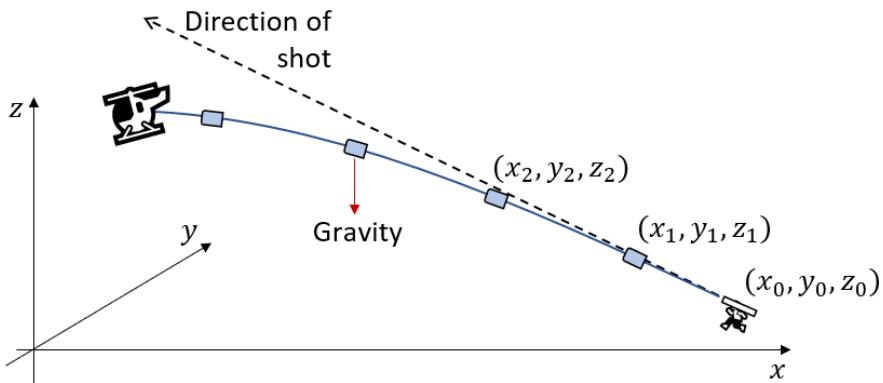


Figure 1.11 The position vector of the projectile changes over time due to its initial speed and the pull of gravity.

As any experienced action gamer will tell you, you need to aim slightly above the helicopter to hit it! To simulate physics, you have to know how forces affect objects and cause continuous change over time. The math of continuous change is called *calculus* and the laws of physics are usually expressed in terms of objects from calculus called *differential equations*. You'll learn how to animate 3D objects in chapters 4 and 5, and then how to do so using ideas from calculus in part 2.

1.1.4 Modeling the physical world

My claim that mathematical software can produce real financial value isn't just speculation, I've seen the value in my own career. In 2013, I founded a company called Tachyus that builds software to optimize oil and gas production. Our software uses mathematical models to understand the flow of oil and gas underground to help producers extract it more efficiently and profitably. Using the insights it generates, our customers have achieved millions of dollars a year in cost savings and production increases. Let me show you how it works:

The setup for conventional, onshore oil production usually looks something like this. Holes called *wells* are drilled into the ground until they reach the target layer of porous (sponge-like) rock containing oil. This layer of oil-rich rock underground is called a *reservoir*. Oil is pumped to the surface and is then sold to refiners who produce products we use every day.

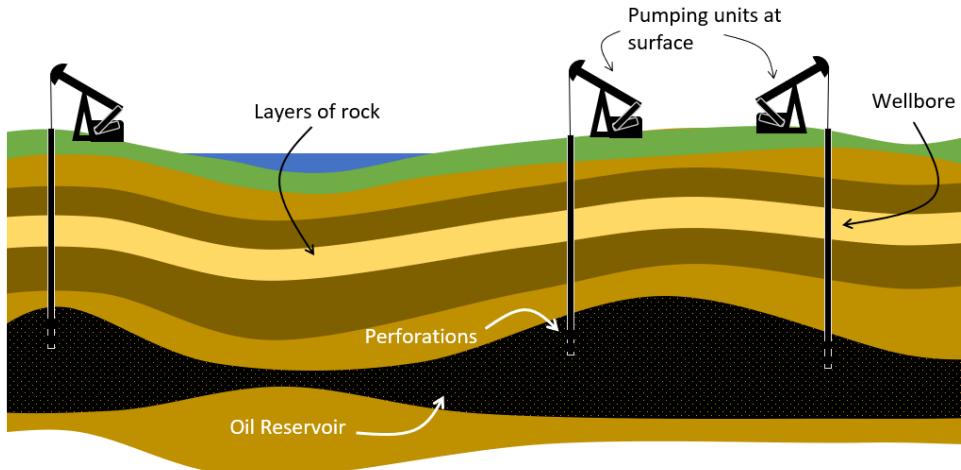


Figure 1.12 A schematic diagram of an oilfield.

Over the past few years, the price of oil has varied from about \$25 per barrel to over \$100 per barrel, where a barrel is a unit of volume equal to 42 gallons or about 159 liters. If, by drilling wells and pumping effectively, a company is able to extract 1000 barrels of oil per day (the volume of a few backyard swimming pools), they will have annual revenues in the tens of millions of dollars. Even a few percentage points of increased efficiency can mean a big amount of money.

The big underlying question is what is going on underground: where is the oil now and how is it moving? This is a very complicated question, but it can also be answered by solving differential equations. The changing quantities here are not positions of a projectile, but rather locations, pressures, and flow-rates of fluids underground. Fluid flow-rate is a special kind of vector valued function called a *vector field*, meaning fluid can be flowing in any rate in any three-dimensional direction *and* that direction and rate may vary across different locations within the reservoir.

With our best guess for some of these parameters, we can use a differential equation called *Darcy's law* to predict flow rate of liquid through a porous rock medium like sandstone. Here's what Darcy's law looks like; don't worry if some symbols are unfamiliar!

$$q = -\frac{\kappa}{\mu} \nabla p$$

Flow rate of fluid

Permeability of porous medium

Pressure gradient

Viscosity (thickness) of fluid

Figure 1.13 Darcy's law (annotated), a physics equation governing how fluid flows within a porous rock.

The most important part of this equation is the upside down triangle symbol which represents the *gradient* operator in vector calculus. The gradient of the pressure p is the 3D vector indicating the direction of increasing pressure and the size of pressure increase. The negative sign tells us that the 3D vector of flow rate is in the *opposite* direction. This equation states, in mathematical terms, that fluid flows from areas of high pressure to areas of low pressure.

Negative gradients are common in laws of physics. One way to think of this is that nature is always seeking to move toward lower potential energy states. The potential energy of a ball on a hill depends on the altitude h of the hill at any lateral point x . If the height of a hill is given by a function $h(x)$, the gradient would point uphill while the ball would roll in the exact opposite direction.

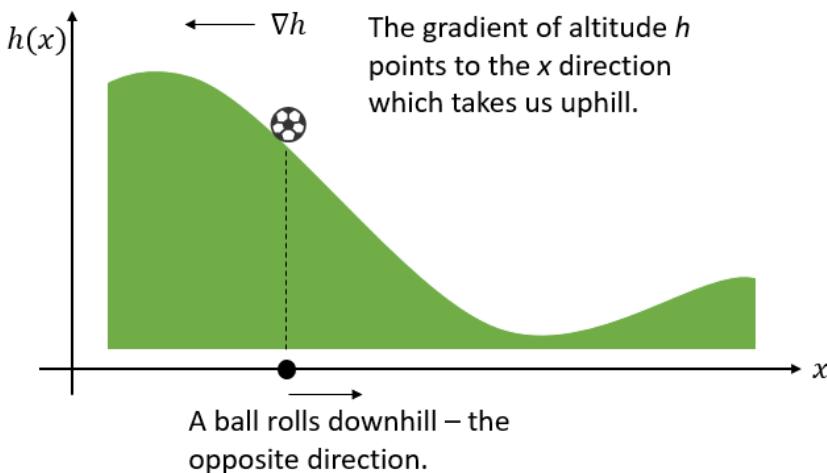


Figure 1.14 The gradient points uphill, while the negative gradient points downhill.

In chapter 8, you'll learn how to calculate gradients. I'll show you how to apply them to simulate physics, and also to solve other mathematical problems; the gradient happens to be one of the most important mathematical concepts in machine learning as well.

I hope these examples have been more compelling and realistic than the "real world" applications you're used to hearing in high school math class. Maybe at this point you're convinced these math concepts are worth learning, but you're worried they'll be too difficult. It's true: learning math can be hard, especially on your own. To make it as smooth as possible, let's talk about some of the pitfalls you can face as a math student, and how I'll help you avoid them in this book.

1.2 How not to learn math

There are plenty of math books out there, but not all of them are equally useful. I have quite a few programmer friends who have tried to learn math concepts like the ones in the previous section, either motivated by intellectual curiosity or career ambitions. When they use traditional math textbooks as their main resource, they often get stuck and give up. Here's what a typical *unsuccessful* math learning story looks like.

1.2.1 Jane wants to learn some math

My (fictional) friend Jane is a full-stack web developer working at a medium size tech company in San Francisco. In college, Jane didn't study computer science or any mathematical subjects in depth, and she started her career as a product manager. Over the last ten years, she picked up coding in Python and JavaScript and was able to transition into software engineering. Now, at her new job, she is one of the most capable programmers on the team, able to build

databases, web services, and user interfaces required to deliver important new features to customers. Clearly she is pretty smart!

Jane realizes that learning data science could help her design and implement better features at work, using data to improve the experience for her customers. Most days on the train to work, Jane reads blogs and articles about new technologies, and recently she's been amazed by a few about a topic called "deep learning". One article talked about Google's AlphaGo, powered by deep learning, which was able to beat the top human players in the world in a board game.

Another article showed stunning impressionist paintings generated from ordinary images, again using a deep learning system. After reading these articles, Jane overheard that her friend-of-a-friend Marcus got a deep learning research job at a "big five" tech company. Marcus is supposedly getting paid over \$400,000 a year in cash and stock. Thinking about the next step in her career, what more could she want than to work on a fascinating and lucrative problem?

Jane did some research and found an authoritative (and free!) resource online: the book *Deep Learning* by Goodfellow et al. The introduction read much like the technical blog posts she was used to, and got her even more excited about learning the topic. But as she kept reading, the content got harder. The first chapter covered the required math concepts and introduced a lot of terminology and notation that Jane had never seen. She skimmed it and tried to get on to the meat of the book, but it got harder and harder.

Jane decided she needed to pause her study of AI until she learned some math. Fortunately the math chapter of *Deep Learning* listed a reference on "linear algebra" for students who had never seen the topic before. She tracked down this textbook-- *Linear Algebra* by Georgi Shilov --and discovered that it was 400 pages long and equally as dense as *Deep Learning*.

After spending an afternoon reading abstruse theorems about concepts like "number fields", "determinants", and "cofactors", she called it quits. She had no idea how these concepts were going to help her write a program to win a board game or generate artwork, and she no longer cared to spend dozens of hours with this dry material to find out.

Jane and I met to catch up over a cup of coffee. She told me about her struggles reading real AI literature because she didn't know linear algebra. Recently, I'm hearing a lot of the same form of lamentation:

I'm trying to read about [new technology] but it seems like I need to learn [math topic] first.

Her approach was admirable: she tracked down the best resource for the subject she wanted to learn and sought out resources for prerequisites she was missing. But in taking that approach to its logical conclusion, she found herself in a nauseating "depth-first" search of technical literature.

1.2.2 Slogging through math textbooks

College-level math books like the linear algebra book Jane picked up tend to be very formulaic. Every section follows the same recipe:

1. A *definition* for a new term is given
2. One or more *propositions* are given, which are facts you can deduce from the definition
3. A major *theorem* is stated, which is like a proposition but more important
4. A formal *proof* is given for the theorem, which is a rigorous argument that the theorem must be true.
5. Finally, *corollaries*, which are applications of the theorem, are stated and proved.

This sounds like a good, logical order -- you introduce what concept you're talking about, state some conclusions that can be drawn, and then defend them. Then why is it so hard to read advanced math textbooks?

The problem is that this is not how math is actually *created*. When you're coming up with new mathematical ideas, there can be a long period of playing around with ideas before you even find the right definitions. I think most professional mathematicians would describe their steps like this:

1. First, invent a *game*: start playing with some mathematical objects by trying to list them all, find patterns among them, or find one with a particular property.
2. Form some *conjectures*. Speculate about some general facts you can state about your game, and at least convince yourself they must be true.
3. Develop some *precise language* to describe your game and your conjectures. Your conjectures won't mean anything until you can communicate them.
4. Finally, with some determination and luck, find a *proof* for your conjecture, showing why it *needs* to be true.

The main lesson to learn from this process is that you should start by thinking about big ideas, and the formalism can wait for later. Once you have a rough idea how the math works, the vocabulary and notation will be an asset for you rather than a distraction. Math textbooks usually work in the opposite order, so I recommend them as references rather than introductions to new subjects.

Instead of reading traditional textbooks, the best way to learn math is to explore ideas and draw your own conclusions. However, you don't have enough hours in the day to reinvent everything yourself. What is the right balance to strike? I'll give you my humble opinion, which guides how I've written this non-traditional book.

1.3 Using your well-trained left brain

This book is designed for people who are either experienced programmers, or who are excited to learn programming as they work through it. It's great to write for an audience of programmers, because if you can write code you've already trained your analytical "left brain."

I think the best way to learn math is with the help of a high-level programming language, and I predict that in the not-so-distant future this will be the norm in math classrooms.

There are several specific ways programmers like you are well equipped to learn math. I will list them here not only to flatter you, but also to remind you what skills you already have that you can lean on in your math studies.

1.3.1 Using a formal language

One of the first hard lessons you learn in programming is that you can't write your code like you write simple English. If your spelling or grammar is slightly off when writing an email, the recipient will probably still know what you were talking about. But any syntactic error or misspelled identifier will cause your program to fail. In some languages, even forgetting a semicolon at the end of an otherwise correct statement will prevent the program from running.

One example simple example is variable assignment. To a non-programmer, these two lines of Python seem to say the same thing:

```
x = 5
```

```
5 = x
```

I could read either of these to mean that the symbol `x` has value 5. But that's not *exactly* what either of these means, and in fact only the first one is correct. The python statement `x = 5` is an instruction to *bind* the value 5 to the symbol `x`. On the other hand you can't bind a symbol to the literal value 5. This may seem pedantic, but you need to know it to write a correct program.

Another example which trips up novice programmers (and experienced ones as well!) is reference equality.

```
>>> class A(): pass
...
>>> A() == A()
False
```

If you define a new Python class and create two identical instances of it, they are not equal! You might expect two identical expressions to be equal, but that's evidently not a rule in Python. Because these are different instances of the `A` class, they are not considered equal.

Be on the lookout for new mathematical objects that look like ones you know but don't behave the same way. For instance, if the letters `A` and `B` represent numbers then $A \cdot B = B \cdot A$. But, as you'll learn in chapter 5, this is not necessarily the case if `A` and `B` are *not* numbers. If instead `A` and `B` are matrices, then the products $A \cdot B$ and $B \cdot A$ are usually different. In fact it's possible that only one of the products is even possible, or that neither product is possible.

When you're writing code, it's not enough to write statements with correct syntax. The ideas that your statements represent need to make sense to be valid. If you apply the same care when you're writing down mathematical statements, you'll catch your mistakes faster. Even better, if you write your mathematical statements in code, you'll have the computer to help check your work.

1.3.2 Build your own calculator

Calculators are prevalent in math classes because it's useful to check your work. You need to know how to multiply 6 by 7 without using your calculator, but it's good to confirm that your answer of 42 is correct by consulting with your calculator. The calculator also helps you save time once you've mastered concepts. If you're doing trigonometry and you need to know $3.14159 / 6$, the calculator is there to handle it so you can think about what the answer means. The more a calculator can do out-of-the-box, the more useful it should theoretically be.

But sometimes our calculators are too complicated for our own good. When I started high school, I was required to get a graphing calculator and I got a TI-84. It had about 40 buttons, each with 2-3 different modes. I only knew how to use maybe 20 of them, so it was a cumbersome tool to learn how to use. The story was the same in first grade, when even the simplest calculator you could buy had buttons I didn't understand yet. If I had to invent a first calculator for students, I would make it look something like this:



Figure 1.15 A calculator for students learning to count.

This calculator only has two buttons. One of them resets the value to 1 and the other advances to the next number. Something like this would be the right "no-frills" tool for kids learning to count. (My example may seem silly but you can actually buy calculators like this! They are usually mechanical, and sold as "tally counters.")

Soon after you master counting, you want to practice writing numbers and adding them. The perfect calculator at that stage of learning might have a few more buttons:



Figure 1.16 A calculator capable of writing whole numbers and adding them.

There's no need for a “-”, “ \times ”, or “ \div ” to get in your way at this phase. As you solve subtraction problems like “5 - 2”, you can still check your answer of 3 with this calculator by confirming the sum “ $3 + 2 = 5$.“ Likewise you can solve multiplication problems by adding numbers repeatedly. You could upgrade to a calculator that does all of the operations of arithmetic when you're done exploring with this one.

In theory, it would be great to add buttons to our calculator as soon as we are ready for them, but that would lead to a lot of hardware floating around. We'd have to solve another problem as well: our calculators would have to hold more types of data than numbers. In algebra you solve equations with symbols alone, and in trigonometry and calculus you often manipulate functions to create new ones.

Extensible calculators that can hold many types of data seem far-fetched, but that's exactly what you get when you use a high-level programming language. Python comes with arithmetic, a `math` module, and numerous third-party mathematical libraries you can pull in to make your programming environment more powerful, whenever you want. Since Python is *Turing complete*, you can (in principle) compute anything that can be computed. You only need a powerful enough computer, a clever enough implementation, or both.

In this book, we'll implement each new mathematical concept we see in reusable Python code. Working through the implementation yourself can be a great way of cementing your understanding of a new concept, and by the end you've added a new tool to your toolbelt. After trying it yourself, you can always swap in a polished, mainstream library if you like. Either way, the new tools you build or import will lay the groundwork to explore even bigger ideas.

1.3.3 Building abstractions with functions

In programming, the process I describe above is called “abstraction.” When you get tired of repeated counting, you create the abstraction of addition. When you get tired of doing so much repeated addition, you create the abstraction of multiplication, and so on.

Of all the ways you can make abstractions in programming, the most important one to carry over to math is the *function*. A function in Python is a way of repeating some task that may take one or more inputs or produce an output. For example:

```
def greet(name):
    print("Hello %s!" % name)
```

This lets me issue multiple greetings with short, expressive code:

```
>>> for name in ["John", "Paul", "George", "Ringo"]:
...     greet(name)
...
Hello John!
Hello Paul!
Hello George!
Hello Ringo!
```

This function may be useful, but it’s not like a mathematical function. Mathematical functions always take input values and they always return output values, with no side effects. In programming, the functions that behave like mathematical functions are called *pure functions*. For example, the square function $f(x)=x^2$ takes a number in and returns the product of the number with itself. When you evaluate $f(3)$ the result is 9. That doesn’t mean that the number 3 has now changed and become 9. Rather, it means 9 is the corresponding output for the input 3 for the function f . You can picture this squaring function as a machine that takes numbers in an input slot and produces result numbers from its output slot.

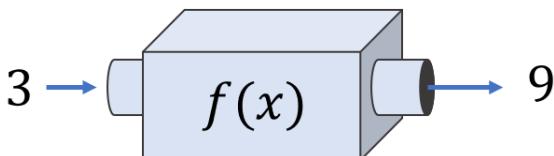


Figure 1.17 A function as a machine with an input slot and an output slot.

This is a simple and useful mental model, and I’ll return to it throughout the book. One of the things I like most about it is that you picture a function as an object in and of itself. In math, as in Python, functions are data that you can manipulate independently and even pass to other functions.

Math can be intimidating because it is abstract. Remember, as in any well-written software, the abstraction is introduced for a reason: it helps you communicate bigger and more powerful ideas. When you grasp these ideas and translate them into code, you’ll open up some exciting possibilities.

1.4 Summary

In this chapter, you learned that

- There are interesting and lucrative applications of math in multiple software engineering domains.
- Math can help you quantify the trend for data that changes over time, for example to predict the movement of a stock price.
- Different types of functions convey different kinds of qualitative behavior. For instance, an *exponential* depreciation function means that a car loses a percentage of its resale value with each mile driven, rather than a fixed amount.
- Tuples of numbers called *vectors* are used to represent multidimensional data. Specifically, 3D vectors are triples of numbers and can represent points in space. You can build complex 3D graphics by assembling triangles specified by vectors.
- *Calculus* is the mathematical study of continuous change, and many of the laws of physics are written in terms of calculus equations called *differential equations*.
- It's hard to learn math from traditional textbooks! Math is learned by exploration, not as a straightforward march through definitions and theorems.
- As a programmer, you've already trained yourself to think and communicate precisely - this skill will help you learn math as well.
- Python is like an extensible calculator. Writing new functions in Python is like adding new buttons to your pocket calculator.

If you didn't already, I hope you now believe there are many exciting applications of math in software development. As a programmer, you already have the right mindset and tools to learn some new mathematical ideas. The ideas in this book provided me with professional and personal enrichment, and I hope they will for you as well. Let's get started!

2

Drawing with 2D Vectors

This chapter covers

- Creating and manipulating 2D drawings as collections of vectors.
- Thinking of 2D vectors as arrows, locations, and ordered pairs of coordinates.
- Using vector arithmetic to transform shapes in the plane.
- Using trigonometry to measure distances and angles in the plane.

You've probably already got some intuition for what it means to be "two-dimensional" or "three-dimensional." A *two-dimensional* (2D) object is flat, like an image on a piece of paper or a computer screen. It has only the dimensions of height and width. Our physical world is *three-dimensional*: real objects have not only height and width but also depth.

Models of 2D and 3D entities are important in programming. Anything that shows up on the screen of your phone, tablet, or PC is a two-dimensional object, occupying some width and height of pixels. Any simulation, game, or animation that represents the physical world is stored as three-dimensional data, and eventually projected to the two dimensions of the screen. In virtual and augmented reality applications, the 3D data of the model must be paired with real, measured 3D data about the user's position and perspective.

Even though our everyday experience takes place in three dimensions, it's useful to think of some data as higher dimensional. In physics it's common to consider time as the fourth dimension. While an object exists at a location in three-dimensional space, an event occurs at a three-dimensional location and a specified moment. In data science problems, it's common for data sets to have far more dimensions. A user tracked on a website may have hundreds of measurable attributes that could be used to predict their usage patterns. Grappling with these problems in graphics, physics, and data analysis requires a framework for dealing with data in higher dimensions. This framework is vector mathematics.

Vectors are objects that live in multi-dimensional spaces. They have their own notions of arithmetic (adding, multiplying, and so on) which are studied in the branch of math called linear algebra. We'll start by studying 2D vectors, which are easy to visualize and compute with. We will use a lot of 2D vectors in this book, and we will also use them as a mental model when reasoning about higher dimensional problems.

2.1 Drawing with 2D Vectors

The two-dimensional world is flat, like a piece of paper or a computer screen. Flat spaces like these are called *planes* in the language of mathematics. An object living in a 2D plane has the two dimensions of height and width but no third dimension of depth. Likewise, locations in 2D can be described by two pieces of information: their vertical and horizontal positions. To describe the location of points in the plane, you need a reference point. We call that special reference point the *origin*.

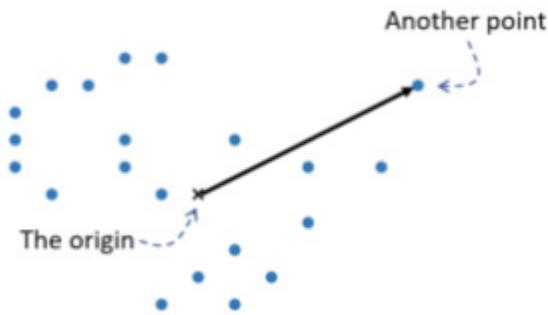


Figure 2.1: Locating one of several points in the plane, relative to the origin.

There are many points to choose from in this picture, but only one is selected to be the origin. To distinguish it, we mark it with an "x" instead of with a dot. From the origin, we can draw an arrow (like the solid one above) to show the relative location of another point.

A *two-dimensional vector* is a point in the plane, relative to the origin. Equivalently you can think of a vector as a straight arrow in the plane: any arrow can be placed to start at the origin, and it will indicate a particular point.

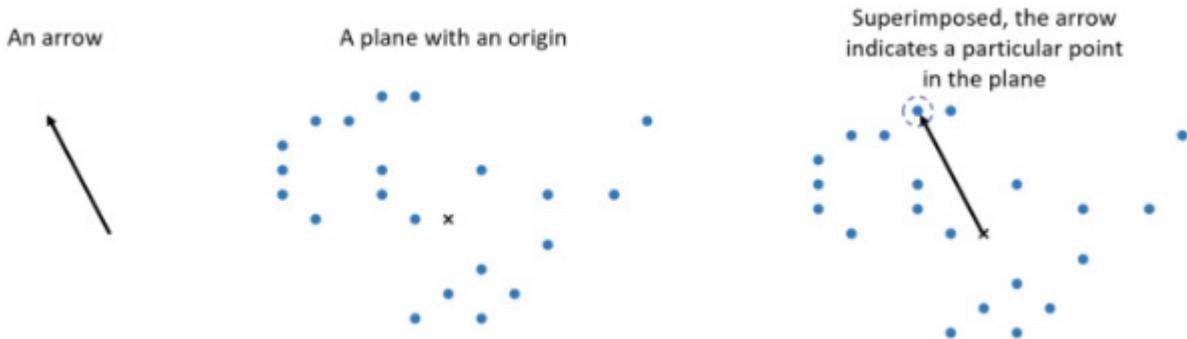


Figure 2.2 Superimposing an arrow on the plane indicates a point relative to the origin.

We'll use both arrows and points to represent vectors in this chapter and beyond. Points are useful to work with because we can build more interesting drawings out of them. If I connect the points above, I get a drawing of a dinosaur:

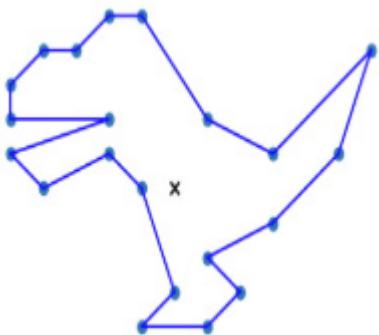


Figure 2.3 Connecting points in the plane to draw a shape.

Any time a 2D or 3D drawing is displayed by a computer, from my modest dinosaur to a feature-length Pixar movie, it is defined by points -- or vectors -- connected to show the desired shape. To create the drawing you want, you need to pick vectors in the right places, requiring careful measurement. Let's take a look at how to measure vectors in the plane.

2.1.1 Representing 2D vectors

With a ruler we can measure one dimension, like the length of an object. To measure in two dimensions, we'll need two rulers. These are called axes (the singular is *axis*), and we lay them out in the plane perpendicular to one another, intersecting at the origin. Drawn with axes, our dinosaur has notions of up and down as well as left and right. The horizontal axis is called the *x-axis* and the vertical one is called the *y-axis*.

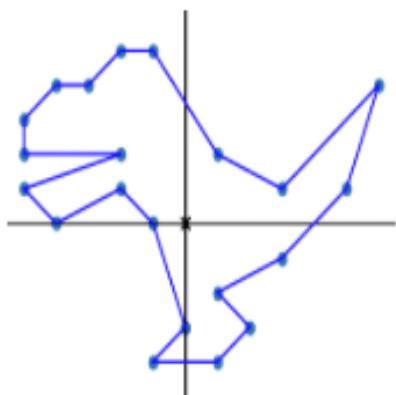


Figure 2.4 The dinosaur drawn with an x-axis and a y-axis.

With axes to orient us, we can say things like “four of the points are above and to the right of the origin.” But we’ll want to get more quantitative than that. A ruler has “ticks” that show how many units along it we’ve measured. Likewise, we can add grid lines perpendicular to the axes that show where points lie relative to them. By convention, we place the origin at tick “0” on both the x and y axis.

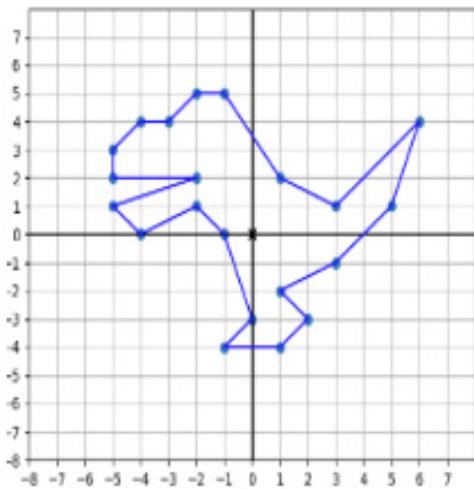


Figure 2.5 Grid lines let us measure the location of points relative to the axes.

In the context of this grid, we can measure vectors in the plane. The tip of the dinosaur’s tail lines up with positive six on the x-axis and positive four on the y-axis. These could be centimeters, inches, pixels, or any other unit of length, but we will leave them unspecified. The numbers six and four are called the *x* and *y* *coordinates* of the point, and they are enough

to tell us exactly what point we are talking about. We typically write coordinates as an *ordered pair* (or *tuple*), for example (6,4), with the x coordinate first and the y coordinate second.

Now we can describe the same vector three ways, pictured below:

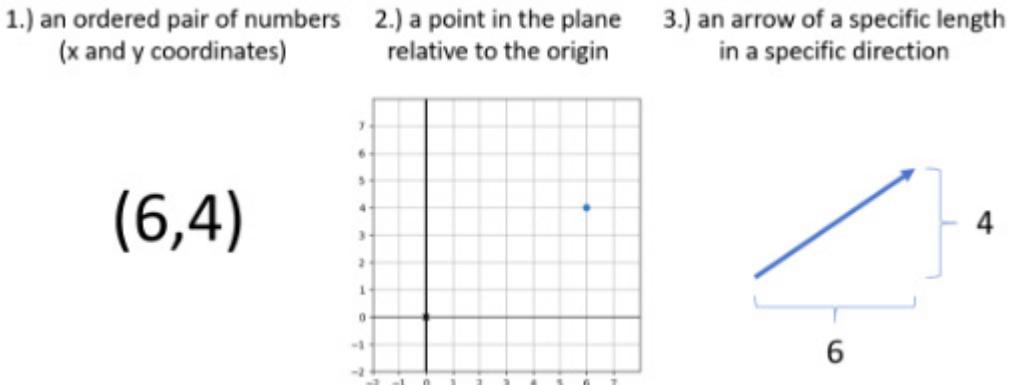


Figure 2.6 Three mental models describing the same vector.

From another pair of coordinates, like (-3,4.5) we could find the point in the plane or the arrow that represents them. To get to the point in the plane, travel three grid lines to the left (since the x coordinate is negative three) and then four and a half grid lines up. The point doesn't lie at the intersection of two grid lines, but that's fine -- any pair of real numbers will give us some point on the plane. The corresponding arrow is the "straight-line" path from the origin to that location, which points up and to the left (northwest if you prefer).

2.1.2 2D Drawing in Python

Whenever you're trying to produce an image on a screen, you're working in a two-dimensional space. The pixels on the screen are the available points in that plane, and they are labeled by whole number coordinates rather than real number coordinates: you can't illuminate the space between pixels. That said, most graphics libraries let you work with floating point coordinates and handle translating graphics to pixels on the screen automatically.

We have plenty of choices of languages and libraries to specify graphics and get them on the screen: OpenGL, CSS, SVG, and so on. Python has libraries like Pillow and Turtle that are well equipped for creating drawings with vector data. In this chapter, I'm going to use a small set of custom-built functions to create drawings, built on top of another Python library called Matplotlib. This will let us focus on using Python to build images with vector data -- once you understand this process, you'll be able to pick up any of the other libraries easily.

The most wrapper function I built is a "draw" function, which takes inputs representing geometric objects, as well as keyword arguments to specify how you want your drawing to

look. Each kind of drawable geometric object is represented by one of the Python classes listed below.

Table2.1 Some classes I invented for drawing with my draw function.

Class	Constructor example	Description
Polygon	Polygon(*vectors)	A polygon whose vertices (corners) are represented by a list of vectors.
Points	Points(*vectors)	Represents a list of points (dots) to draw, one at each of the input vectors.
Arrow	Arrow(tip) Arrow(tip, tail)	Draws an arrow from the origin to the tip vector, or from the tail vector to the head vector if a tail is specified.
Segment	Segment(start,end)	Draws a line segment from the start to the vector end.

With these functions in hand, we can draw the points outlining the dinosaur:

```
from vector_drawing import *
dino_vectors = [(6,4), (3,1), (1,2), (-1,5), (-2,5), (-3,4), (-4,4),
    # insert 16 remaining vectors here
]

draw(
    Points(*dino_vectors)
)
```

I didn't write out the complete list of `dino_vectors`, but with the suitable collection of vectors this code will give you the points we saw before.

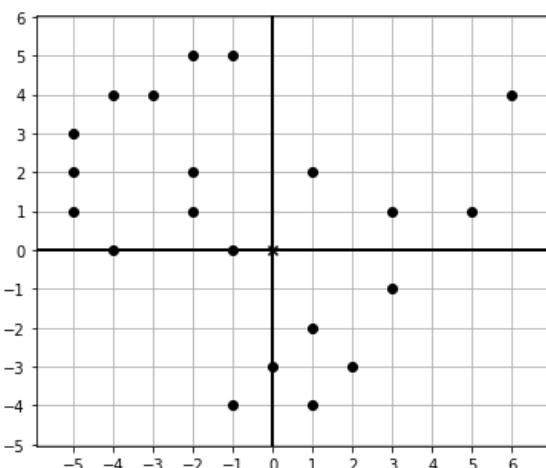


Figure 2.7

As a next step in our drawing process, we can connect some dots. A first segment might connect the point (6,4) with the point (3,1) on the dinosaur's tail. We can draw the points as well as this new segment using as follows:

```
draw_segment(plane, (6,4), (3,1), color='blue')
```

and the result is:

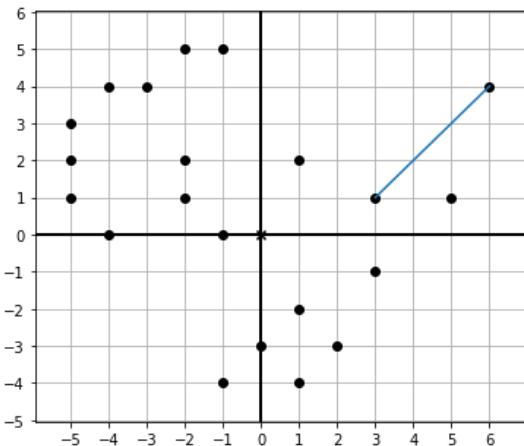


Figure 2.8 The dinosaur's points with a line segment connecting the first two of them.

The line segment is actually the collection consisting of the points (6,4) and (3,1), as well as all of the points lying on the straight line between them. The draw function automatically fills in all of the pixels at those points blue at once. The `Segment` class is a useful abstraction because we don't have to worry about building every segment from the points that make it up. Drawing 20 more segments, we get the complete outline of the dinosaur.

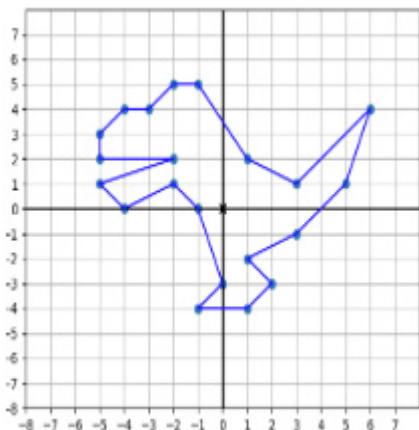


Figure 2.9 21 total function calls give us 21 line segments, completing the outline of the dinosaur.

In principle we can now outline any kind of 2D shape we want, provided we have all of the vectors to specify it. Coming up with all of the coordinates by hand can be tedious, so we'll start to look at ways to do computations with vectors to find their coordinates automatically.

2.1.3 Exercises

EXERCISE

What are the x and y coordinates of the point at the tip of the dinosaur's toe?

SOLUTION

(-1, -4)

EXERCISE

Draw the point in the plane and the arrow corresponding to the point (2, -2).

SOLUTION

Represented as a point on the plane and an arrow, (2,-2) looks like this:

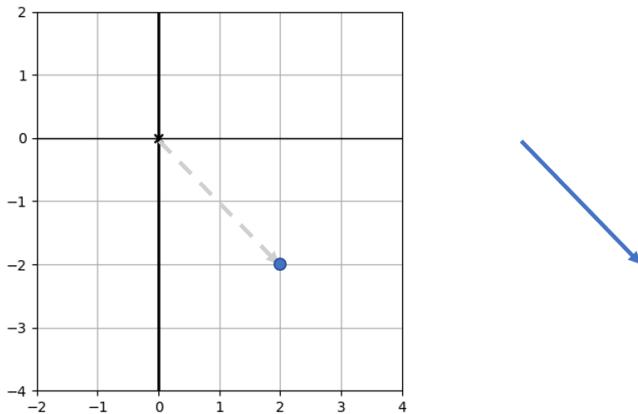


Figure 2.10 The point and arrow representing (2,-2).

EXERCISE

By looking at the locations of these points, infer the remaining vectors I didn't include in the `dino_vectors` list. For instance, I already included (6,4), the tip of the dinosaur's tail, but I didn't include the point (-5,3) on the dinosaur's nose. When you're done, `dino_vectors` should be a list of 21 vectors represented as coordinate pairs.

SOLUTION

The complete set of vectors outlining the dinosaur is:

```
dino_vectors = [(6,4), (3,1), (1,2), (-1,5), (-2,5), (-3,4), (-4,4),
                  (-5,3), (-5,2), (-2,2), (-5,1), (-4,0), (-2,1), (-1,0), (0,-3),
                  (-1,-4), (1,-4), (2,-3), (1,-2), (3,-1), (5,1)]
```

EXERCISE

Draw the dinosaur with the dots connected by constructing a `Polygon` object with the `dino_vectors` as its vertices.

SOLUTION:

```
draw(
    Points(*dino_vectors),
    Polygon(*dino_vectors)
)
```

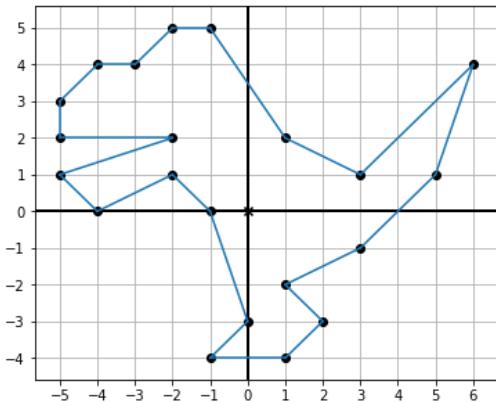


Figure 2.11 The dinosaur drawn as a polygon.

EXERCISE

Draw the vectors `[(x, x**2) for x in range(-10, 11)]` as points (dots) using the `draw` function. What is the result?

SOLUTION

These pairs give the graph of the function $y = x^2$, plotted for integers from -10 to 10.

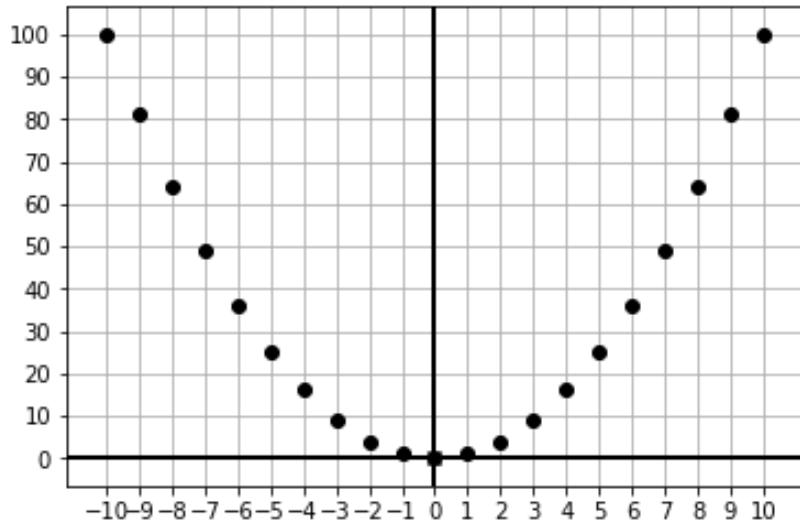


Figure 2.12 Points on the graph of $y = x^2$.

To make this I used two keyword arguments for the draw function. The grid keyword argument of (1,10) specifies to draw vertical grid lines every one unit and horizontal grid lines every ten units. The nice_aspect_ratio keyword argument set to false tells the graph it doesn't have to keep the x-axis scale and the y-axis scale the same.

```
draw(
    Points(*[(x,x**2) for x in range(-10,11)]),
    grid=(1,10),
    nice_aspect_ratio=False # don't require x scale to match y scale
)
```

2.2 Plane vector arithmetic

Like numbers, vectors have their own kind of arithmetic that lets us combine them to make new ones. The exciting difference is that we can visualize the results: operations from vector arithmetic all accomplish useful *geometric* transformations, not just algebraic ones. We'll start with the most basic operation: *vector addition*.

Vector addition is simple to calculate: for two input vectors, you add the x coordinates to get a sum x coordinate and you sum the y coordinates to get a sum y coordinate. These new coordinates yield the *vector sum* of the original vectors. For instance $(4,3) + (-1, 1) = (3, 4)$, since $4+(-1) = 3$ and $3+1 = 4$. That's a one-liner to implement in Python:

```
def add(v1,v2):
    return (v1[0] + v2[0], v1[1] + v2[1])
```

Since we can interpret vectors as arrows or as points in the plane, we can visualize the result of the addition in both ways. As a point in the plane, you can reach $(-1,1)$ by starting at the origin -- which is $(0,0)$ -- and moving one unit to the left and one unit up. You reach the vector sum of $(4,3) + (-1,1)$ by starting instead at $(4,3)$ and moving one unit to the left and one unit up. This is the same as saying you traverse one arrow and then traverse the second.

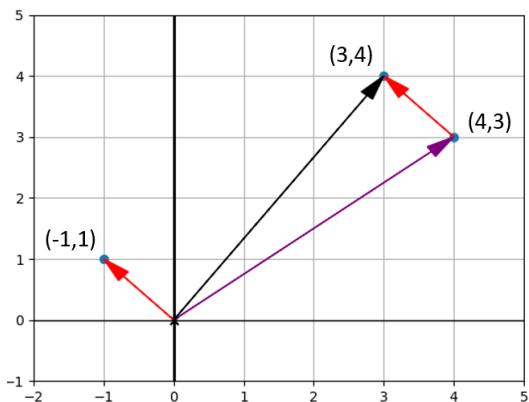


Figure 2.13 Picturing the vector sum of $(4,3)$ and $(-1,1)$.

The rule for vector addition of arrows is sometimes called “tip-to-tail” addition, since if you move the tail of the second arrow to the tip of the first (without rotating either!) then the sum is the arrow from the start of the first to the end of the second.

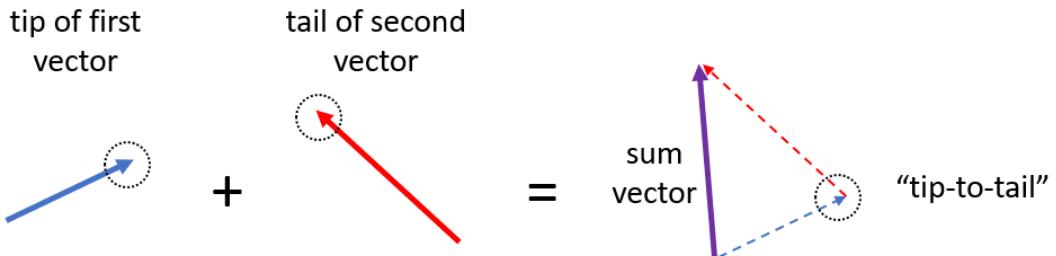


Figure 2.14 : “Tip-to-tail” addition of vectors.

When we talk about arrows, we really mean “a specific distance in a specific direction.” If you walk one distance in one direction, and another distance in another direction, the vector sum tells you the overall distance and direction you traveled.

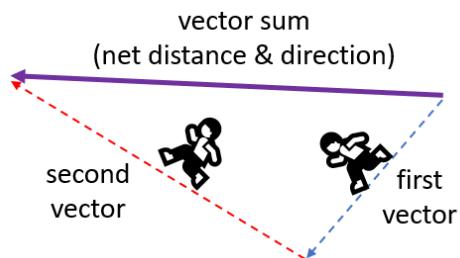


Figure 2.15 The vector sum as an overall distance and direction traveled in the plane.

Adding a vector has the effect of moving or *translating* an existing point or collection of points. If we add the vector $(-1.5, -2.5)$ to every vector of `dino_vectors`, we get a new list of vectors, each of which is 1.5 units left and 2.5 units down from one of the original vectors.

```
dino_vectors2 = [add((-1.5, -2.5), v) for v in dino_vectors]
```

The result is the same dinosaur shape shifted down and to the left by the vector $(-1.5, -2.5)$. To see this, we can draw both dinosaurs as polygons.

```
draw_vectors(plane, dino_vectors, color='blue')
draw_vectors(plane, dino_vectors2, color='red')
connect_the_dots(plane, dino_vectors, color='blue')
connect_the_dots(plane, dino_vectors2, color='red')
```

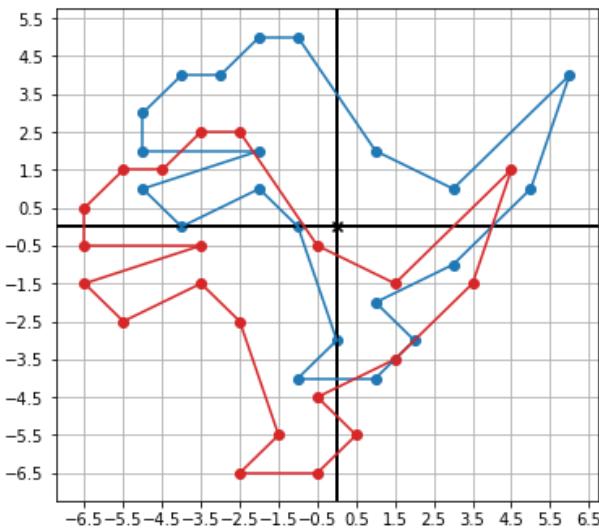


Figure 2.16 The original dinosaur (blue) and the translated copy (red). Each point on the translated dinosaur is moved by $(-1.5, 2.5)$, down and to the left, from its location on the original dinosaur.

The arrows in the copy on the right shows that each point was moved down and to the left by the same vector: $(-1.5, -2.5)$. Translation like this would be useful if we wanted to make the dinosaur the protagonist in a 2D computer game; depending on the button pressed by the user, the dinosaur would translate in the corresponding direction on the screen.

2.2.1 Vector components and lengths

Sometimes it's useful to take a vector we already have and decompose it as a sum of smaller vectors. If I were receiving directions in New York City, it would be much more useful to hear "four blocks east and three blocks north" than "800 meters northeast". Similarly it can be useful to think of vectors as sums of vectors which lie in the x and y directions.

As an example, the vector $(4,3)$ can be re-written as the sum $(4,0) + (0,3)$. Thinking of the vector $(4,3)$ as a navigation path in the plane, the sum $(4,0) + (0,3)$ gets us to the same point along a different path.

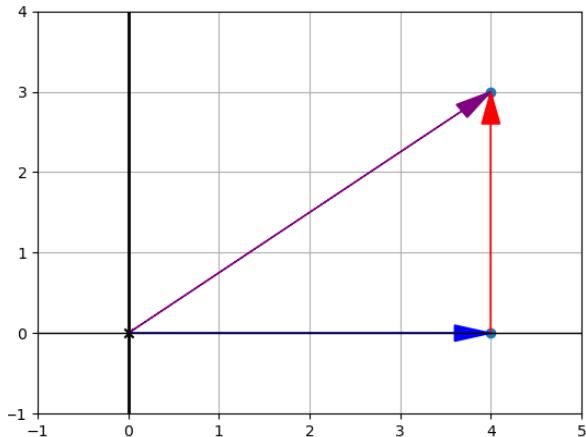


Figure 2.17 Breaking the vector $(4,3)$ into a sum: $(4,0) + (0,3)$.

The two vectors $(4,0)$ and $(0,3)$ are respectively called the *x* and *y components*. If, like on New York's city blocks, you couldn't walk diagonally, you would need to walk four units to the right and *then* three units up -- a total of seven units.

The *length* of a vector is the length of the arrow that represents it, or the equivalently the distance from the origin to the point that represents it. In New York, this could be the distance between two intersections "as the crow flies." The length of a vector in the x or y direction can be read off immediately as a number of ticks passed on the corresponding axis: $(4,0)$ or $(0,-4)$ are both vectors of the same length, four, albeit in different directions. In general though, vectors lie diagonally and we need to do a calculation to get their lengths.

You may recall the relevant formula: the *Pythagorean theorem*. For a *right triangle*, a triangle having two sides meeting at a 90 degree angle, the Pythagorean theorem says that the square of the longest side length is the sum of squares of the other two side lengths. The longest side, called the *hypotenuse* and named c in the formula can be solved for as a square root.

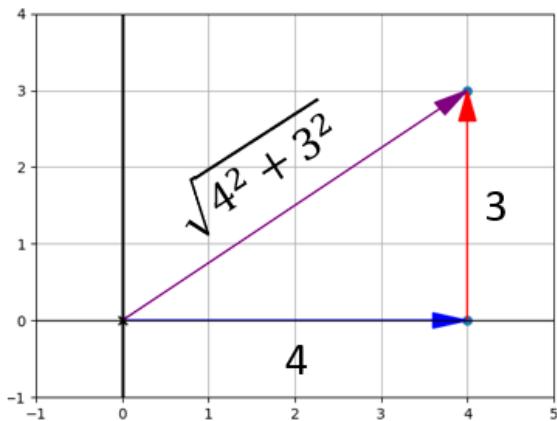


Figure2.18 Using the Pythagorean Theorem to find the length of a vector from the lengths of its *x* component and *y* component.

Breaking a vector into components gives us a right triangle. If we know the lengths of the components we can compute the length of the hypotenuse, which is the length of the vector. Our vector $(4,3)$ is equal to $(4,0) + (0,3)$, a sum of two perpendicular vectors having side-lengths four and three respectively. The length of the vector $(4,3)$ is the square root of $4^2 + 3^2$, which is the square root of 25, or 5. In a city with perfectly square blocks, traveling four blocks east and three blocks north would take us the equivalent of five blocks of distance northeast.

This is a special case where the distance turned out to be an integer, but typically lengths that come out of the pythagorean theorem are not whole numbers. The length of $(-3, 7)$ is given in terms of the lengths of its components, three and seven, by the following computation:

$$\sqrt{3^2 + 7^2} = \sqrt{9 + 49} = \sqrt{58} = 7.61577 \dots$$

Figure2.19 Computing the length of the vector $(-3,7)$ with the Pythagorean theorem.

We can translate this formula into a length function in Python, which takes a vector and returns its floating point length.

```
from math import sqrt
def length(v):
    return sqrt(v[0]**2 + v[1]**2)
```

2.2.2 Multiplying Vectors by Numbers

Repeated addition of vectors is unambiguous: you can keep stacking arrows tip-to-tail as long as you want. If a vector named v has coordinates $(2,1)$, then the five-fold sum $v + v + v + v + v$ would look like this:

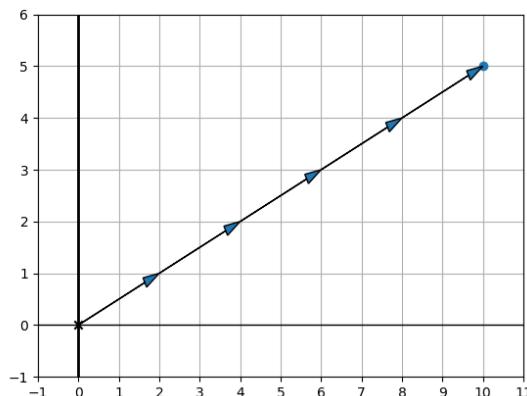


Figure 2.20 Repeated addition of the vector $v = (2,1)$ with itself.

If v were a number, we wouldn't bother writing $v + v + v + v + v$. Instead we'd write the simpler product $5v$. There's no reason we can't do the same for vectors. The result of adding v to itself 5 times is a vector in the same direction but with five times the length. We can run with this definition, which lets us multiply a vector by any whole or fractional number.

The operation of multiplying a vector by a number is called *scalar multiplication*. When working with vectors, ordinary numbers are often called *scalars* to distinguish them. It's also an appropriate term because the effect of this operation is *scaling* the target vector by the given factor. It doesn't matter if the scalar is a whole number: we can easily draw a vector which is 2.5 times the length of another.

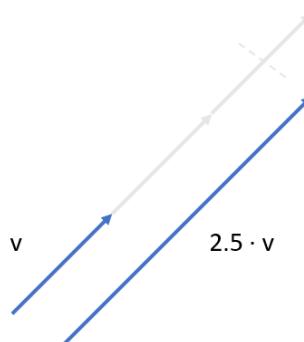


Figure 2.21 Scalar multiplication of a vector v by 2.5.

The result on the vector components is that each component is scaled by the same factor. You can picture scalar multiplication as changing the size of the right triangle defined by a vector and its components -- but not affecting its aspect ratio. In this picture, a vector v and a scalar multiple $1.5 \cdot v$ are superimposed, and scalar multiple is 1.5 times as long. Its components are also 1.5 times the length of the original components of v .

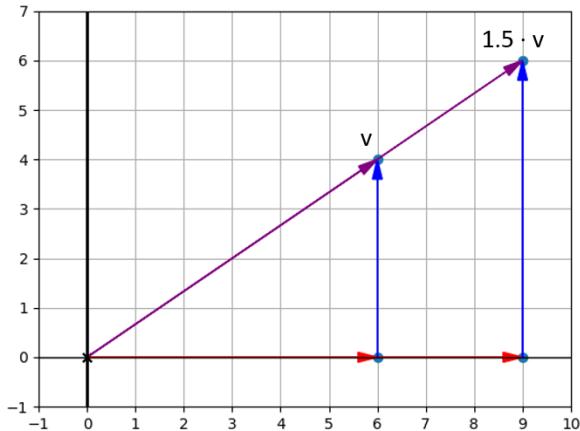


Figure 2.22 Scalar multiplication of a vector scales both components by the same factor.

In coordinates, the scalar multiple of 1.5 times the vector $v = (6,4)$ gives a new vector $(9,6)$, with each component 1.5 times the original. Computationally, we execute any scalar multiplication on a vector by multiplying each coordinate of the vector by the scalar. As a second example, scaling a vector $w = (1.2, -3.1)$ by a factor 6.5 can be accomplished like this:

$$6.5 \cdot w = 6.5 \cdot (1.2, -3.1) = (6.5 \cdot 1.2, 6.5 \cdot -3.1) = (7.8, -20.15).$$

We tested this method for a fractional number as the scalar, but we should also test a negative number. If our original vector is $(6,4)$, what is $-1/2$ times that vector? Multiplying the coordinates, we expect the answer to be $(-3, -2)$. This is a vector which is not only half the length of the original, but it also points in the exact opposite direction.

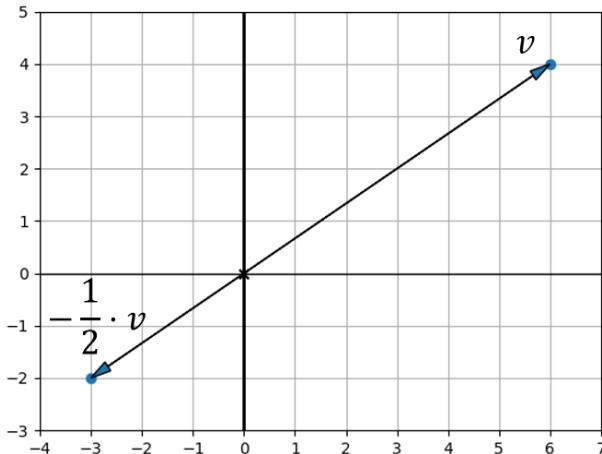


Figure 2.23 Scalar multiplication of a vector by a negative number, $-\frac{1}{2}$.

2.2.3 Subtraction, displacement, and distance

Scalar multiplication agrees with our intuition for multiplying numbers. A whole number multiple of a number is the same as a repeated sum, and the same holds for vectors. We can make a similar argument for “negative” vectors and vector subtraction.

Given a vector v , the negative vector $-v$ is the same as the scalar multiple $-1 \cdot v$. If v is $(-4, 3)$, its *opposite*, $-v$, is $(4, -3)$. We get this by multiplying each coordinate by -1 , or in other words changing the sign of each.

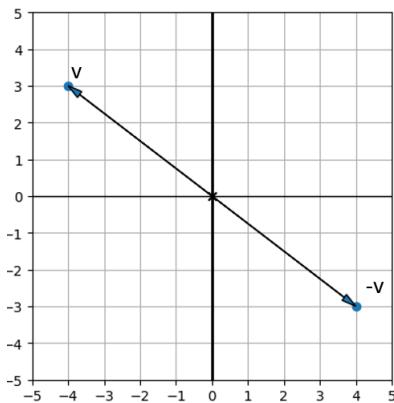


Figure 2.24 The vector $v = (-4, 3)$ and its opposite $-v = (4, -3)$.

On the number line, there are only two directions from zero: positive and negative. In the plane there are many directions (infinitely many, in fact) so we can't say that v is positive

while $-v$ is negative, for instance. What's we can say is that for any vector v , the opposite vector $-v$ will have the same length but it will point in the opposite direction.

Having a notion of negating a vector, we can define *vector subtraction*. For numbers, $x - y$ is the same as $x + (-y)$. We set the same convention for vectors. To subtract a vector w from a vector v , you add the vector $-w$ to v . Thinking of vectors v and w as points, $v - w$ is the position of v relative to w . Thinking instead of v and w as arrows beginning at the origin, $v - w$ is the arrow from the tip of w to the tip of v .

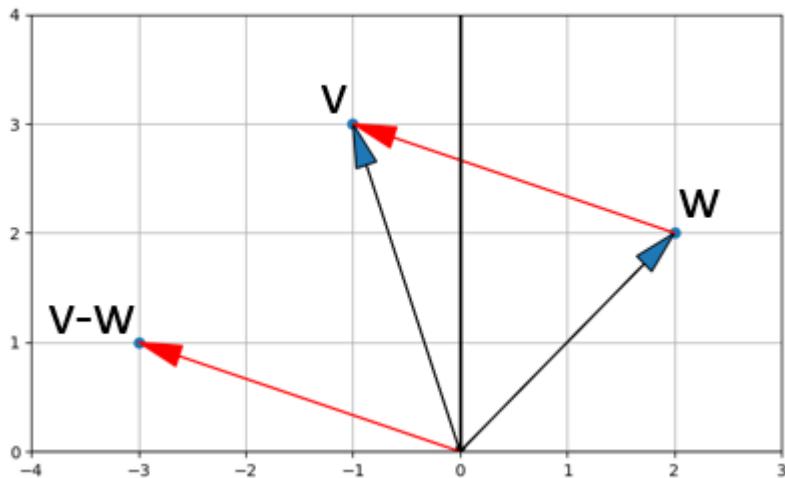


Figure 2.25 The result of subtracting $v - w$ is an arrow from the tip of w to the tip of v .

The coordinates of $v - w$ are the differences of coordinates of v and w . In the drawing above, $v = (-1, 3)$ and $w = (2, 2)$. The difference $v - w$ has coordinates $(-1 - 2, 3 - 2) = (-3, 1)$.

Let's look at the difference of the vectors $v = (-1, 3)$ and $w = (2, 2)$ again. You can use the draw function I gave you to plot the points v and w and draw a segment between them. The code looks like this:

```
set_bounds(plane, (-2,0), (3,4))
draw_vectors(plane, [(2,2),(-1,3)])
draw_segment(plane, (2,2), (-1,3), 'red')
```

The difference $v - w = (-3, 1)$ tells us that if we start at point w , we need to go three units left and one unit up to get to point v . This vector is sometimes called the *displacement* from w to v . The straight, red line segment from w to v drawn by this python code shows us the *distance* between the two points.

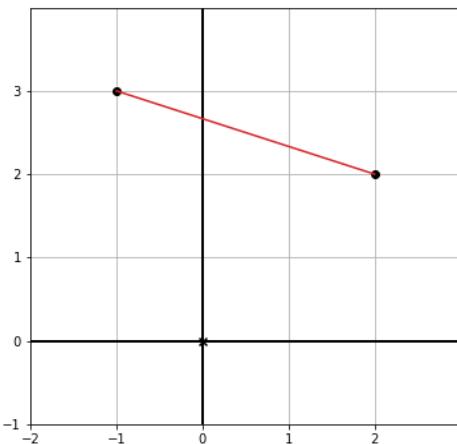


Figure 2.26 The distance between two points in the plane.

The length of the line segment is computed as follows.

$$\sqrt{(-3)^2 + 1^2} = \sqrt{9 + 1} = \sqrt{10} = 3.162\dots$$

Figure 2.27 Computing the distance between the vectors with the Pythagorean Theorem.

While the displacement is a vector, the distance is a scalar (a single number). The distance on its own is not enough to specify how to get from w to v ; there are plenty of points that have the same distance from w . Here are a few others with whole number coordinates:

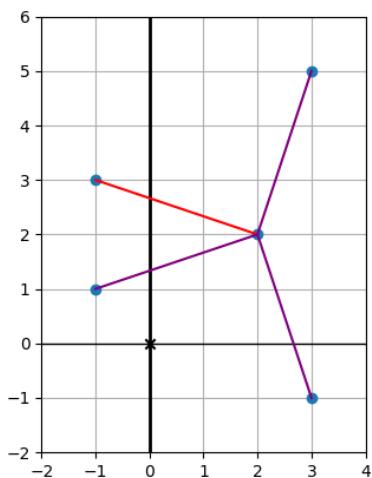


Figure 2.28 Several points equidistant from $w = (2,2)$.

2.2.4 Exercises

EXERCISE

If the vector $u = (-2, 0)$, the vector $v = (1.5, 1.5)$, and the vector $w = (4, 1)$, what are the results of $u + v$, $v + w$, and $u + w$? What is the result of $u + v + w$?

SOLUTION

With the vector $u = (-2, 0)$, the vector $v = (1.5, 1.5)$, and the vector $w = (4, 1)$,

$$\begin{aligned} u + v &= (-0.5, 1.5) \\ v + w &= (5.5, 2.5) \\ u + w &= (2, 1) \\ u + v + w &= (3.5, 2.5) \end{aligned}$$

MINI-PROJECT

You can add any number of vectors together by summing *all* of their x coordinates and *all* of their y coordinates. For instance the four-fold sum $(1,2) + (2,4) + (3,6) + (4,8)$ has x component $1 + 2 + 3 + 4 = 10$ and y component $2 + 4 + 6 + 8 = 20$, making the result $(10,20)$. Implement a revised add function that takes any number of vectors as arguments.

SOLUTION:

```
def add(*vectors):
    return (sum([v[0] for v in vectors]), sum([v[1] for v in vectors]))
```

EXERCISE

Write a function `translate(translation, vectors)` that takes in a translation vector and a list of input vectors and returns a list of the input vectors all translated by the translation vector. For instance, `translate((1,1), [(0,0), (0,1), (-3,-3)])` should return `[(1,1), (1,2), (-2,-2)]`.

SOLUTION:

```
def translate(translation, vectors):
    return [add(translation, v) for v in vectors]
```

MINI-PROJECT

Any sum of vectors $v + w$ gives the same result as $w + v$. Explain why this is true using the definition of the vector sum on coordinates. Also, draw a picture to show why it is true geometrically.

SOLUTION

If you add two vectors $u = (a,b)$ and $v = (c,d)$, the coordinates a, b, c, and d are all real numbers. The result of the vector addition is $u + v = (a+b, c+d)$. The result of $v + u$ is $(b+a, d+c)$, which is the same pair of coordinates since order doesn't matter when adding real numbers.

Visually we can see this by adding an example pair of vectors tip-to-tail.

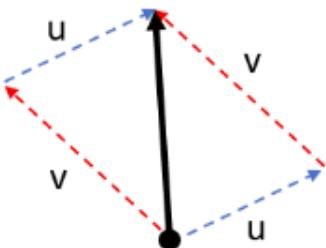


Figure 2.29 Tip-to-tail addition in either order yields the same sum vector.

It doesn't matter whether you add $u + v$ or $v + u$ (dashed), you get the same result vector (solid). In geometry terms, u and v defined a parallelogram and the vector sum is the length of the diagonal.

EXERCISE

Among the following three arrow vectors, labeled u , v , and w , which pair has the sum that gives the *longest* arrow? Which pair sums to give the *shortest* arrow?

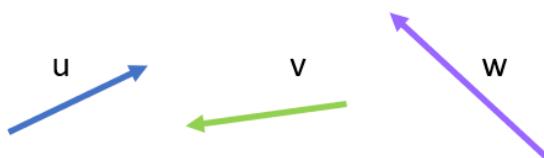


Figure 2.30 Which pair sums to the longest or shortest arrow?

SOLUTION

Solution 10: We can measure each of the vector sums by placing the vectors tip-to-tail:

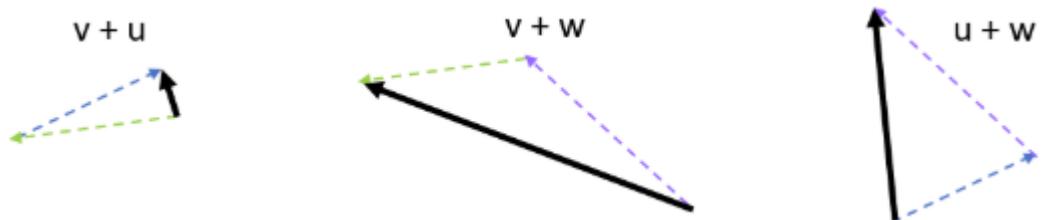


Figure 2.31 Tip-to-tail addition of the vectors in question.

Inspecting the results, we can see that $v+u$ is the shortest vector (u and v are in nearly opposite directions and come close to "cancelling each other out"). The longest vector is $v+w$.

MINI-PROJECT

Write a Python function using vector addition to show 100 simultaneous and non-overlapping copies of the dinosaur. This shows the power of computer graphics: imagine how tedious it would be to specify all 2,100 coordinate pairs by hand!

SOLUTION

With some trial and error, you can translate the dinos in the vertical and horizontal direction so that they don't overlap, and set the boundaries to appropriately. I decided to leave out the grid lines, axes, origin, and points to make the drawing clearer. My code looks like this.

```
def hundred_dinos():
    translations = [(12*x,10*y)
                    for x in range(-5,5)
                    for y in range(-5,5)]
    dinos = [Polygon(*translate(t, dino_vectors),color=blue)
             for t in translations]
    draw(*dinos, grid=None, axes=None, origin=None)
```

The result is as follows.

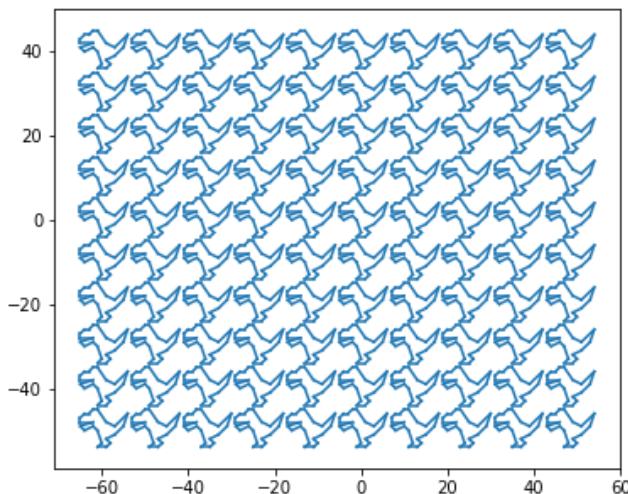


Figure 2.32 100 Dinosaurs. Run for your life!

EXERCISE

Which is longer, the x or y component of $(3,-2) + (1,1) + (-2, -2)$?

SOLUTION

The result of the vector sum $(3,-2) + (1,1) + (-2, -2)$ is $(2, -3)$. The x component is $(2,0)$ and the y component is $(0,-3)$. The x component has length 2 units (to the right) while the y component has length 3 units (downward, since it is negative), making the y component longer.

EXERCISE

What are the components and lengths of the vectors $(-6,-6)$ and $(5,-12)$?

SOLUTION

The components of $(-6,-6)$ are $(-6,0)$ and $(0,-6)$, both having length 6. The length of $(-6,-6)$ is the square root of $6^2 + 6^2 = 72$, which is approximately 8.485.

The components of $(5,-12)$ are $(5,0)$ and $(0,-12)$, having lengths 5 and 12 respectively. The length of $(5,-12)$ is given by the square root of $5^2 + 12^2 = 25 + 144 = 169$. The result of the square root is exactly 13.

EXERCISE

Suppose I have a vector v that has length 6 and x component $(1,0)$. What are the possible coordinates of v ?

SOLUTION

The x component of $(1,0)$ has length 1 and the total length is 6, so the length b of the y component must satisfy the equation $1^2 + b^2 = 36$, or $1 + b^2 = 36$. Then $b^2 = 35$ and the length of the y component is approximately 5.916. This doesn't tell us the direction of the y component: the vector v could either be $(1,5.916)$ or $(1,-5.916)$.

EXERCISE

What vector in the `dino_vectors` list has the longest length? Use the `length` function we wrote to compute the answer quickly.

SOLUTION:

```
>>> max(dino_vectors, key=length)
(6, 4)
```

EXERCISE

Suppose a vector w has coordinates $(\sqrt{2}, \sqrt{3})$. What are the approximate coordinates of the scalar multiple πw ?

- w ? Draw an approximation of the original vector and the new vector.

SOLUTION

The value of `(sqrt(2), sqrt(3))` is approximately

`(1.4142135623730951, 1.7320508075688772)`.

Scaling each coordinate by a factor of pi, we get

(4.442882938158366, 5.441398092702653).

The scaled vector is longer than the original:

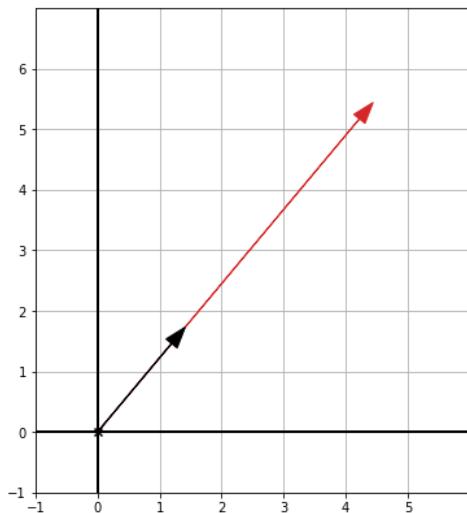


Figure 2.33 The original vector (shorter) and its scaled version (longer).

EXERCISE

Write a python function `scale(s, v)` that multiplies the input vector `v` by the input scalar `s`.

SOLUTION:

```
def scale(scalar,v):
    return (scalar * v[0], scalar * v[1])
```

MINI-PROJECT

Convince yourself algebraically that scaling the coordinates by a factor also scales the length of the vector by the same factor. Suppose a vector of length c has coordinates (a,b) . Show that for any real number s , the length of $(s \cdot a, s \cdot b)$ is $s \cdot c$.

SOLUTION

We use absolute value bars to denote the length of a vector. So, the premise of the exercise tells us:

$$c = \sqrt{a^2 + b^2} = |(a, b)|$$

Figure 2.34 The length of a vector (a,b) .

From that, we can compute the length of (sa,sb).

$$\begin{aligned}
 |(sa, sb)| &= \sqrt{(sa)^2 + (sb)^2} \\
 &= \sqrt{s^2 a^2 + s^2 b^2} \\
 &= \sqrt{s^2(a^2 + b^2)} \\
 &= |s| \sqrt{(a^2 + b^2)} \\
 &= |s| \cdot c
 \end{aligned}$$

Figure 2.35 The length of the vector (sa,sb) in terms of the length of (a,b).

MINI-PROJECT

Suppose $u = (-1,1)$ and $v = (1,1)$ and suppose r and s are real numbers. Specifically, let's assume $-1 < r < 1$ and $-3 < s < 3$.

Where are the possible points on the plane where the vector $r \cdot u + s \cdot v$ could end up?

Note: the order of operations is the same for vectors as it is for numbers: we assume scalar multiplication is carried out first, and then vector addition (unless parentheses specify otherwise).

SOLUTION

If $r = 0$, the possibilities lie on the line segment from $(-1,1)$ to $(1,1)$. If r is not zero, the possibilities can leave that line segment in the direction of $(-1,1)$ or $(1,1)$ by up to three units. The region of possible results is the parallelogram with vertices at $(-2,4)$, $(-4,2)$, $(2,-4)$, and $(4,-2)$.

We can test many random, allowable values of r and s to validate this.

```

u = (-1,1)
v = (1,1)
r = lambda: uniform(-3,3)
s = lambda: uniform(-1,1)
possibilities = [add(scale(r(), u), scale(s(), v)) for i in range(0,500)]
draw_vectors(plane, possibilities)
display(plane)
    
```

If you run this code, you'll get a picture like the following.

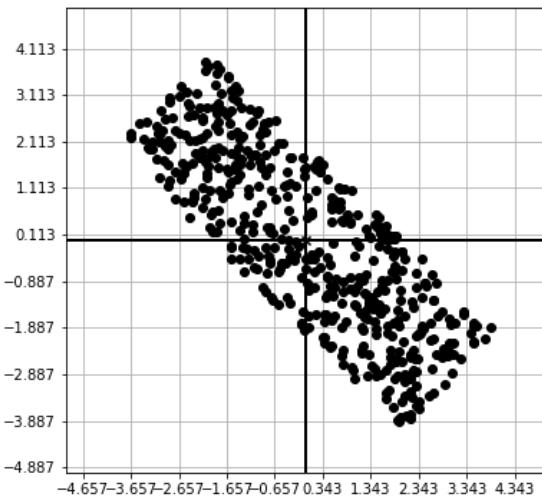


Figure 2.36 : Possible points where $r \cdot u + s \cdot v$ could end up, given the constraints.

EXERCISE

Show algebraically why a vector and its opposite have the same length? Hint: plug coordinates and their opposites into the Pythagorean theorem.

SOLUTION

The opposite vector of (a,b) has coordinates $(-a,-b)$, but this doesn't affect the length.

$$\sqrt{(-a)^2 + (-b)^2} = \sqrt{(-a) \cdot (-a) + (-b) \cdot (-b)} = \sqrt{a^2 + b^2}$$

Figure 2.37 The vector $(-a,-b)$ has the same length as (a,b) .

EXERCISE

Of the following seven vectors, represented as arrows, which two are a pair of opposite vectors?

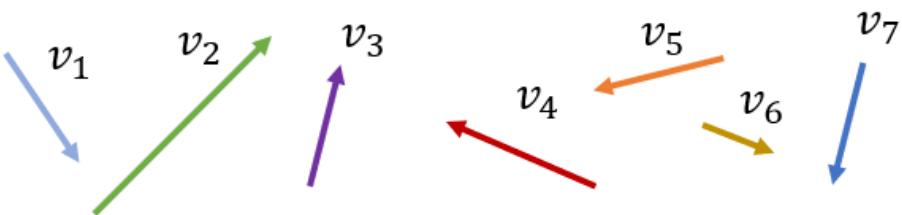


Figure 2.38 Which two are a pair of opposite vectors?

SOLUTION

The two circled vectors are opposites, since they have equal length and opposite direction.

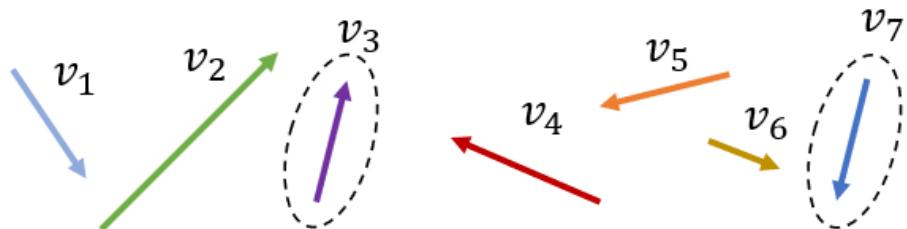


Figure 2.39 Vectors v_3 and v_7 are the pair of opposite vectors.

EXERCISE

Suppose u is any plane vector. What are the coordinates of $u + -u$?

SOLUTION

A plane vector u has some coordinates (a,b) . Its opposite has coordinates $(-a,-b)$, so

$$u + (-u) = (a,b) + (-a,-b) = (a-a,b-b) = (0,0)$$

The answer is $(0,0)$. Geometrically, this means that if you follow one vector and then its opposite, you end up back at the origin, $(0,0)$.

EXERCISE

For vectors $u = (-2, 0)$, $v = (1.5, 1.5)$, and $w = (4, 1)$, what are results of the vector subtractions $v - w$, $u - v$, and $w - v$?

SOLUTION:

With $u = (-2, 0)$, $v = (1.5, 1.5)$, and $w = (4, 1)$, we have $v - w = (-2.5, 0.5)$, $u - v = (-3.5, -1.5)$ and $w - v = (2.5, -0.5)$.

EXERCISE

Write a Python function `subtract(v1, v2)` that returns the result of $v1 - v2$, where the inputs and output are tuples of coordinates as we've seen so far.

SOLUTION:

```
def subtract(v1,v2):
    return (v1[0] - v2[0], v1[1] - v2[1])
```

EXERCISE

Write a Python function `distance(v1, v2)` that returns the *distance* between two input vectors (noting that the `subtract` function from the previous exercise already gives the *displacement*).

Write another Python function `perimeter(vectors)` that takes a list of vectors as an argument and returns the sum of distances from each vector to the next, including the distance from the last vector to the first. What is the perimeter of the dinosaur defined by `dino_vectors`?

SOLUTION

The distance is just the length of the difference of the two input vectors:

```
def distance(v1,v2):
    return length(subtract(v1,v2))
```

For the perimeter, we sum the distances of every pair of subsequent vectors in the list, as well as the pair of the first and the last.

```
def perimeter(vectors):
    distances = [distance(vectors[i], vectors[(i+1)%len(vectors)])
                 for i in range(0,len(vectors))]
    return sum(distances)
```

We can use a square with side-length one as a sanity check.

```
>>> perimeter([(1,0),(1,1),(0,1),(0,0)])
4.0
```

Then, we can calculate the perimeter of the dinosaur.

```
>>> perimeter(dino_vectors)
44.77115093694563
```

MINI-PROJECT

Let u be the vector $(1,2)$. Suppose there is another vector, v , with positive integer coordinates (n, m) such that $n > m$, and having distance 13 from u . What is the displacement from u to v ? Hint: you can use Python to search for the vector v .

SOLUTION

We only need to search possible integer pairs (n,m) where n is within 13 units of 1 and m is within 13 units of 1.

```
for n in range(-12,15):
    for m in range(-14, 13):
        if distance((n,m), (1,-1)) == 13 and n > m > 0:
            print((n,m))
```

There is one result $(13,4)$. It is 12 units to the right and five units up from $(1,-1)$, so the displacement is $(12,5)$.

The length of a vector is not enough to describe it, nor is the distance between two vectors enough information to get from one to the other. In both cases, the missing ingredient is *direction*. If you know how long a vector is and you know what direction it is pointing, you can identify it and find its coordinates. To a large extent, this is what the subject of *trigonometry* is about, and we'll review it in the next section.

2.3 Angles and Trigonometry in the Plane

So far, we've used two "rulers" called the x axis and y axis to measure vectors in the plane. An arrow from the origin covers some measurable displacement in the horizontal and vertical directions and these values uniquely specify the vector. Instead of using two rulers, we could just as well use a ruler and a protractor.

Starting with the vector $(4,3)$, we can measure or calculate its length to be 5 units, and then use our protractor to identify the direction:

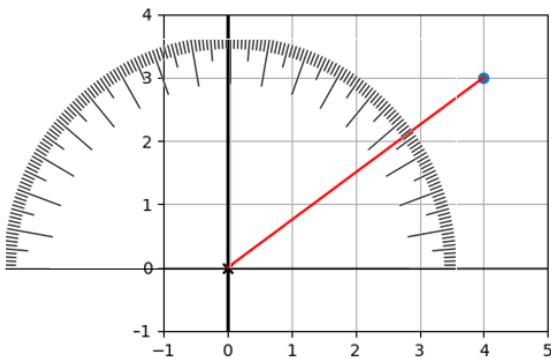


Figure 2.40 Using a protractor to measure the angle at which a vector points.

This vector goes 5 units in a direction which is approximately 37 degrees counterclockwise from the positive half of the x axis. This gives us a new pair of numbers $(5, 37^\circ)$ that, like our original coordinates, uniquely specify the position vector we are talking about. These are called *polar coordinates*, and they are just as good at describing points in the plane as the ones we've been working with, called *cartesian coordinates*.

Sometimes, like when we're adding vectors, we'll want to think about cartesian coordinates. Other times, polar coordinates are more useful -- for instance when we want to look at vectors rotated by some angle. In code, we don't have literal rulers or protractors available, so we'll use trigonometric functions to convert back and forth instead.

2.3.1 From angles to components

Let's look at the reverse problem: say we already have an angle and a distance -- say 116.57° and 3, constituting the pair of polar coordinates $(3, 116.57^\circ)$. How can we find the cartesian coordinates for this vector?

First, we can position our protractor at the origin to find the right direction. We measure out 116.57° counterclockwise from the positive x axis, and draw a line in that direction. Somewhere on this line will lie our vector $(3, 116.57^\circ)$.

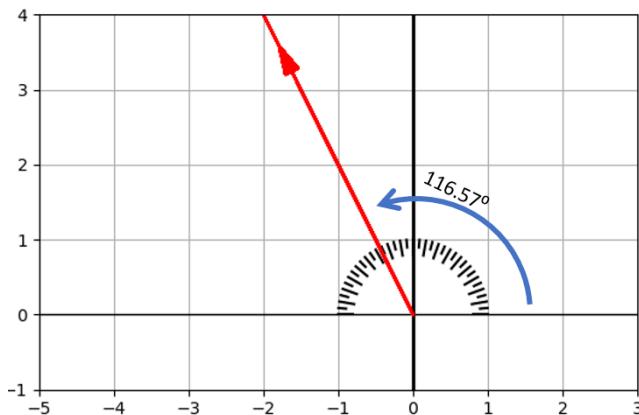


Figure 2.41 First, measuring 116.57° from the positive x-axis using a protractor.

The next step is to take the ruler and measure out a point which is three units from the origin in this direction. Once we've found it, we can measure the components and get our approximate coordinates: $(-1.34, 2.68)$.

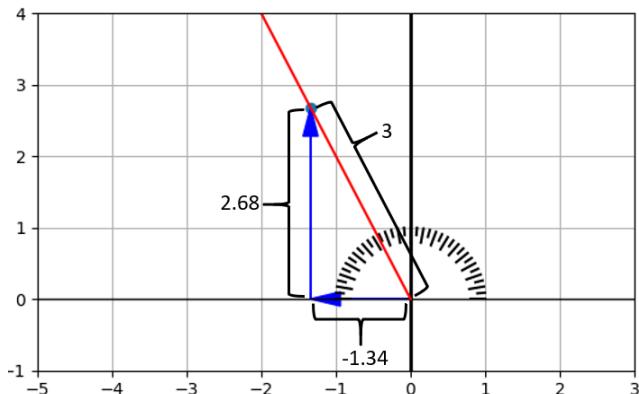


Figure 2.42 Next, using a ruler to measure the coordinates of the point which is three units in this direction.

How could we have found the cartesian coordinates without measurement tools? It may look like the angle 116.57° was a random choice, but it has a useful property. Starting from the origin and moving in that direction, you happen to go up two units every time you go one unit to the left. Vectors that approximately lie along that line include $(-1,2)$, $(-3, 6)$, and of course $(-1.34, 2.68)$; their y coordinates are -2 times their x coordinates.

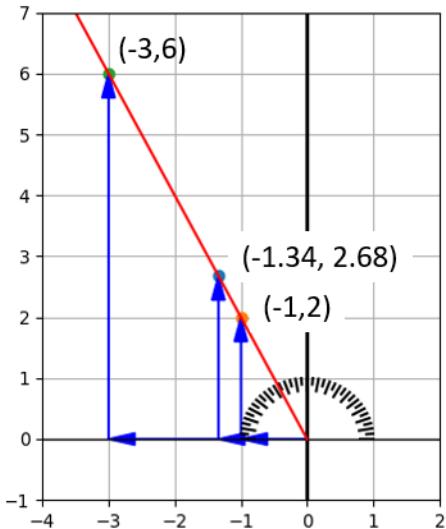


Figure 2.43 Traveling in the direction 116.57° , you travel two units up for every unit you travel to the left.

The strange angle 116.57° happens to give us a nice, round ratio of -2. We won't always be lucky enough to get a whole number ratio, but every angle does give us a *constant* ratio. The angle 45° happens to give us a one vertical unit for every one horizontal unit, or a ratio of 1. Another angle, -200° , gives us a constant ratio of -0.36 vertical units for every -1 horizontal unit covered, or a ratio of 0.36.

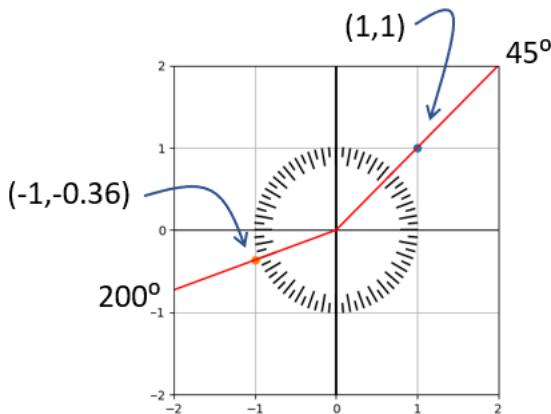


Figure 2.44 How much vertical distance is covered per unit of horizontal distance at different angles?

Given an angle, the coordinates of vectors along that angle will have a constant ratio. This ratio is called the *tangent* of the angle. The tangent function is written “tan”, and we’ve seen a few of its approximate values so far:

$$\tan(37^\circ) \approx \frac{3}{4}$$

$$\tan(116.57^\circ) \approx -2$$

$$\tan(45^\circ) = 1$$

$$\tan(200^\circ) \approx 0.36$$

(Here I use the symbol “ \approx ” as opposed to “ $=$ ” to denote *approximate* equality.) The tangent function is a *trigonometric* function, named such because it helps us measure triangles. Note, I haven’t told you *how* to calculate the tangent yet -- only what a few of its values are. Fortunately, Python has a built-in tangent function that I’ll show you how to use shortly. You’ll almost never have to worry about calculating (or measuring) the tangent of an angle yourself.

The tangent function is clearly related to our original problem of finding cartesian coordinates for a vector given an angle and a distance. But it doesn’t actually provide the coordinates, only their ratio. Two other trigonometric functions will be helpful: *sine* and *cosine*. If we measure some distance at some angle, the tangent of the angle tells us the vertical distance covered divided by the horizontal.

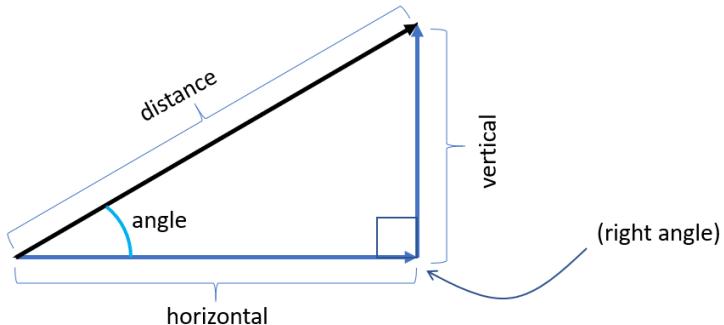


Figure 2.45 Schematic of distances and angles for a given vector.

By comparison, the *sine* and *cosine* tell us the vertical and horizontal distance covered relative to the overall distance. They are written *sin* and *cos* for short:

$$\sin(\text{angle}) = \frac{\text{vertical}}{\text{distance}} \quad \cos(\text{angle}) = \frac{\text{horizontal}}{\text{distance}}$$

Figure 2.46 Definitions of the sine and cosine functions.

Let's look at the angle 37° for a concrete example. We saw that the point $(4,3)$ lies at a distance 5 from the origin at this angle:

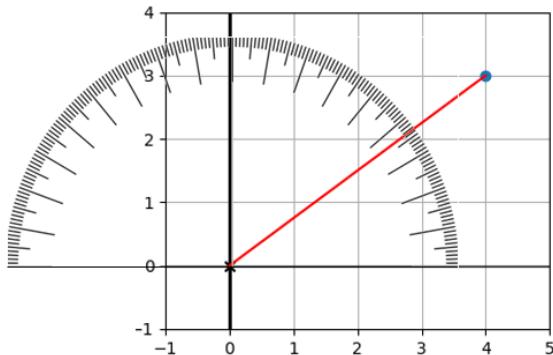


Figure 2.47 Measuring the angle to the point $(4,3)$ with a protractor.

For every 5 units you travel at 37° , you will cover approximately 3 vertical units. Therefore, we write:

$$\sin(37^\circ) \approx 3/5.$$

Similarly, for every 5 units you travel at 37° , you cover approximately 4 horizontal units. Therefore,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/math-for-programmers>

$$\cos(37^\circ) \approx 4/5.$$

This is a general recipe for converting a vector in polar coordinates to corresponding cartesian coordinates. If you know the sine and cosine of an angle θ (the Greek letter “theta”, commonly used for angles) and a distance r traveled in that direction, the cartesian components are given by:

$$(r \cdot \cos(\theta), r \cdot \sin(\theta)).$$

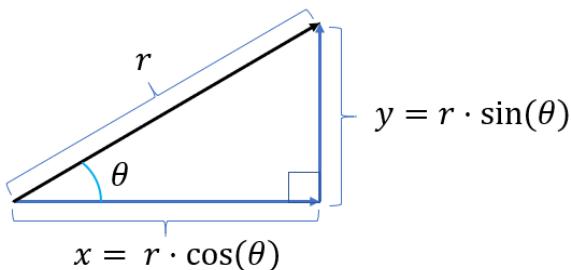


Figure 2.48 Picturing the conversion from polar to cartesian coordinates with a right triangle.

2.3.2 Radians and trigonometry in Python

Let’s turn what we’ve reviewed about trigonometry into Python code. Specifically, let’s build a function that takes a pair of polar coordinates (a length and an angle) and outputs a pair of cartesian coordinates (lengths of x and y components).

The main hurdle is that Python’s built-in trigonometric functions use different units than the ones we’ve been using. We expect $\tan(45^\circ) = 1$, for instance, but Python gives us a much different result:

```
>>> from math import tan
>>> tan(45)
1.6197751905438615
```

Python doesn’t use degrees, and neither do most mathematicians. Instead, they use a units of angle measure called *radians*. The conversion factor is:

$$1 \text{ radian} \approx 57.296^\circ.$$

This may seem like an arbitrary conversion factor. Some more suggestive relationships between degrees and radians are given by:

$$\begin{aligned}\pi \text{ radians} &= 180^\circ \\ 2\pi \text{ radians} &= 360^\circ.\end{aligned}$$

In radians, half a trip around a circle is an angle of π and a whole revolution is 2π . These respectively agree with the half and whole circumference of a circle of radius 1.

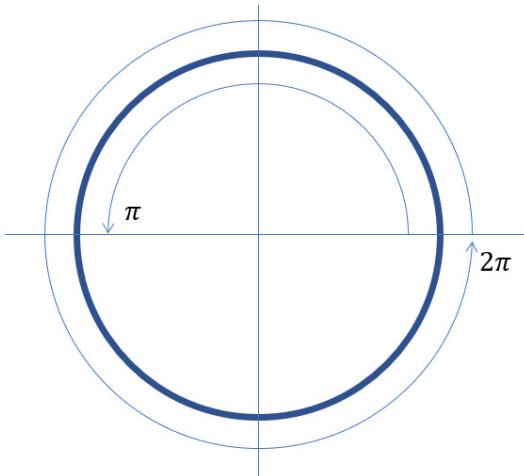


Figure 2.48 A half revolution is π radians while a whole revolution is 2π radians.

You can think of radians as another kind of ratio: for a given angle, they tell you “how many radii you’ve gone around the circle.” Because of this special property, angle measures without units are assumed to be radians. We can note that $45^\circ = \pi/4$ (radians) and get results we expect out of Python’s trigonometric functions.

```
>>> from math import tan, pi
>>> tan(pi/4)
0.9999999999999999
```

We can now make use of Python’s trigonometric functions to write a `to_cartesian` function, taking a pair of polar coordinates and returning corresponding cartesian coordinates.

```
from math import sin, cos
def to_cartesian(polar_vector):
    length, angle = polar_vector[0], polar_vector[1]
    return (length*cos(angle), length*sin(angle))
```

Using this, we can verify that 5 units at an angle of 37 degrees gets us very close to the point (4,3).

```
>>> from math import pi
>>> angle = 37*pi/180
>>> to_cartesian((5,angle))
(3.993177550236464, 3.0090751157602416)
```

2.3.3 From components back to angles

Now that we can convert from polar coordinates to cartesian coordinates, let’s see how to convert in the other direction. Given a pair of cartesian coordinates, like (-2,3) we know how

to find the length with the pythagorean theorem. In this case it is $\sqrt{13}$ which is the first of the two polar coordinates we are looking for. The second is the angle, which we can call θ indicating the direction of this vector.

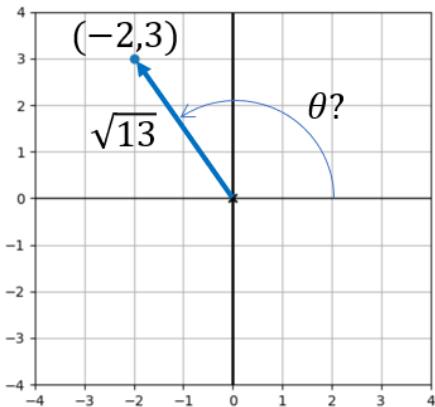


Figure 2.49 In what angle does the vector $(-2,3)$ point?

We can say some facts about the angle θ that we're looking for. Its tangent, $\tan(\theta)$, is $-3/2$, while $\sin(\theta) = 3/\sqrt{13}$ and $\cos(\theta) = -2/\sqrt{13}$. So all that's left is finding a value of θ that satisfies these. If you like, try the next exercise and see if you can find the angle by brute force.

Ideally we'd like a more efficient method than this. It would be great if there were a function that took the value of $\sin(\theta)$, for instance, and gave you back θ . This turns out to be easier said than done, but Python's `math.asin` function makes a good attempt. This is an implementation of the *inverse trigonometric function* called the *arcsine*, and it returns a satisfactory value of θ :

```
>>> from math import asin
>>> sin(1)
0.8414709848078965
>>> asin(0.8414709848078965)
1.0
```

So far, so good. What about the sine of our angle, $3/\sqrt{13}$?

```
>>> from math import sqrt
>>> asin(3/sqrt(13))
0.9827937232473292
```

But this angle is roughly 56.3 degrees, and that's the wrong direction!

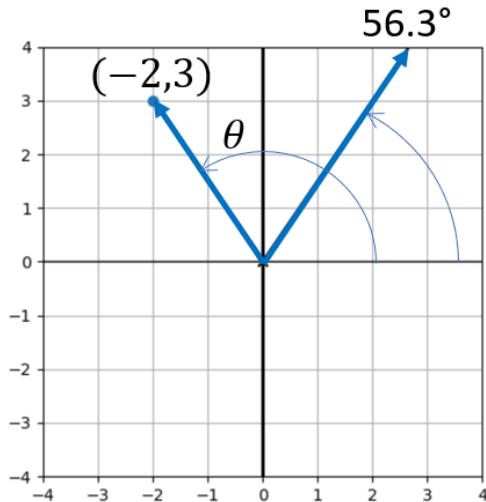


Figure 2.50 Python's `math.asin` appears to give us the wrong angle.

It's not wrong of `math.asin` to give us this answer: another point $(2, 3)$ *does* lie in this direction. It is at length $\sqrt{13}$ from the origin, so the sine of this angle is *also* $3/\sqrt{13}$. This is why `math.asin` is not a full solution for us. There are multiple angles that can have the same sine.

The inverse trigonometric function called *arccosine*, implemented in Python as `math.acos`, happens to give us the right value:

```
>>> from math import acos
>>> acos(-2/sqrt(13))
2.1587989303424644
```

This many radians is about the same as 123.7 degrees, which we can confirm to be correct using a protractor. But this is only by happenstance, there are other angles that could have given us the same cosine. For instance, $(-2, -3)$ also has distance $\sqrt{13}$ from the origin, so it lies at an angle with the same cosine as θ : $-2/\sqrt{13}$.

To find the value of θ that we actually want, we'll have to make sure the sine *and* cosine agree with our expectation. The angle 2.158... satisfies this.

```
>>> cos(2.1587989303424644)
-0.5547001962252293
>>> -2/sqrt(13)
-0.5547001962252291
>>> sin(2.1587989303424644)
0.8320502943378435
>>> 3/sqrt(13)
0.8320502943378437
```

So none of the arcsine, arccosine, or arctangent are sufficient to find the angle to a point in the plane. It *is* possible to find the correct angle by a tricky geometric argument you probably

learned in high school trigonometry class. I'll leave that as an exercise and cut to the chase; Python can do the work for you. The `math.atan2` takes the cartesian coordinates of a point in the plane (in reverse order!) and gives you back the angle at which it lies.

```
>>> from math import atan2
>>> atan2(3,-2)
2.158798930342464
```

I apologize for burying the lede, but did so because it's worth knowing the potential pitfalls of using inverse trigonometric functions. In summary, trigonometric functions are tricky to invert; multiple different inputs can produce the same output, so an output can't be traced back to a unique input. This lets us complete the function we set out to write: a converter from cartesian to polar coordinates.

```
def to_polar(vector):
    x, y = vector[0], vector[1]
    angle = atan2(y,x)
    return (length(vector), angle)
```

We can verify some simple examples: `to_polar((1,0))` should be one unit in the positive x direction, or an angle of zero degrees. Indeed, the function gives us an angle of zero and a length of one.

```
>>> to_polar((1,0))
(1.0, 0.0)
```

(The fact that the input and the output are the same is coincidence; they have different geometric meanings.) Likewise, we get the expected answer for (-2,3).

```
>>> to_polar((-2,3))
(3.605551275463989, 2.158798930342464)
```

2.3.4 Exercises

EXERCISE

Confirm that the vector given by cartesian coordinates (-1.34,2.68) has length approximately 3, as expected.

SOLUTION:

```
>>> length((-1.34,2.68))
2.9963310898497184
```

Close enough.

EXERCISE

The line below makes a 22° angle in the counterclockwise direction from the positive x axis. Based on the picture, what is the approximate value of $\tan(22^\circ)$?

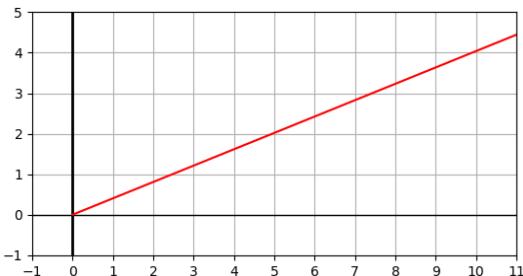


Figure 2.51 A line at 22° .

SOLUTION

The line passes close to the point $(10,4)$, so $4/10 = 0.4$ is a reasonable approximation of $\tan(22^\circ)$.

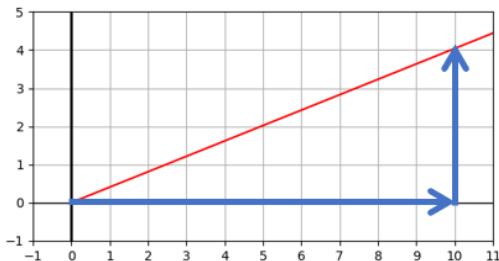


Figure 2.52 The line nearly passes through the point $(10,4)$.

EXAMPLE

Turning the question around, suppose we know the length and direction of a vector and want to find its components. What are the x and y components of a vector with length 15 pointing at a 37° angle?

The sine of 37° is $\frac{3}{5}$, which tells us that every 5 units of distance covered at this angle takes us 3 units upward.

So 15 units of distance give us a vertical component of $\frac{3}{5} \cdot 15$, or 9.

The cosine of 37° is $\frac{4}{5}$, which tells us that each 5 units of distance in this direction take us 4 units to the right, so the horizontal component is $\frac{4}{5} \cdot 15$, or 12. In summary, the polar coordinates $(15, 37^\circ)$ corresponds approximately to the cartesian coordinates $(9, 12)$.

EXERCISE

Suppose I travel 8.5 units from the origin at an angle of 125° , measured counterclockwise from the positive x axis. Given that $\sin(125^\circ) = 0.819$ and $\cos(125^\circ) = -0.574$, what are my final coordinates? Draw a picture to show the angle and path traveled.

SOLUTION:

$$x = r \cdot \cos(\theta) = 8.5 \cdot -0.574 = -4.879$$

$$y = r \cdot \sin(\theta) = 8.5 \cdot 0.819 = 6.962$$

The final position is $(-4.879, 6.962)$.

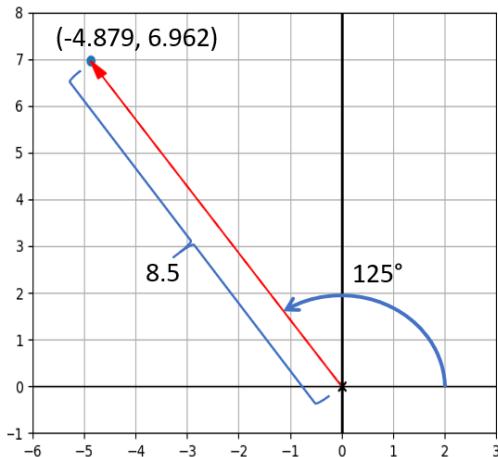


Figure 2.53 8.5 units at 125° takes you to the coordinates $(-4.879, 6.962)$.

EXERCISE

What are the sine and cosine of 0 degrees? Of 90 degrees? Of 180 degrees? In other words, how many vertical and horizontal units are covered per unit distance in any of these directions?

SOLUTION

At zero degrees, no vertical distance is covered so $\sin(0^\circ) = 0$; rather, every unit of distance traveled is a unit of horizontal distance, so $\cos(0^\circ) = 1$.

For 90° , a quarter turn counterclockwise, every unit traveled is a positive vertical unit. So, $\sin(90^\circ) = 1$ while $\cos(90^\circ) = 0$.

Finally, at 180° , every unit of distance traveled is a negative unit in the x direction, so $\cos(180^\circ) = -1$ and $\sin(180^\circ) = 0$.

EXERCISE

The following diagram gives some exact measurements for a right triangle.

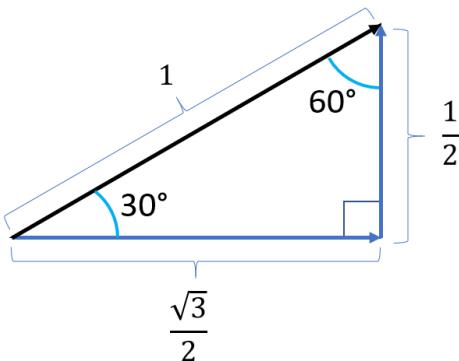


Figure 2.54 Exact measurements of a certain right triangle.

First, confirm that these lengths are valid for a right triangle since they satisfy the Pythagorean theorem. Then, calculate the values of $\sin(30^\circ)$, $\cos(30^\circ)$, and $\tan(30^\circ)$ to three decimal places using the measurements in this diagram.

SOLUTION

These side lengths indeed satisfy the Pythagorean theorem.

$$\sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{\sqrt{3}}{2}\right)^2} = \sqrt{\frac{1}{4} + \frac{3}{4}} = \sqrt{\frac{4}{4}} = 1$$

Figure 2.55 Plugging the side-lengths into the Pythagorean theorem.

The trigonometric function values are given by the appropriate ratios of side lengths.

$$\sin(30^\circ) = \frac{\left(\frac{1}{2}\right)}{1} = 1.000$$

$$\cos(30^\circ) = \frac{\left(\frac{\sqrt{3}}{2}\right)}{1} = 0.866$$

$$\tan(30^\circ) = \frac{\left(\frac{1}{2}\right)}{\left(\frac{\sqrt{3}}{2}\right)} = \frac{1}{\sqrt{3}} = 0.577$$

Figure 2.56 Calculating the sine, cosine, and tangent by their definitions.

EXERCISE

Look at the triangle from the previous exercise from a different perspective, and use it to calculate the values of $\sin(60^\circ)$, $\cos(60^\circ)$, and $\tan(60^\circ)$ to three decimal places.

SOLUTION

Rotating and reflecting the triangle from the previous exercise has no effect on its side lengths or angles.

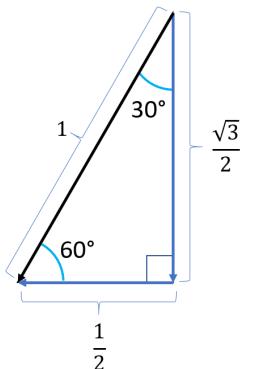


Figure 2.57 A rotated copy of the triangle from the previous exercise.

The ratios of the side lengths give the trigonometric function values for 60 degrees.

$$\sin(60^\circ) = \frac{\left(\frac{\sqrt{3}}{2}\right)}{1} = 0.866$$

$$\cos(60^\circ) = \frac{\left(\frac{1}{2}\right)}{1} = 1.000$$

$$\tan(60^\circ) = \frac{\left(\frac{\sqrt{3}}{2}\right)}{\left(\frac{1}{2}\right)} = \sqrt{3} = 1.732$$

Figure 2.58 Calculating the defining ratios, when horizontal and vertical components have switched.

EXERCISE

The cosine of 50° is 0.643. What is sin(50°) and what is tan(50°)? Draw a picture to help you calculate the answer.

SOLUTION

Given that the cosine of 50° is 0.643, the following triangle is valid:

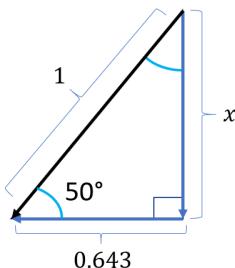


Figure 2.59 A valid triangle with a 50° angle.

That is, it has the right ratio of the two known side lengths: $0.643/1 = 0.643$. To find the unknown side length we can use the Pythagorean theorem.

$$\sqrt{0.643^2 + x^2} = 1$$

$$0.643^2 + x^2 = 1$$

$$0.413 + x^2 = 1$$

$$x^2 = 0.587$$

$$x = 0.766$$

Figure 2.60 Finding the missing side length of the triangle.

With the known side lengths, $\sin(50^\circ) = 0.766 / 1 = 0.766$. Also, $\tan(50^\circ) = 0.766/0.643 = 1.192$.

EXERCISE

What is 116.57° in radians? Use Python to compute the tangent of this angle, and confirm that it is close to -2 as we saw above.

SOLUTION

$$116.57^\circ \cdot (1 \text{ radian} / 57.296^\circ) = 2.035 \text{ radians.}$$

```
>>> from math import tan
>>> tan(2.035)
-1.9972227673316139
```

EXERCISE

Locate the angle $10\pi/6$. Do you expect the values of $\cos(10\pi/6)$ and $\sin(10\pi/6)$ to be positive or negative? Use Python to calculate their values and confirm.

SOLUTION

The angle $\pi/6$ is one third of a quarter-turn, so $10\pi/6$ is less than a quarter turn short of a full rotation. This means that it points "down and to the right". The cosine should be positive and the sine should be negative,

since distance in this direction corresponds with positive horizontal displacement and negative vertical displacement.

```
>>> from math import pi, cos, sin
>>> sin(10*pi/6)
-0.8660254037844386
>>> cos(10*pi/6)
0.5000000000000001
```

EXERCISE

The following list comprehension creates 1000 points in polar coordinates.

```
[ (cos(5*x*pi/500.0), 2*pi*x/1000.0) for x in range(0,1000)]
```

In Python code, convert them to cartesian coordinates, and connect them in a closed loop with line segments to draw a picture.

SOLUTION

Including the set-up and the original list of data, the code is:

```
polar_coords = [(cos(x*pi/100.0), 2*pi*x/1000.0) for x in range(0,1000)]
vectors = [to_cartesian(p) for p in polar]
draw(Polygon(*vectors, color=green))
```

And the result is a five-leaved flower:

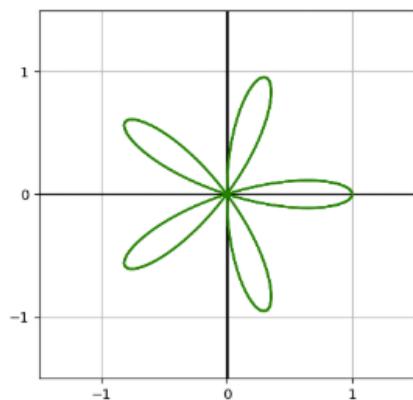


Figure 2.61 The plot of the 1000 connected points is a flower shape.

EXERCISE

Find the angle to get to the point (-2,3) by “guess-and-check”.

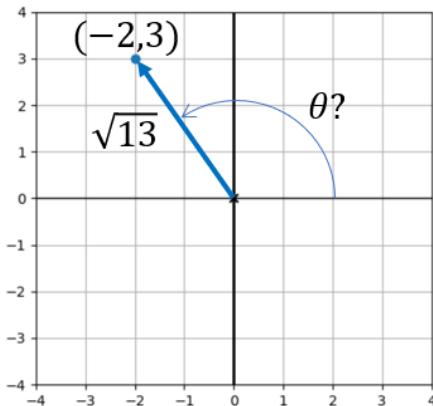


Figure 2.62 What is the angle to get to the point $(-2,3)$?

Hint: we can tell visually that the answer is between $\pi/2$ and π . On that domain, the values of sine and cosine always decrease when the angle increases.

SOLUTION

Here's an example of guessing and checking between $\pi/2$ and π , looking for an angle with tangent close to $-3/2 = -1.5$.

```
>>> from math import tan, pi
>>> pi, pi/2
(3.141592653589793, 1.5707963267948966)
>>> tan(1.8)
-4.286261674628062
>>> tan(2.5)
-0.7470222972386603
>>> tan(2.2)
-1.3738230567687946
>>> tan(2.1)
-1.7098465429045073
>>> tan(2.15)
-1.5289797578045665
>>> tan(2.16)
-1.496103541616277
>>> tan(2.155)
-1.5124173422757465
>>> tan(2.156)
-1.5091348993879299
>>> tan(2.157)
-1.5058623488727219
>>> tan(2.158)
-1.5025996395625054
>>> tan(2.159)
-1.4993467206361923
```

The value must be between 2.158 and 2.159.

EXERCISE

Find another point in the plane with the same tangent as $\theta: -3/2$. Use Python's implementation of the *arctangent* function, `math.atan`, to find the value of this angle.

SOLUTION

Another point with tangent $-3/2$ is $(3,-2)$. Python's `math.atan` finds the angle to this point.

```
>>> from math import atan
>>> atan(-3/2)
-0.982793723247329
```

This is slightly less than a quarter turn in the clockwise direction.

EXERCISE

Without using Python, what are polar coordinates corresponding to the cartesian coordinates $(1,1)$ and $(1,-1)$? Once you've found the answers, use `to_polar` to check your work.

SOLUTION

In polar coordinates, $(1,1)$ becomes $(\sqrt{2}, \pi/2)$ and $(1,-1)$ becomes $(\sqrt{2}, -\pi/2)$.

With some care, you can find any angle on a shape made up of known vectors. The angle between two vectors will be either a sum or difference of angles they make with the x axis. You can try measuring some trickier angles in the next mini-project.

MINI-PROJECT

What is the angle of the Dinosaur's mouth? What is the angle of the dinosaur's toe? Of the point of its tail?

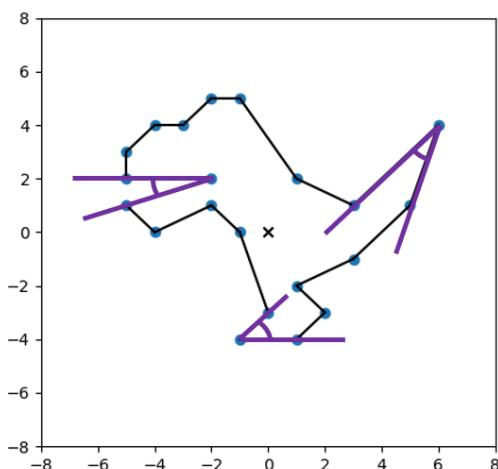


Figure 2.63 Some angles we could measure or calculate on our dinosaur.

2.4 Transforming collections of vectors

Collections of vectors store spatial data -- like drawings of dinosaurs -- regardless of what coordinate system we use, polar or cartesian. It turns out when we want to manipulate vectors, one coordinate system may be better than another. We already saw that moving (or translating) a collection of vectors is easy in cartesian coordinates. It turns out to be much less natural in polar coordinates. However, since polar coordinates have angles built-in, they make it simple to carry out rotations.

In polar coordinates, adding to the angle rotates a vector further counterclockwise, while subtracting from it rotates the vector clockwise. The polar coordinate $(1, 2)$ is at distance one at an angle of two radians. (Remember that we are working in radians if there is no degree symbol!) Starting with the angle two and adding or subtracting one takes the vector either one radian counterclockwise or clockwise respectively.

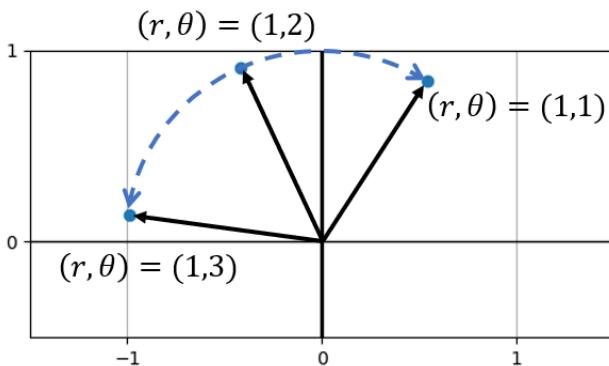


Figure 2.64 Adding or subtracting from the angle rotates the vector about the origin.

Rotating a number of vectors simultaneously has the effect of rotating the figure they represent about the origin. In code, we are only equipped to draw in cartesian coordinates, so we need to convert from polar to cartesian before passing the vectors to our drawing routines. Likewise, we have only seen how to rotate vectors in polar coordinates so we need to convert cartesian coordinates to polar coordinates before executing a rotation. Using this approach, we can rotate the dinosaur:

```
rotation_angle = pi/4
dino_polar = [to_polar(v) for v in dino_vectors]
dino_rotated_polar = [(l,angle + rotation_angle) for l,angle in dino_polar]
dino_rotated = [to_cartesian(p) for p in dino_rotated_polar]
draw(
    Polygon(*dino_vectors, color=gray),
    Polygon(*dino_rotated, color=red)
)
```

The result of this code is a gray copy of the original dinosaur, plus a superimposed red copy which is rotated by $\pi/4$, or an eighth of a full revolution counterclockwise.

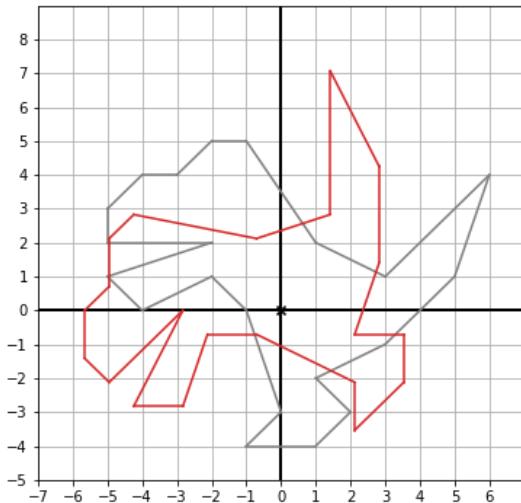


Figure 2.65 The original dinosaur in gray and a rotated copy in red.

As an exercise at the end of the section, I've invited you to write a general-purpose “`rotate`” function that rotates a list of vectors by the same, specified angle. I invite you to skip ahead and try to implement it now, as I'll start using it in examples.

2.4.1 Combining vector transformations

So far, we've seen how to translate, re-scale, and rotate vectors. Applying any of these transformations to a collection of vectors achieves the same effect on the shape that they define in the plane. The full power of these vector transformations comes when we apply them in sequence.

For instance, we could first rotate and *then* translate the dinosaur. Using the `translate` function from before and a `rotate` function you'll write as an exercise, we can write such a transformation concisely.

```
new_dino = translate((8,8), rotate(5 * pi/3, dino_vectors))
```

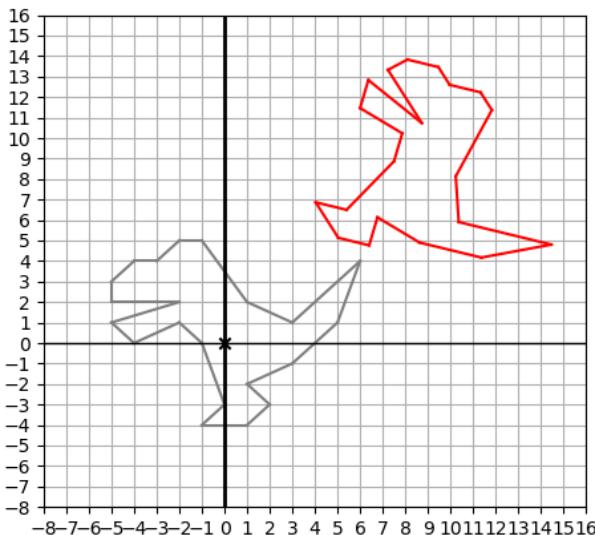


Figure 2.66 The original dinosaur in gray, and a red copy which has been rotated and then translated.

The rotation comes first, and rotates the dinosaur counterclockwise by $5\pi/3$, or most of a full revolution. Then, the dinosaur is translated up and to the right by 8 units each. As you can imagine, combining rotations and translations appropriately can move the dinosaur (or any shape) to any desired location and orientation in the plane. Whether we're animating our dinosaur in a movie or a game, the flexibility to move it around with vector transformations lets us give it life programmatically.

Our applications will soon take us past cartoon dinosaurs -- there are plenty of other operations on vectors, and many generalize to higher dimensions. Real-world data sets often live in dozens or hundreds of dimensions, but we'll apply the same kinds of transformations to them. It's often useful to both translate and rotate data sets to make their important features clearer. We won't be able to picture rotations in 100 dimensions, but now we can always fall back on two dimensions as a trusty metaphor.

2.4.2 Exercises

EXERCISE

Create a `rotate(angle, vectors)` function which takes an array of input vectors in cartesian coordinates and returns them by the specified angle (counterclockwise or clockwise according to whether the angle is positive or negative).

SOLUTION:

```
def rotate(angle, vectors):
    polars = [to_polar(v) for v in vectors]
```

```
return [to_cartesian((l, a+angle)) for l,a in polars]
```

EXERCISE

Create a function `regular_polygon(n)` which returns cartesian coordinates of a *regular n-sided polygon* (that is, having all angles and side lengths equal). For instance, `polygon(7)` could produce vectors defining the following heptagon:

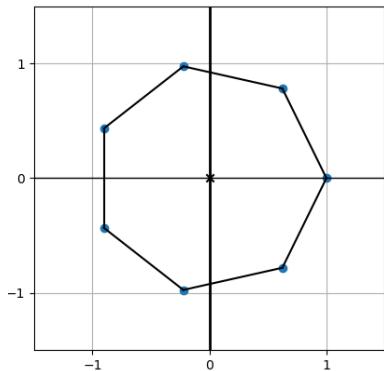


Figure 2.67 A regular heptagon, having points at seven evenly spaced angles around the origin.

Hint: In this picture I used the vector $(1,0)$ and copies which are rotated by seven evenly spaced angles about the origin.

SOLUTION:

```
def regular_polygon(n):
    return [to_cartesian((1, 2*pi*k/n)) for k in range(0,n)]
```

EXERCISE

What is the result of first translating the dinosaur by the vector $(8,8)$ and then rotating by $5\pi/3$? Is the result the same?

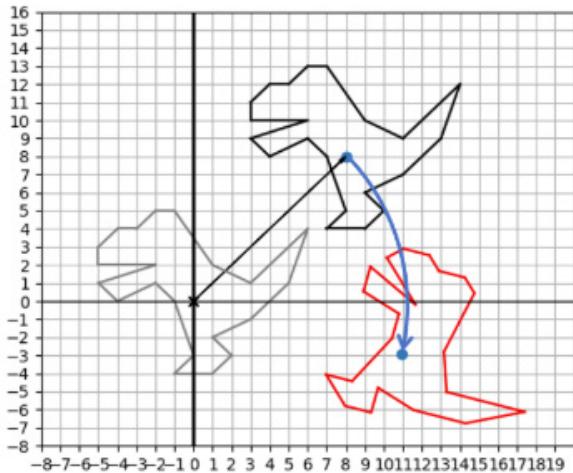
SOLUTION:

Figure 2.68 First translating and then rotating the dinosaur.

The result is *not* the same. In general, applying rotations and translations in different orders yield different results.

2.5 Drawing with Matplotlib

As promised, I'll conclude by showing you how to build the drawing functions used in this chapter "from scratch" using the Matplotlib library. After installing matplotlib with pip, you can import it (and some of its submodules):

```
import matplotlib
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
```

The Polygon, Points, Arrow, and Segment classes are not that interesting -- they simply hold the data passed to them in their constructors. For instance, the Points class contains only a constructor that receives and stores a list of vectors and a color keyword argument:

```
class Points():
    def __init__(self, *vectors, color='black'):
        self.vectors = list(vectors)
        self.color = color
```

The draw function starts by figuring out how big the plot should be, and then it draws each of the objects it is passed one-by-one. For instance, to draw dots on the plane represented by a Points object, it uses Matplotlib's scatter-plotting functionality:

```
def draw(*objects, ...):
    # ...
    for object in objects:
        #1
        #2
```

```
# ...
    elif type(object) == Points: #③
        xs = [v[0] for v in object.vectors]
        ys = [v[1] for v in object.vectors]
        plt.scatter(xs, ys, color=object.color)
# ...
```

- ① Some set-up work happens here, which is not shown.
- ② Iterate over the objects passed in.
- ③ If the current object is an instance of the Points class, draw dots for all of its vectors using Matplotlib's scatter function.

Arrows, segments, and polygons are handled in much the same way, using different pre-built Matplotlib functions to make the geometric objects appear on the plot. You can find all of these implemented in the source code file `vector_drawing.py`. We'll use Matplotlib throughout this book to plot data and mathematical functions, and I'll provide periodic refreshers on its functionality as we use it.

2.6 Summary

In this chapter, you learned that

- Vectors are mathematical objects that live in multi-dimensional spaces. These can be concrete spaces like the 2D plane of a computer screen or the 3D world we inhabit.
- You can think of vectors equivalently as arrows, having specified length and direction, or as points in the plane relative to a reference point called the *origin*. Given a point, there is a corresponding arrow that shows how to get there from the origin.
- Collections of vectors as points in the plane can be connected to form interesting shapes, like a drawing of a dinosaur.
- In 2D, coordinates are pairs of numbers that help us measure the location of points in the plane. Written as a tuple (x,y) , the x and y values tell us how far horizontally and vertically to travel to get to the point. We can store points as coordinate tuples in Python and choose from a number of libraries to draw the points on the screen.
- Vector addition has the effect of translating (or moving) a first vector in the direction of the second vector added. Thinking of a collection of vectors as paths to travel, their vector sum gives the overall direction and distance traveled.
- Scalar multiplication of a vector by a numeric factor yields a vector which is longer by that factor, and pointing in the same direction as the original.
- Subtracting one vector from a second gives the relative position of the second from the first.
- Vectors can be specified by their length and direction (as an angle). These two numbers define the polar coordinates of a given 2D vector. Trigonometric functions sine, cosine, and tangent are used to convert between ordinary (cartesian) coordinates and polar coordinates.
- It's easy to rotate shapes defined by collections of vectors in polar coordinates. You

only need to add or subtract the given rotation angle from the angle of each vector. Rotating and translating shapes in the plane let us place them anywhere and in any orientation.

Now that you've mastered two dimensions, you're ready to add another one. With the third dimension, we can fully describe the world we live in. In the next chapter, you'll see how to model three-dimensional objects in code.

3

Ascending to the 3D World

This chapter covers

- Building a mental model for 3D vectors
- Doing 3D vector arithmetic
- Using the dot product and cross product to measure lengths and directions
- Rendering a 3D object in 2D

The 2D world is easy to visualize, but the real world has three dimensions. Whether we are using software to design a building, animate a movie, or run an action game, it needs to be aware of the three spatial dimensions in which we live.

In a two-dimensional space like a page of this book, we have a vertical and a horizontal direction. Adding a third dimension, we could also talk about points or arrows that lie outside of the page, either toward us or away from us. But, even when they simulate three dimensions, most computer displays are two-dimensional. Our mission in this chapter will be to build the tools we need to take 3D objects measured by 3D vectors and convert them to 2D so they can show up on the screen.

A sphere is one example of a 3D shape. A successfully drawn 3D sphere could look like this:



Figure 3.1 : Shading on a 2D circle makes it look like a 3D sphere.

Without the shading, it would look like a circle. The shading shows that light hits it at a certain angle in 3D, and gives it an illusion of depth. Our general strategy will not be to draw a perfectly round sphere, but an approximation made of polygons. Each polygon can be shaded according to the precise angle it makes with the light source. Believe it or not, the above is not a picture of a round ball, but of 8,000 triangles in varying shades. Here's another example with fewer triangles:

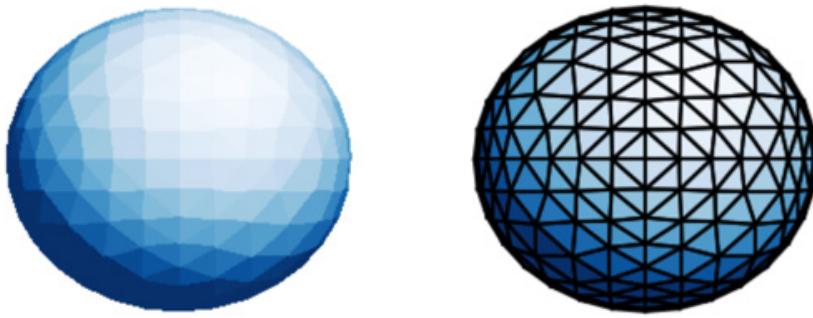


Figure 3.2 Drawing a shaded sphere using many small, solid-colored triangles.

We have the mathematical machinery to define a triangle on a 2D screen: we only need the three 2D vectors defining the corners. But we can't decide how to shade them unless we also think of them as having a life in three dimensions. For this we'll need to learn to work with 3D vectors.

Of course, this is a solved problem, and we will start by using a pre-built library to draw our 3D shapes. Once we've got the feel for the world of 3D vectors, we'll build our own renderer and show how to draw the sphere.

3.1 Picturing vectors in three-dimensional space

In the plane, we worked with three interchangeable mental models of a vector. Coordinate pairs, arrows of fixed length and direction, and points positioned relative to the origin. Since the pages of this book have a finite size, we limited our view to a small portion of the plane -- a rectangle of fixed height and width.

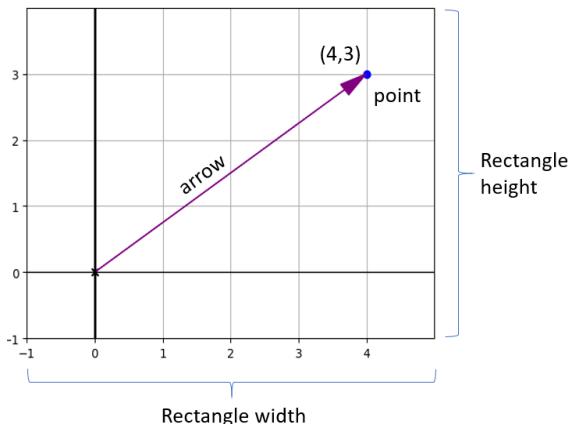


Figure 3.3 The height and width of a small segment of the 2D plane.

We can interpret a three-dimensional vector in similar ways. Instead of viewing a rectangular portion of the plane, we will start with a finite “box” of 3D space. This box has finite height, width, and depth. In this box, we keep the notions of x and y directions, and we add a z direction in which we measure the depth.

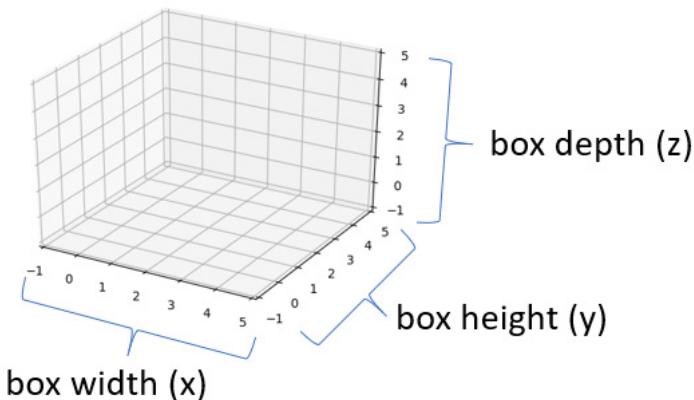


Figure 3.4 A small, finite box of 3D space has a width, a height, and a depth.

We can describe any 2D vectors as living in this 3D space, having their same size and orientation but fixed to a plane where the depth, z , is zero. Here's the 2D drawing of the vector $(4,3)$ embedded in 3D space, with all the same features as it had before. The second drawing annotates all of the features that are still included.

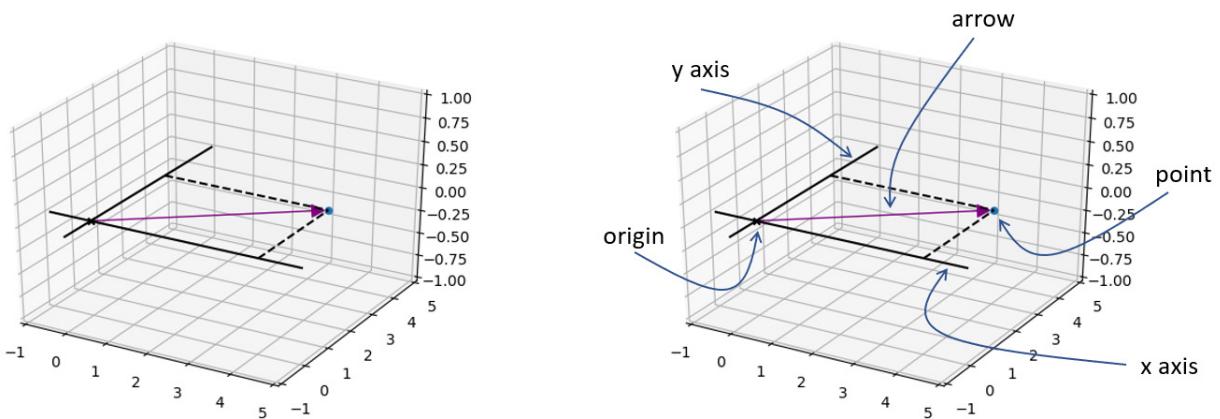


Figure 3.5 The 2D world and inhabitant vector $(4,3)$ are contained in the 3D world.

The dashed lines make a rectangle in the 2D plane where depth is zero. It will be helpful to draw dashed lines -- always at right angles -- to help us locate points in 3d. Otherwise our perspective might deceive us, and a point may not be where we think it is.

Our vector still lives in a plane, but now we can also see it lives in a bigger 3D space. We can draw another 3d vector (a new arrow and a new point) that lives off of the original plane, extending to a higher depth value.

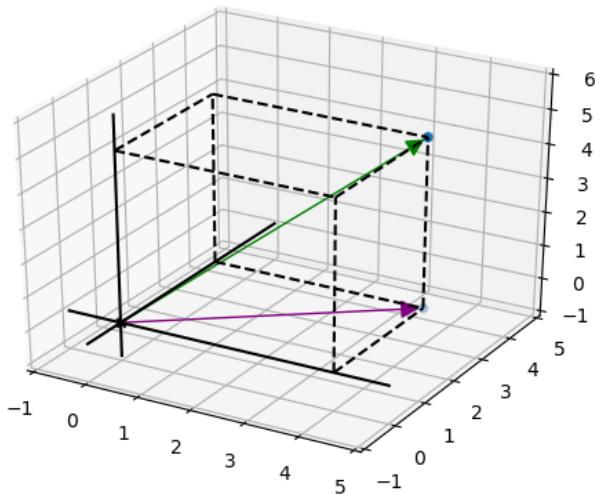


Figure 3.6 A vector extending into the third dimension, as compared with (4,3).

To make the location of this second vector clear, I drew a dashed box instead of a dashed rectangle. This shows the length, width, and depth it covers in three-dimensional space. Arrows and points seem to work as mental models for vectors in 3D, and we'll see that coordinates work as well.

3.1.1 Representing 3D vectors with coordinates

The pair of numbers (4,3) is enough to specify a single point or arrow in 2D, but in 3D there are numerous points with x-coordinate 4 and y-coordinate 3. In fact, there is a whole line of points in 3D with these coordinates, each having different positions in the z (or depth) direction.

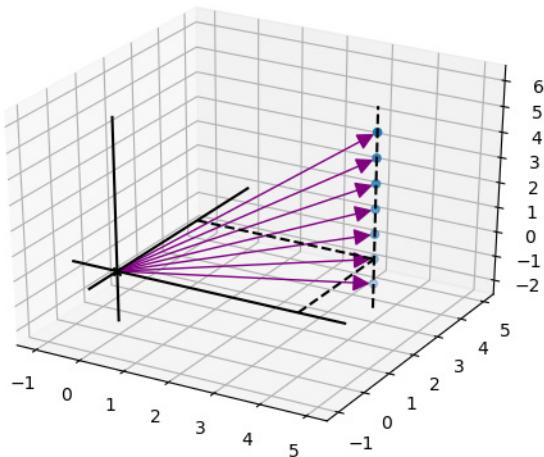


Figure 3.7 Several vectors with the same x and y coordinates but different z coordinates.

To specify a unique point in 3D, we need three numbers in total. A triple of numbers like $(4,3,5)$ are called the x , y , and z coordinates for a vector in 3D. As before, we can read these as instructions to find the desired point. First, we go +4 units in the x direction, then go +3 units in the y direction, then finally go +5 units in the z direction.

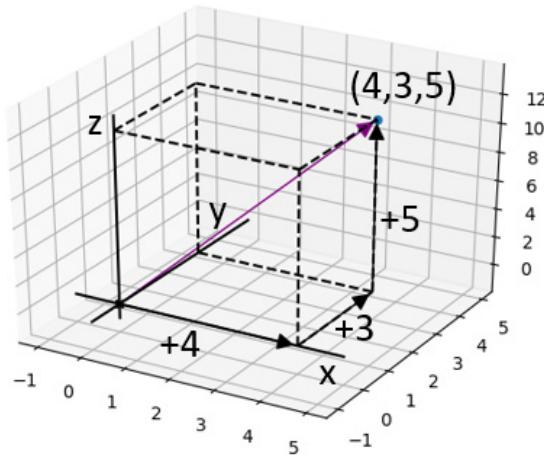


Figure 3.8 The three coordinates $(4,3,5)$ give us directions to a point in 3D.

3.1.2 3D Drawing in Python

As in the previous chapter, I'll use a wrapper around Python's Matplotlib library to make vector drawings in 3D. You can find the implementation in the source code, but I'll stick with the wrapper to focus on the conceptual process of drawing rather than the details of Matplotlib.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/math-for-programmers>

My wrapper uses new classes like `Points3D` and `Arrow3D` to distinguish 3D objects from their 2D counterparts. A new function, `draw3d`, knows how to interpret these objects and render them so as to make them look three-dimensional. By default, `draw3d` shows the axes and the origin, as well as a small “box” of 3D space, even if no objects are specified for drawing.

```
draw3d()
```

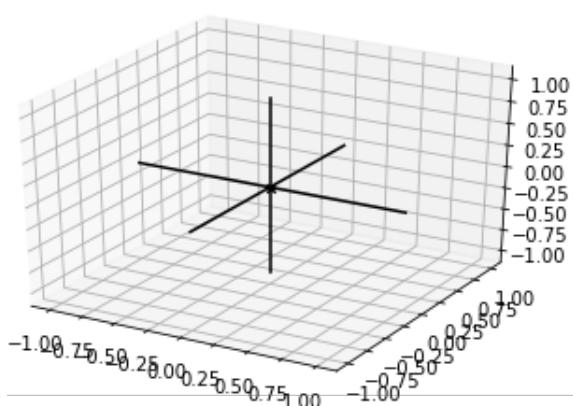


Figure 3.9 Drawing an empty region of 3D with Matplotlib.

The `x`, `y`, and `z` axes that are drawn are perpendicular in the space, despite being skewed by our perspective. For visual clarity, Matplotlib shows the units outside the box, but the origin and the axes themselves are shown within the box. The origin is the coordinate $(0,0,0)$, and the axes emanate from it in the positive and negative `x`, `y`, and `z` directions.

The `Points3D` class stores a collection of vectors we want to think of as points, and therefore draw as dots in 3D space. For instance, we could plot the vectors $(2,2,2)$ and $(1,-2,-2)$.

```
draw3d(
    Points3D((2,2,2),(1,-2,-2))
)
```

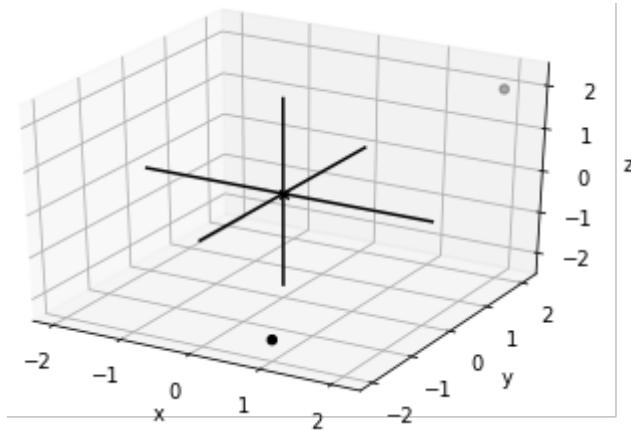


Figure 3.10 Drawing the points $(2,2,2)$ and $(1,-2,-2)$.

To visualize these vectors instead as arrows, we can represent the vectors as `Arrow3D` objects. We can also connect the tips of arrows with a `Segment3D` object.

```
draw3d(
    Points3D((2,2,2),(1,-2,-2)),
    Arrow3D((2,2,2)),
    Arrow3D((1,-2,-2)),
    Segment3D((2,2,2), (1,-2,-2))
)
```

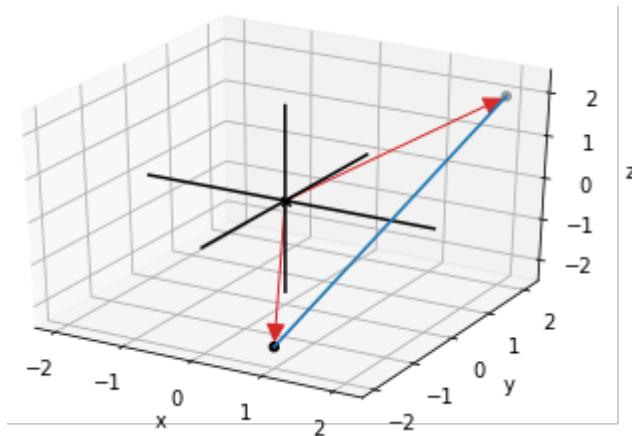


Figure 3.11 : Drawing 3D arrows.

It's a bit hard to see which direction the arrows are pointing in this diagram. To make it clearer, we can draw dashed boxes around the arrows to make them look more "3D." Since

we'll draw these boxes so frequently, I created a `Box3D` class to represent a box with one corner at the origin and the opposite one at a given point..

```
draw3d(
    Points3D((2,2,2), (1,-2,-2)),
    Arrow3D((2,2,2)),
    Arrow3D((1,-2,-2)),
    Segment3D((2,2,2), (1,-2,-2)),
    Box3D(2,2,2),
    Box3D(1,-2,-2)
)
```

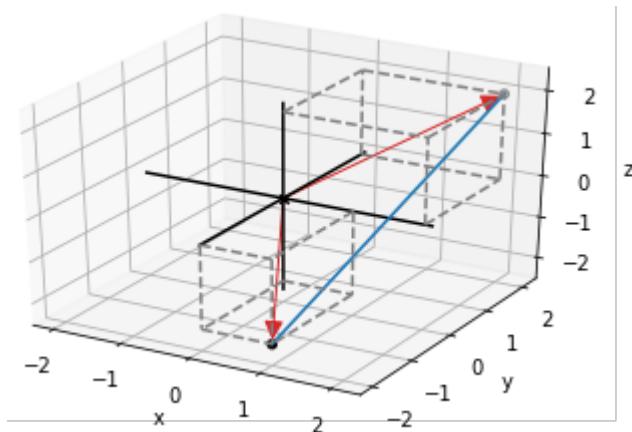


Figure 3.12 Drawing boxes to make our arrows look more “three-dimensional.”

Finally, I'll use a number of keyword arguments without introducing them explicitly. For instance, a “color” keyword argument can be passed to most of these methods, controlling the color of the object that shows up in the drawing.

3.1.3 Exercises

EXERCISE

Draw the 3D arrow and point representing the coordinates (-1,-2,2), as well as the dashed box that makes the arrow look 3D. Do this drawing by hand as practice, but from now on we'll use Python to draw for us.

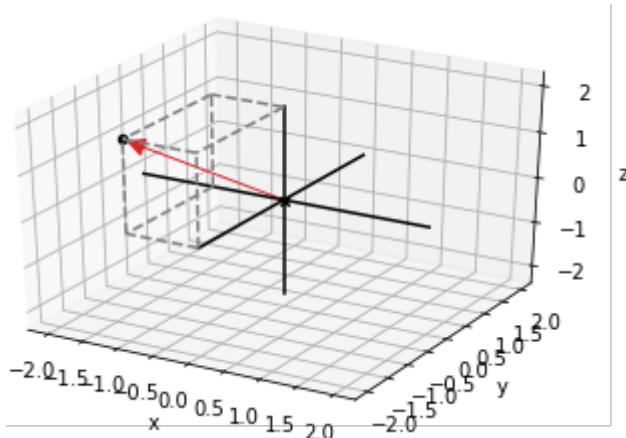
SOLUTION:

Figure 3.13 The vector $(-1, -2, 2)$ and the box that makes it look 3 dimensional.

MINI PROJECT

There are eight 3D vectors that have every coordinate equal to $+1$ or -1 . For instance, $(1, -1, 1)$ is one of them. Plot all of these eight vectors as points. Then, figure out how to connect them with line segments (using `Segment3D` objects) to form the outline of a cube. Hint: you'll need 12 segments in total.

SOLUTION

Since there are only eight vertices and 12 edges, it's not too tedious to list them all out. I decided to enumerate them with a list comprehension. For the vertices, I let x , y , and z range over the list of two possible values, $\{-1, 1\}$, and collected the eight results. For the edges, I grouped them into the three sets of four that point in each coordinate direction. For instance, there are four edges that go from $x=-1$ to $x=1$, while their x and z coordinates are the same at both endpoints.

```
pm1 = [1, -1]
vertices = [(x,y,z) for x in pm1 for y in pm1 for z in pm1]
edges = [((-1,y,z),(1,y,z)) for y in pm1 for z in pm1] + \
        [((x,-1,z),(x,1,z)) for x in pm1 for z in pm1] + \
        [((x,y,-1),(x,y,1)) for x in pm1 for y in pm1]
draw3d(
    Points3D(*vertices,color=blue),
    *[Segment3D(*edge) for edge in edges]
)
```

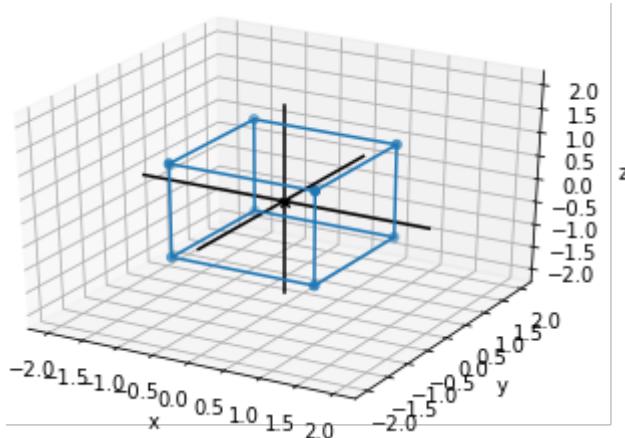


Figure 3.14 The cube with all vertex coordinates equal to +1 or -1.

3.2 Vector arithmetic in 3D

With these Python functions in hand, it will be easy to visualize the results of vector arithmetic in three dimensions. All of the arithmetic operations we saw in 2D have analogies in 3D, and the geometric effects of each are similar.

3.2.1 Adding 3D vectors

In 3D, vector addition can still be accomplished by adding coordinates. The vectors $(2,1,1)$ and $(1,2,2)$ sum to $(2+1, 1+2, 1+2) = (3,3,3)$. We can start at the origin and place the two input vectors tip-to-tail in either order to get to the sum point $(3,3,3)$.

$$(2,1,1) + (1,2,2) = (3,3,3)$$

$$(1,2,2) + (2,1,1) = (3,3,3)$$

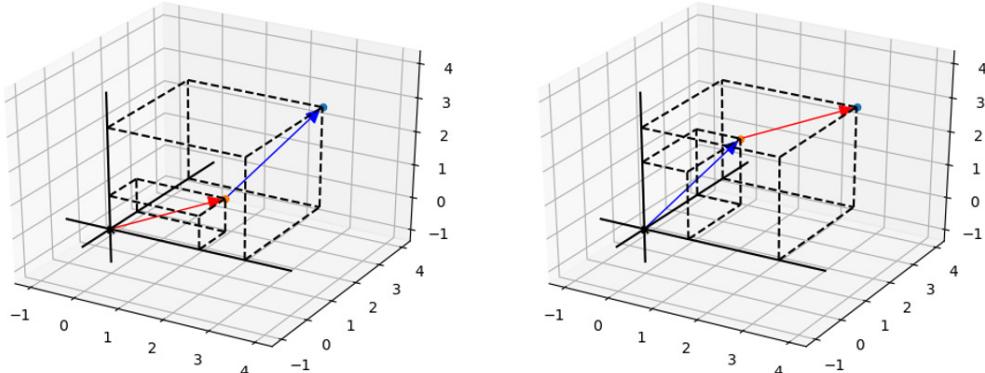


Figure 3.15 Two visual examples of vector addition in 3D.

Like in 2D, we can add any number of 3D vectors together by summing all of their x coordinates, all of their y coordinates, and all of their z coordinates. These three sums give us the coordinates of the new vector. For instance in the sum $(1,1,3) + (2,4,-4) + (4,2,-2)$, the respective x coordinates are 1, 2, and 4, which sum to 7. The y coordinates sum to 7 as well and the z coordinates sum to -3. Therefore, the vector sum is $(7,7,-3)$. Tip to tail, the three vectors look like this.

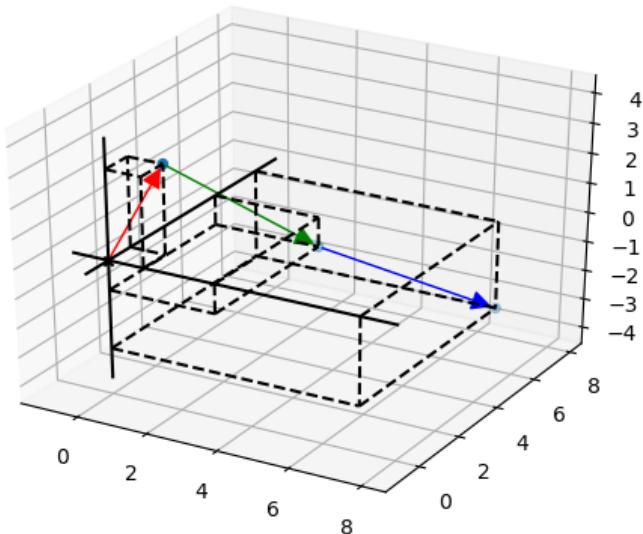


Figure 3.16 Adding three vectors tip-to-tail in 3D.

In Python, we can write a concise function to sum over any number of input vectors and that works in two or three dimensions (or an even higher a number of dimensions, as we'll see later). Here it is:

```
def add(*vectors):
    by_coordinate = zip(*vectors)
    coordinate_sums = [sum(coords) for coords in by_coordinate]
    return tuple(coordinate_sums)
```

Let's break it down. Calling Python's `zip` function with the identical arguments re-groups them by coordinate. Passing in the three vectors above, `zip` gives us a list containing a tuple of all the x coordinates, a tuple of all the y-coordinates, and a tuple of all the z-coordinates.

```
>>> list(zip([(1,1,3),(2,4,-4),(4,2,-2)]))
[(1, 2, 4), (1, 4, 2), (3, -4, -2)]
```

(You need to convert the `zip` result to a list to see its values printed.) If we apply Python's `sum` function to each of these, we get sums of x, y, and z values, respectively.

```
[sum(coords) for coords in [(1, 2, 4), (1, 4, 2), (3, -4, -2)]]
```

```
[7, 7, -3]
```

Finally, for consistency, we convert this from an array to a tuple, since we've represented all of our vectors as tuples to this point. The result is the tuple (7,7,3). We could also have written the add function as a one-liner (which is perhaps less Pythonic):

```
def add(*vectors):
    return tuple(map(sum,zip(*vectors)))
```

3.2.2 Scalar Multiplication in 3D

To multiply a 3D vector by a scalar, we multiply all of its components by the scalar factor. For instance the vector (1,2,3) multiplied by the scalar 2 gives (2, 4, 6). This resulting vector is twice as long but pointing in the same direction, just as we saw in the 2D case. Here are $v = (1,2,3)$ and its scalar multiple $2v = (2,4,6)$:

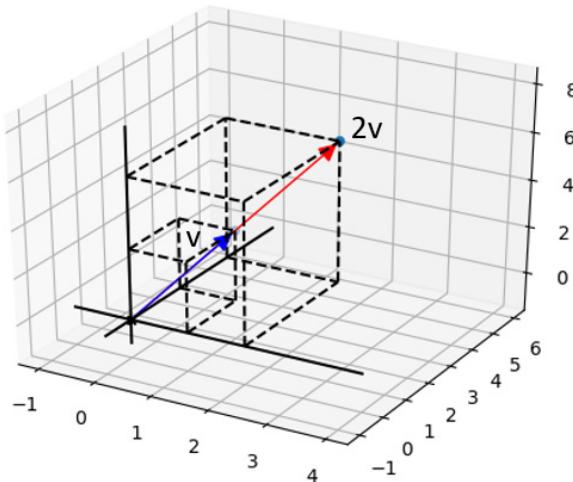


Figure 3.17 Scalar multiplication by 2 gives a vector which is twice as long in the same direction.

3.2.3 Subtracting 3D vectors

In 2D, the difference of two vectors $v - w$ gave the vector "from w to v ," called the displacement. In 3D the story is the same. In other words, $v - w$ is the vector you can add to w to get v . Thinking of v and w as arrows from the origin, their difference $v - w$ is an arrow that can be positioned to have its tip at the tip of v and its tail at the tip of w . For $v = (-1, -3, 3)$ and $w = (3, 2, 4)$ the difference is shown below, both as an arrow from w to v and as a point in its own right.

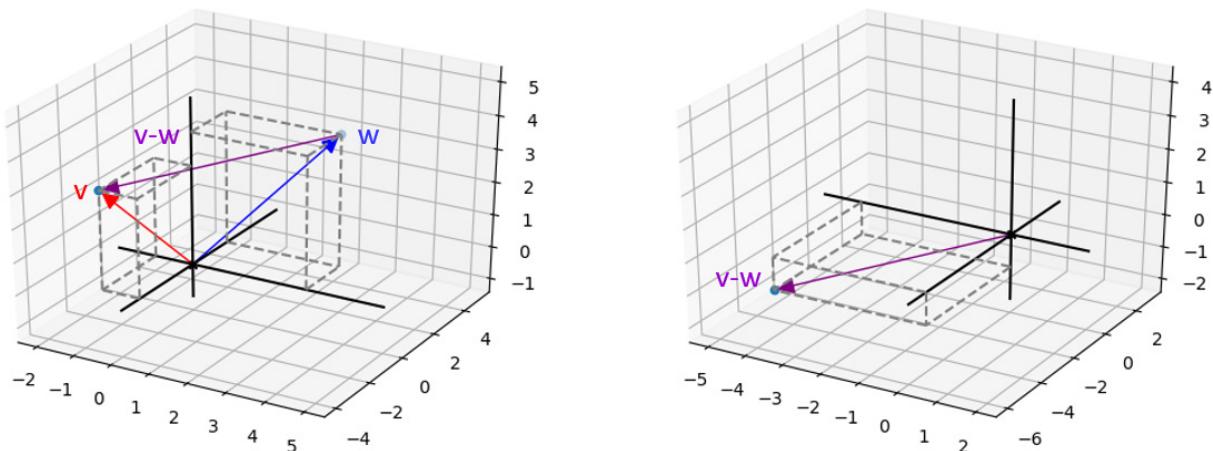


Figure 3.18 Subtracting the vector w from the vector v gives the displacement from w to v .

Subtracting a vector w from a vector v is accomplished in coordinates by taking the difference of the coordinates of v and w . For instance $v - w$ above give us $(-1-3, -3-2, 3-4) = (-4, -5, -1)$ as a result. These coordinates agree with the picture of $v - w$, which shows it pointing in the negative x , negative y , and negative z directions.

When I claim scalar multiplication by two makes a vector “twice as long,” I’m thinking in terms of geometric similarity. If each of the three components of v are doubled, corresponding to doubling the length, width, and depth of the box, the diagonal distance from one corner to the other should also double. To actually measure and confirm this, we’ll need to know how to calculate distances in 3D.

3.2.4 Computing lengths and distances

In 2D, the length of a vector was calculated with the Pythagorean theorem, using the fact that an arrow vector and its components make a right triangle. Likewise, the distance between two points in the plane was just the length of their difference vector.

We have to look a bit closer, but we can still find a suitable right triangle in 3D to help us calculate the length of a vector. Let’s try to find the length of the vector $(4, 3, 12)$. The x and y components still give us a right triangle, lying in the plane where $z = 0$. This triangle’s hypotenuse, or diagonal side, has length $\sqrt{4^2 + 3^2} = \sqrt{25} = 5$. If this were a two-dimensional vector we’d be done, but the z -component of 12 makes this vector quite a bit longer.

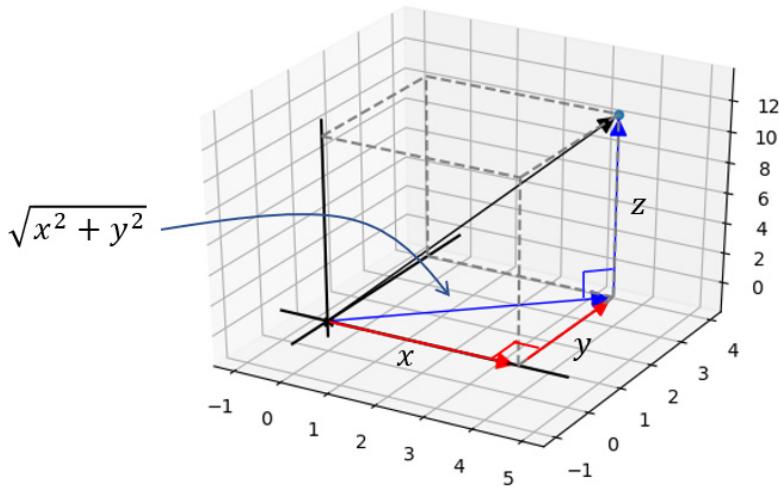


Figure 3.19 Applying the Pythagorean theorem to find the length of a hypotenuse in the x,y plane.

So far all of the vectors we considered lie in the “x,y plane,” where $z = 0$. The x component is $(4,0,0)$, the y component is $(0,3,0)$, and their vector sum is $(4,3,0)$. The z component of $(0,0,12)$ is perpendicular to all three of these. That’s useful, because it gives us a second right triangle in the diagram, the one formed by $(4,3,0)$ and $(0,0,12)$ placed tip-to-tail. The hypotenuse of this triangle is our original vector $(4,3,12)$, whose length we want to find.

Let’s focus in on this second right triangle and invoke the Pythagorean theorem again to find the hypotenuse length.

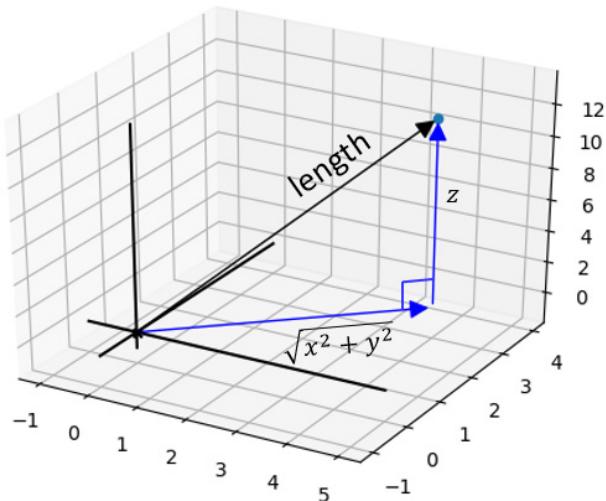


Figure 3.20 A second application of the Pythagorean theorem gives us the length of the 3D vector.

Squaring both known sides and taking the square root should give us the length. Here, the lengths are 5 and 12, so the result is $\sqrt{5^2 + 12^2} = 13$. In general, the result is

$$\text{length} = \sqrt{\left(\sqrt{x^2 + y^2}\right)^2 + z^2} = \sqrt{x^2 + y^2 + z^2}$$

Figure 3.21 The formula for the length of a vector in 3D.

This is conveniently similar to the 2D length formula; in either 2D or 3D, the length of a vector is the square root of the sum of squares of its components. Because we don't explicitly reference the length of the input tuple anywhere in the following implementation, it will work on 2D or 3D vectors.

```
from math import sqrt
def length(v):
    return sqrt(sum([coord ** 2 for coord in v]))
```

So, for instance, `length((3, 4, 12))` returns 13.

3.2.5 Computing angles and directions

As in 2D, you can think of a 3D vector as an arrow or a displacement of a certain length in a certain direction. In 2D, this meant that two numbers -- one length and one angle making a pair of polar coordinates -- were sufficient to specify any 2D vector. In 3D, one angle is not sufficient to specify a direction, but two angles are.

For the first angle, we again think of the vector without its z coordinate, as if it still lived in the x, y plane. Another way of thinking of this is as the shadow cast by the vector from a light at a very high z position. This shadow makes some angle with the positive x axis, analogous to the angle we used in polar coordinates, and we label it with the greek letter phi: ϕ . The second angle is the one that the vector makes with the z axis, which is labeled θ .

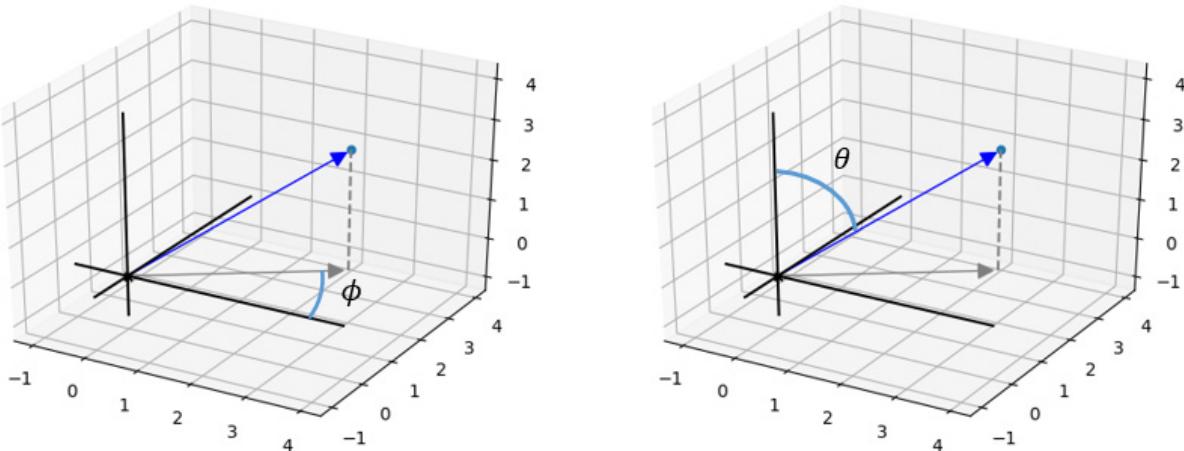


Figure 3.22 Two angles which together measure the direction of a 3D vector.

The length of the vector, labeled r , along with the angles ϕ and θ can describe any vector in three dimensions. Together, the three numbers r , ϕ , and θ are called spherical coordinates, as opposed to the cartesian coordinates x , y , and z . Calculating them is an exercise in trigonometry, and you can give it a try yourself.

We won't spend too much time with spherical coordinates, except to observe their shortcomings. Polar coordinates were useful because they allowed us to perform any rotation of a collection of plane vectors by simply adding or subtracting from the angle. The angle between two vectors could also be read off by taking the difference of their angles in polar coordinates. In three dimensions, neither of the angles ϕ and θ would let us immediately decide the angle between two vectors. And while we could rotate vectors easily around the z axis by adding or subtracting from the angle ϕ , it's not convenient to rotate about any other axis in spherical coordinates.

We need some more general tools to handle angles and trigonometry in 3D. We'll see two of them now, called vector products.

3.2.6 Exercises

EXERCISE

Draw $(4,0,3)$ and $(-1,0,1)$ as `Arrow3D` objects, such that they are placed tip-to-tail in both orders in 3D. What is their vector sum?

SOLUTION

We can find the vector sum using the `add` function we built.

```
>>> add((4,0,3),(-1,0,1))
(3, 0, 4)
```

Then to draw them tip-to-tail, we draw arrows from the origin to each point, and from each point to the vector sum $(3,0,4)$. Like the 2D Arrow object, Arrow3D takes the “tip” vector of the arrow first, and then optionally the “tail” vector if it is not the origin.

```
draw3d(
    Arrow3D((4,0,3),color=red),
    Arrow3D((-1,0,1),color=blue),
    Arrow3D((3,0,4),(4,0,3),color=blue),
    Arrow3D((-1,0,1),(3,0,4),color=red),
    Arrow3D((3,0,4),color=purple)
)
```

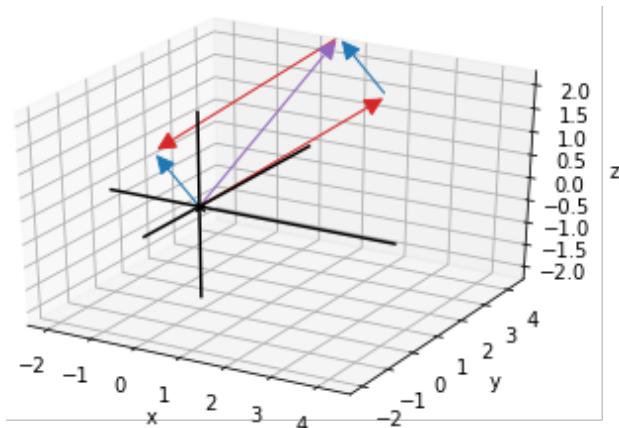


Figure 3.23 Tip to tail addition shows $(4,0,3) + (-1,0,1) = (-1,0,1) + (4,0,3) = (3,0,4)$.

EXERCISE

Suppose we set `vectors1=[(1,2,3,4,5), (6,7,8,9,10)]` and `vectors2=[(1,2), (3,4), (5,6)]`. Without evaluating in Python, what are the lengths of `zip(*vectors1)` and `zip(*vectors2)`?

SOLUTION

The first zip has length 5: since there are five coordinates in each of the two input vectors, `zip(*vectors1)` contains five tuples, having two elements each. Likewise `zip(*vectors2)` has length 2; the two entries of `zip(*vectors2)` are tuples containing all of the x components and all of the y components, respectively.

MINI PROJECT

The comprehension below creates a list of 24 Python vectors

```
from math import sin, cos, pi
vs = [(sin(pi*t/6), cos(pi*t/6), 1.0/3) for t in range(0,24)]
```

What is the sum of the 24 vectors? Draw all 24 of them tip-to-tail as `Arrow3D` objects.

SOLUTION

Drawing these vectors tip-to-tail ends up producing a “helix” shape.

```
from math import sin, cos, pi
vs = [(sin(pi*t/6), cos(pi*t/6), 1.0/3) for t in range(0,24)]

running_sum = (0,0,0) # ❶
arrows = []
for v in vs:
    next_sum = add(running_sum, v) ❷
    arrows.append(Arrow3D(next_sum, running_sum))
    running_sum = next_sum
print(running_sum)
draw3d(*arrows)
```

- ❶ We begin a running sum at (0,0,0), where the tip-to-tail addition begins.
- ❷ To draw each subsequent vector tip-to-tail, we add it to the running sum. The latest arrow connects the previous running sum to the next.

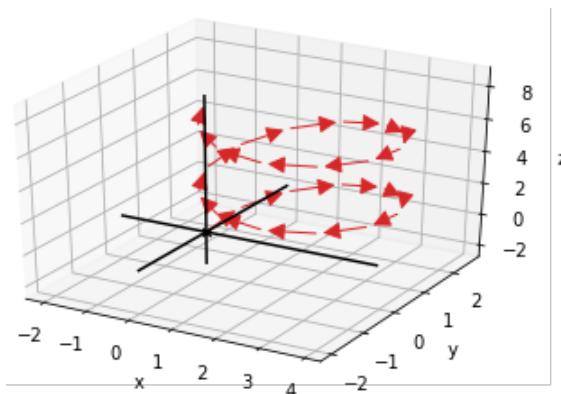


Figure 3.24 Finding the vector sum of 24 vectors in 3D.

The sum is

(-4.440892098500626e-16, -7.771561172376096e-16, 7.9999999999999964)

which is very close to (0,0,8).

EXERCISE

Write a function `scale(scalar, vector)` that returns the input scalar times the input vector. Specifically, write it so it works on 2D or 3D vectors, or vectors of any number of coordinates.

SOLUTION

With a comprehension, we multiply each coordinate in the vector by the scalar. This is a generator comprehension which is converted to a tuple.

```
def scale(scalar,v):
    return tuple(scalar * coord for coord in v)
```

EXERCISE

Let $u = (1, -1, -1)$ and $v = (0, 0, 2)$. What is the result of $u + \frac{1}{2} * (v - u)$?

SOLUTION

With $u = (1, -1, -1)$ and $v = (0, 0, 2)$ we can first compute $(v-u) = (0-1, 0-(-1), 2-(-1)) = (-1, 1, 3)$. Then $\frac{1}{2} * (v-u)$ is $(-\frac{1}{2}, \frac{1}{2}, \frac{3}{2})$. The final desired result of $u + \frac{1}{2} * (v - u)$ is then $(\frac{1}{2}, -\frac{1}{2}, \frac{1}{2})$. Incidentally, this is the point exactly halfway between the point u and the point v .

EXERCISE

Try to find these answers without using code, and then check your work. What is the length of the 2D vector $(1, 1)$? What is the length of the 3D vector $(1, 1, 1)$? We haven't yet talked about 4D vectors, but they have four coordinates instead of two or three. If you had to guess, what is the length of the 4D vector with coordinates $(1, 1, 1, 1)$?

SOLUTION

The length of $(1, 1)$ is $\sqrt{1^2 + 1^2} = \sqrt{2}$. The length of $(1, 1, 1)$ is $\sqrt{1^2 + 1^2 + 1^2} = \sqrt{3}$. As you might guess, we use the same distance formula for higher dimensional vectors as well. The length of $(1, 1, 1, 1)$ follows the same pattern: it is $\sqrt{1^2 + 1^2 + 1^2 + 1^2} = \sqrt{4}$ which is 2.

MINI PROJECT

The coordinates 3, 4, 12 in any order create a vector of length 13, a whole number. This is unusual because most numbers are not perfect squares, so the square root in the length formula typically returns an irrational number. Find a different triple of whole numbers that define coordinates of a vector with whole number length.

SOLUTION

The following code searches for triples of descending whole numbers less than 100 (an arbitrary choice).

```
def vectors_with_whole_number_length(max_coord=100):
    for x in range(1,max_coord):
        for y in range(1,x+1):
            for z in range(1,y+1):
                if length((x,y,z)).is_integer():
                    yield (x,y,z)
```

It finds 869 vectors with whole number coordinates and whole number lengths. The shortest of these is $(2, 2, 1)$, with length exactly 3 and the longest is $(99, 90, 70)$ with length exactly 150.

EXERCISE

Find a vector in the same direction as $(-1,-1,2)$ but which has length 1. Hint: find the appropriate scalar to multiply the original vector to change its length appropriately.

SOLUTION

The length of $(-1,-1,2)$ is about 2.45, so we'll have to take a scalar multiple of this vector with $(1/2.45)$ to make its length 1.

```
>>> length((-1,-1,2))
2.449489742783178
>>> s = 1/length((-1,-1,2))
>>> scale(s,(-1,-1,2))
(-0.4082482904638631, -0.4082482904638631, 0.8164965809277261)
>>> length(scale(s,(-1,-1,2)))
1.0
```

Rounding to the nearest hundredth in each coordinate, the vector is $(-0.41, -0.41, 0.82)$.

3.3 The dot product: measuring alignment of vectors

One kind of multiplication we've already seen for vectors is scalar multiplication, combining a scalar (a real number) and a vector to get a new vector. We haven't yet talked about any ways to multiply one vector with another. It turns out there are two important ways to do this, and they both give important geometric insights. One is called the dot product, and we write it with a dot operator like " $u \cdot v$ ", while the other is called the cross product, written " $u \times v$ ".

When we multiply numbers, these notations mean the same thing. For vectors, they are entirely different. In fact, the dot product takes two vectors and returns a scalar, while the cross product takes two vectors and returns another vector. Both, however, are operations that will help us reason about lengths and directions of vectors in 3D. Let's start by focusing on the dot product.

3.3.1 Picturing the dot product

The dot product (also called the inner product) is an operation on two vectors that returns a scalar. In other words, given two vectors u and v , the result of $u \cdot v$ is a real number. The dot product works on vectors in 2D, 3D, or any number of dimensions. You can think of it as measuring "how aligned" the pair of input vectors are. Let's first look at some vectors in the x,y plane and show their dot products to give you some intuition.

These two vectors u and v have lengths 4 and 5 respectively, and they point in nearly the same direction. Their dot product is positive, meaning they are aligned.

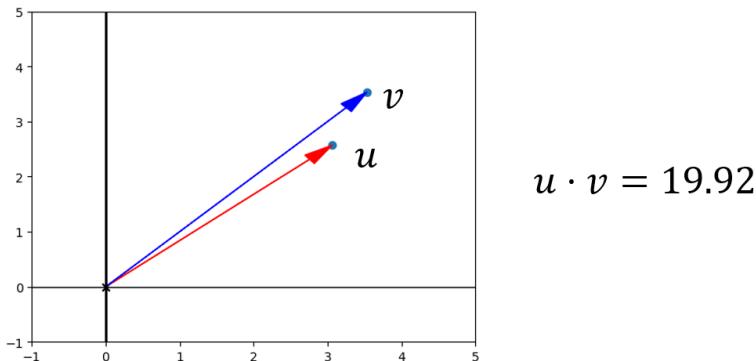


Figure 3.25 Two vectors that are relatively aligned give a large, positive dot product.

Two vectors that are pointing in similar directions will have a positive dot product, and the larger the vectors the larger the product. Smaller vectors that are similarly aligned will have a smaller -- but still positive -- dot product. These new vectors u and v both have length 2:

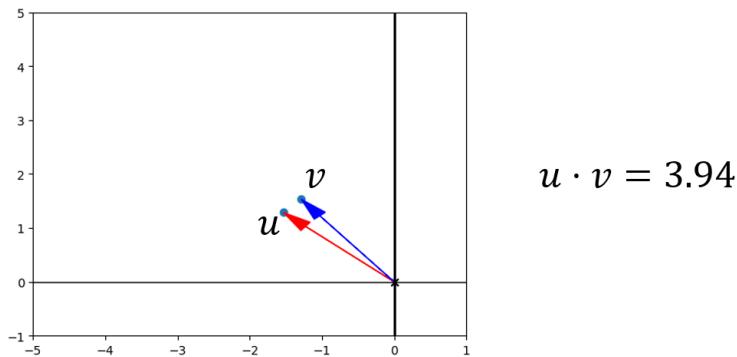


Figure 3.26 Two shorter vectors pointing in similar directions give a smaller but still positive dot product.

By contrast, if two vectors point in opposite or near opposite directions, their dot product will be negative. The bigger the magnitude of the vectors, the more negative their dot product will be.

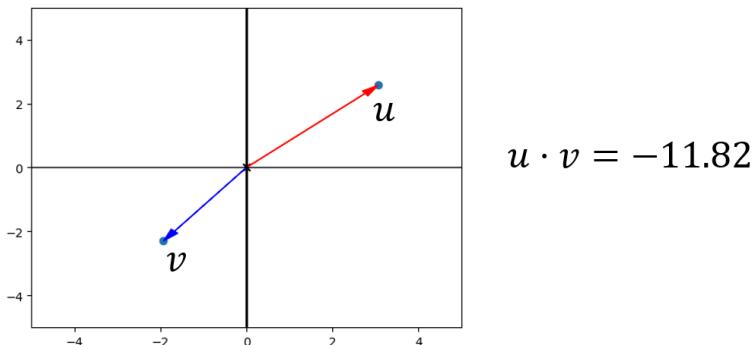


Figure 3.27 Vectors pointing in opposite directions will have a negative dot product.

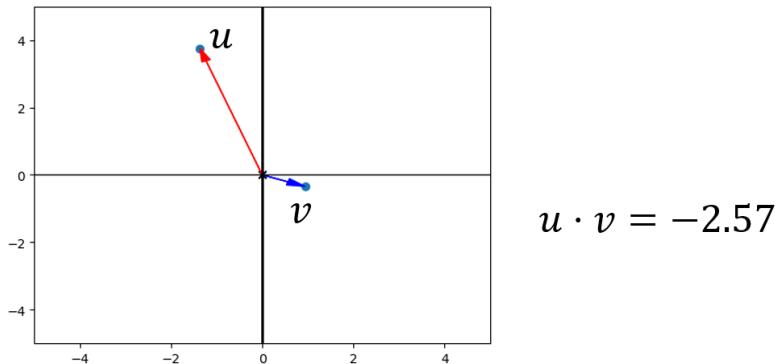
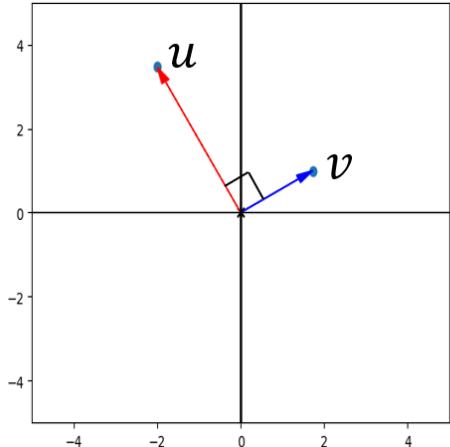


Figure 3.28 Shorter vectors pointing in opposite directions will have a smaller but still negative dot product.

Not all pairs of vectors clearly point in similar or opposite directions, and the dot product detects this. If two vectors point in exactly perpendicular directions, their dot product is zero regardless of their lengths.



$$u \cdot v = 0$$

Figure 3.29 Perpendicular vectors always have dot product equal to zero.

This turns out to be one of the most important applications of the dot product: it lets us compute whether two vectors are perpendicular without doing any trigonometry. This perpendicular case also serves to separate the other cases: if the angle between two vectors is less than 90 degrees, they will have a positive dot product. If it is greater than 90 degrees, they will have a negative dot product. While I haven't yet told you how to compute a dot product, you now know how to interpret the value. We'll move on to computing it next.

3.3.2 Computing the dot product

Given coordinates for two vectors, there's a simple recipe to compute the dot product: multiply the corresponding coordinates and then add up the products.

For instance in the dot product $(1,2,-1) \cdot (3, 0, 3)$ the product of x coordinates is 3, the product of y coordinates is 0, and the product of z coordinates is -3. The sum is $3 + 0 + (-3) = 0$, so the dot product is zero. If I kept my promise, these two vectors should be perpendicular. Drawing them proves this, but you have to look at them from the right direction!

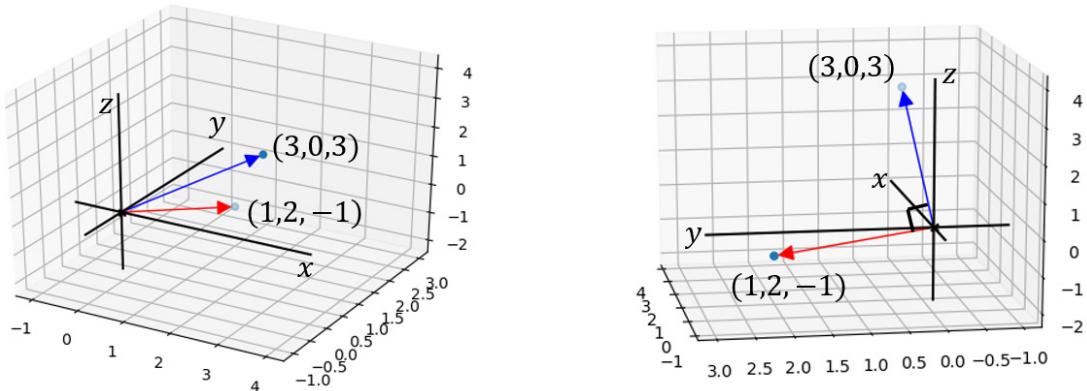
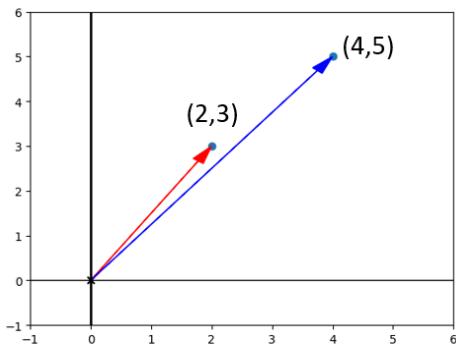


Figure 3.30 Using the dot product to detect whether two 3D vectors are perpendicular.

Our perspective can be misleading in 3D, making it all the more valuable to be able to compute relative directions rather than eyeballing them.

As another example, the vectors $(2,3)$ and $(4,5)$ lie in similar directions in the x,y plane. The product of x coordinates is $2 * 4 = 8$ while the sum of y coordinates is $3 * 5 = 15$. The sum $8 + 15 = 23$ is the dot product. As a positive number this confirms the vectors are separated by less than 90 degrees. These vectors have the same relative geometry whether we consider them as 2D vectors or as the 3D vectors $(2,3,0)$ and $(4,5,0)$ that happen to lie in the plane where $z = 0$.

$$(2,3) \cdot (4,5) = 2 \cdot 4 + 3 \cdot 5 = 23$$



$$(2,3,0) \cdot (4,5,0) = 2 \cdot 4 + 3 \cdot 5 + 0 \cdot 0 = 23$$

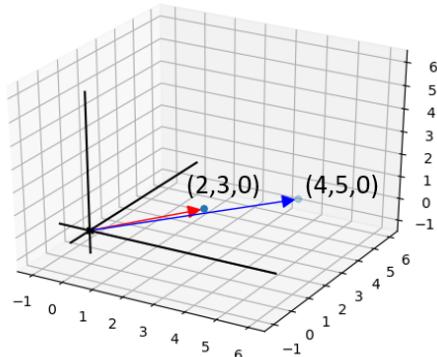


Figure 3.31 Another example of computing a dot product.

In Python, we can write a dot product function that handles any pair of input vectors as long as they have a matching number of coordinates.

```
def dot(u,v):
```

```
    return sum([coord1 * coord2 for coord1,coord2 in zip(u,v)])
```

This code uses Python's zip to pair the appropriate coordinates, multiplies each pair in a comprehension, and then adds them up. Let's use this to explore how the dot product behaves some more.

3.3.3 Dot products by example

It's not surprising that two vectors lying on different axes have zero dot product, we know they are perpendicular:

```
>>> dot((1,0),(0,2))
0
>>> dot((0,3,0),(0,0,-5))
0
```

We can also confirm that longer vectors give longer dot products. For instance, scaling either input vector by a factor of 2 doubles the output of the dot product.

```
>>> dot((3,4),(2,3))
18
>>> dot(scale(2,(3,4)),(2,3))
36
>>> dot((3,4),scale(2,(2,3)))
36
```

It turns out the dot product is proportional to each of the lengths of its input vectors. If you take the dot product of two vectors in the same direction, the dot product is precisely equal to the product of the lengths. For instance, (4,3) has length 5 and (8,6) has length 10. The dot product is equal to 5×10 .

```
>>> dot((4,3),(8,6))
50
```

Of course, the dot product is not always equal to the product of the lengths of its inputs. The vectors (5,0), (-3,4), (0,-5), and (-4,-3) all have the same length of 5 but different dot products with the original vector (4,3).

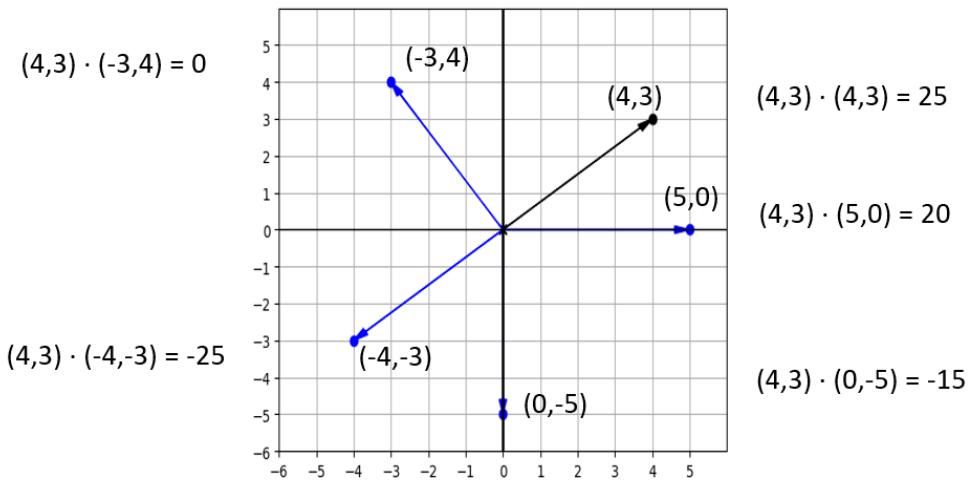


Figure 3.32 Vectors of the same length have different dot products with the vector $(4,3)$ depending on their direction.

The dot product of two vectors of length 5 ranges from $5 \cdot 5 = 25$ when they are aligned down to -25 when they are in opposite directions. In the next exercise, convince yourself that the dot product of two vectors can range from the product of the lengths down to the opposite of that value.

3.3.4 Measuring angles with the dot product

I introduced vector products because I claimed they'd help us measure angles between vectors, and we're almost there. We can see that the dot product $u \cdot v$ ranges from 1 to -1 times the product of the lengths of u and v as the angle ranges from 0 to 180 degrees. We've already seen a function that behaves that way, the cosine:

It turns out that the dot product has an alternate formula. Where $|u|$ and $|v|$ denote the lengths of vectors u and v , the dot product is given by:

$$u \cdot v = |u| \cdot |v| \cdot \cos(\theta)$$

Where θ is the angle between the vectors u and v . In principle this gives us a new way to compute a dot product. We could measure the lengths of two vectors and measure the angle between them to get the result. Suppose we knew the length of two vectors to be 3 and 2 respectively, and using our protractor discovered that they were 75 degrees apart:

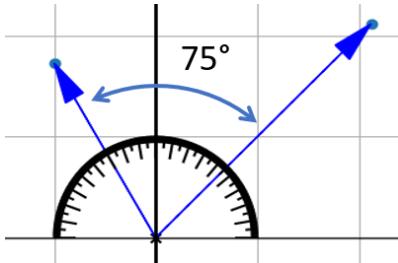


Figure 3.33 Two vectors of lengths 3 and 2, respectively, at 75 degrees apart.

Their dot product would be $3 \cdot 2 \cdot \cos(75^\circ)$. With the appropriate conversion to radians, we can compute this in Python to be about 1.55.

```
>>> from math import cos,pi
>>> 3 * 2 * cos(75 * pi / 180)
1.5529142706151244
```

When doing computations with vectors, it's more common that you'll start with coordinates and need to compute angles from them. We can combine both of our formulas to recover an angle: first we compute the dot product and lengths using coordinates, then we solve for the angle.

Let's try to find the angle between the vectors $(3,4)$ and $(4,3)$. Their dot product is 24 and each of their lengths is 5. Our new dot product formula tells us that

$$(3,4) \cdot (4,3) = 24 = 5 \cdot 5 \cdot \cos(\theta) = 25 \cdot \cos(\theta)$$

From $24 = 25 \cdot \cos(\theta)$ we can simplify to $\cos(\theta) = 24/25$. Using Python's `math.acos`, we find that a θ value of 0.284 radians or 16.3 degrees gives us a cosine of 24/25.

This exercise reminds us why we didn't need the dot product in 2D: we already found a formula to get the angle of a vector. Using that creatively, we could quickly find any angles we wanted in the plane. The dot product really starts to shine in 3D, where a change of coordinates can't help us as much.

For instance, we can use the same formula to find the angle between $(1,2,2)$ and $(2,2,1)$. The dot product is $1 \cdot 2 + 2 \cdot 2 + 2 \cdot 1 = 8$ and the lengths are both 3. This means $8 = 3 \cdot 3 \cdot \cos(\theta)$, so $\cos(\theta) = 8/9$ and $\theta = 0.476$ radians or 27.3 degrees.

This process is the same in 2D or 3D, and it's one we'll use over and over. We can save some future effort by implementing a Python function to find the angle between two vectors. Since neither our dot function nor our length function have a hard-coded number of dimensions, this new function won't either. We can make use of the fact that $u \cdot v = |u| \cdot |v| \cdot \cos(\theta)$ and therefore that

$$\cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}| \cdot |\mathbf{v}|}$$

and

$$\theta = \arccos\left(\frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}| \cdot |\mathbf{v}|}\right)$$

This formula translates neatly to Python code as follows.

```
def angle_between(v1,v2):
    return acos(
        dot(v1,v2) /
        (length(v1) * length(v2))
    )
```

Nothing in this Python code depends on the number of dimensions of the vectors \mathbf{v}_1 and \mathbf{v}_2 . They could both be tuples of 2 coordinates or they could both be tuples of 3 coordinates (or, in fact, tuples of four or more coordinates, which we'll discuss shortly). By contrast, the next vector product we will meet only works in three dimensions.

3.3.5 Exercises

EXERCISE

Based on the picture below, rank $\mathbf{u} \cdot \mathbf{v}$, $\mathbf{u} \cdot \mathbf{w}$, and $\mathbf{v} \cdot \mathbf{w}$ from largest to smallest.

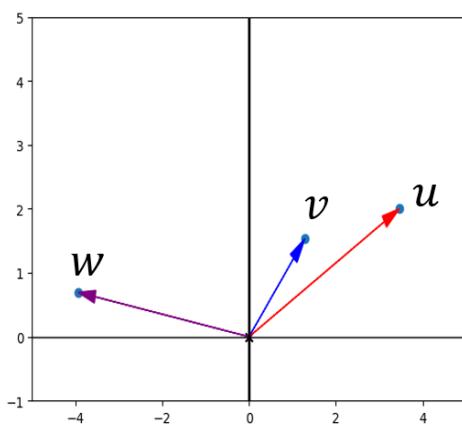


Figure 3.34 Decide which of the dot products is the smallest.

SOLUTION

The product $u \cdot v$ is the only positive dot product, since u and v are the only pair with less than a right angle between them. Further, $u \cdot w$ is smaller (more negative) than $v \cdot w$ since u is both bigger and further from w . So $u \cdot v > v \cdot w > u \cdot w$.

EXERCISE 13

What is the dot product of $(-1, -1, 1)$ and $(1, 2, 1)$? Are these two 3D vectors separated by more than 90 degrees, less than 90 degrees, or exactly 90 degrees?

SOLUTION

$(-1, -1, 1)$ and $(1, 2, 1)$ have dot product $-1 * 1 + -1 * 2 + 1 * 1 = -2$. Since this is a negative number, the two vectors are more than 90 degrees apart.

MINI PROJECT

This example shows that for two 3D vectors u and v , the values of $(2u) \cdot v$ and $u \cdot (2v)$ are both equal to $2(u \cdot v)$. In this case $u \cdot v = 18$ and both $(2u) \cdot v$ and $u \cdot (2v)$ are 36, twice the original result. Show that this works for any real number s , not just 2. In other words, show that for any s the values of $(su) \cdot v$ and $u \cdot (sv)$ are both equal to $s(u \cdot v)$.

SOLUTION

Let's name the coordinates of u and v , say $u = (a, b, c)$ and $v = (d, e, f)$. Then $u \cdot v = ad + be + cf$. Since $su = (sa, sb, sc)$ and $sv = (sd, se, sf)$, we can show both of the results by expanding the dot products.

write out the coordinates

$$\begin{aligned}
 (su) \cdot v &= (sa, sb, sc) \cdot (d, e, f) && \text{do the dot product} \\
 &= sad + sbe + scf \\
 &= s(ad + be + cf) \\
 &= s(u \cdot v)
 \end{aligned}$$

factor out s ; recognize the original dot product

Figure 3.35 Proving that scalar multiplication scales the result of the dot product accordingly.

And the other product works the same way:

$$\begin{aligned}
 u \cdot (sv) &= (a, b, c) \cdot (sd, se, sf) \\
 &=asd + bse + csf \\
 &= s(ad + be + cf) \\
 &= s(u \cdot v)
 \end{aligned}$$

Figure 3.36 Proving the same fact holds for the second vector input to the dot product.

MINI PROJECT

Explain algebraically why the dot product of a vector with itself is the square of its length.

SOLUTION

If a vector has coordinates (a,b,c) then the dot product with itself is $a^2 + b^2 + c^2$. Its length is $\sqrt{a^2 + b^2 + c^2}$, so this is indeed the square.

MINI PROJECT

Find a vector u of length 3 and a vector v of length 7, such that $u \cdot v = 21$. Find another pair of vectors u and v such that $u \cdot v = -21$. Finally, find three more pairs of vectors of respective lengths 3 and 7 and show that all of their lengths lie between -21 and 21.

SOLUTION

Two vectors in the same direction, for instance along the positive x axis, will have the highest possible dot product.

```
>>> dot((3,0),(7,0))
21
```

Two vectors in the opposite direction, for instance the positive and negative y directions, will have the lowest possible dot product.

```
>>> dot((0,3),(0,-7))
-21
```

Using polar coordinates, we can easily generate some more vectors of length 3 and 7 with random angles.

```

from vectors import to_cartesian
from random import random
from math import pi

def random_vector_of_length(l):
    return to_cartesian((l, 2*pi*random()))

pairs = [(random_vector_of_length(3), random_vector_of_length(7))
          for i in range(0,3)]
```

```

for u,v in pairs:
    print("u = %s, v = %s" % (u,v))
    print("length of u: %f, length of v: %f, dot product :%f" %
          (length(u), length(v), dot(u,v)))

```

EXERCISE

Let u and v be vectors, with $|u| = 3.61$ and $|v| = 1.44$. If the angle between u and v is 101.3 degrees, what is $u \cdot v$?

- a.) 5.198
- b.) 5.098
- c.) -1.019
- d.) 1.019

SOLUTION

Again, we can plug these values into the new dot product formula and, with the appropriate conversion to radians, evaluate the result in Python.

```

>>> 3.61 * 1.44 * cos(101.3 * pi / 180)
-1.0186064362303022

```

Rounding to three decimal places, the answer agrees with c.

MINI PROJECT

Find the angle between (3,4) and (4,3) by converting them to polar coordinates and taking the difference of the angles.

- a.) 1.569
- b.) 0.927
- b.) 0.643
- d.) 0.284

(Hint: The result should agree with the value from the dot product formula.)

SOLUTION

The vector (3,4) is further from the positive x axis counterclockwise, so we subtract the angle of (4,3) from the angle of (3,4) to get our answer. It matches exactly.

```

>>> from vectors import to_polar
>>> r1,t1 = to_polar((4,3))
>>> r2,t2 = to_polar((3,4))
>>> t1-t2
-0.2837941092083278
>>> t2-t1
0.2837941092083278

```

EXERCISE

What is the angle between $(1,1,1)$ and $(-1,-1,1)$ in degrees?

- a.) 180°
- b.) 120°
- c.) 109.5°
- d.) 90°

SOLUTION

The lengths of both vectors are $\sqrt{3}$, or approximately 1.732. Their dot product is $1*(-1) + 1*(-1) + 1*1 = -1$.

So $-1 = \sqrt{3} \cdot \sqrt{3} \cdot \cos(\theta)$ and therefore $\cos(\theta) = -1/3$. This makes the angle approximately 1.911 radians or 109.5 degrees (answer c).

3.4 The cross product: measuring oriented area

As previously introduced, the cross product takes two 3D vectors u and v as inputs and its output $u \times v$ is another 3D vector. It is similar to the dot product in that the lengths and relative directions of the input vectors determine the output, but is different in that the output has not only a magnitude but also a direction. We need to think carefully about the concept of direction in 3D to understand the power of the cross product.

3.4.1 Orienting ourselves in 3D

When I introduced the x , y , and z axes at the beginning of the chapter, I made two clear assertions. First, I promised that the familiar x,y plane exists within the 3D world. Second, I set the z direction to be perpendicular to the x,y plane, with the x,y plane living where $z = 0$. What I didn't announce clearly was that the positive z direction was up instead of down.

In other words, if we looked at the x,y plane from the usual perspective would see the positive z axis emerging out of the plane toward us. The other choice we could have made was sending the positive z axis away us.

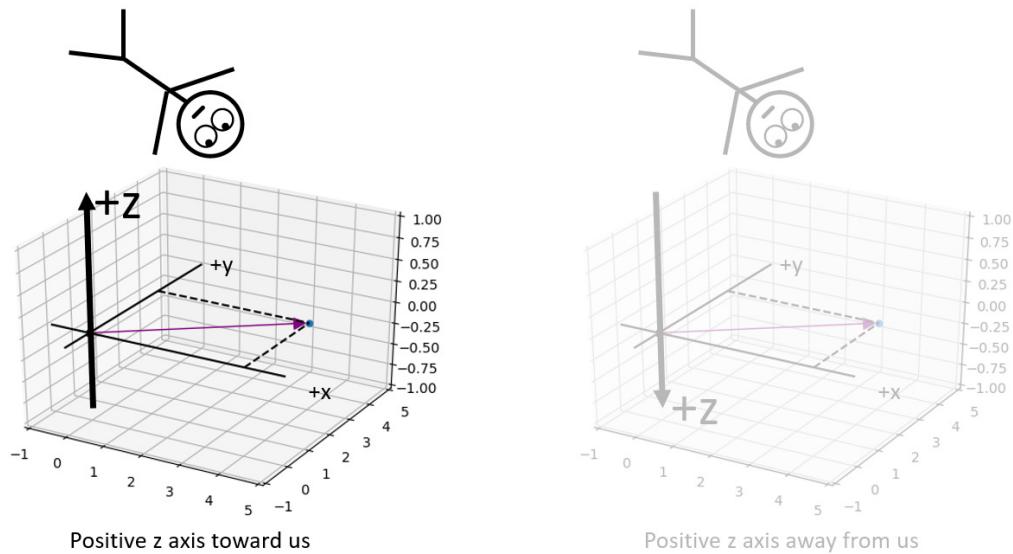


Figure 3.37 Positioning ourselves in 3D to see the x,y plane as we saw it in chapter 2. When looking at the x,y plane, we chose the positive z -axis to point toward us as opposed to away from us.

The difference here is not a matter of perspective; the two choices here are called different orientations of 3D space, and they are distinguishable from any perspective. Suppose we are floating at some positive z coordinate like the stick figure on the left above. We should see the positive y -axis positioned a quarter-turn counterclockwise from the positive x -axis. Otherwise, the axes are arranged in the wrong orientation.

Plenty of things in the real world have orientations, and don't look identical to their mirror images. For instance, left and right shoes have identical size and shape but different orientations. A plain coffee mug does not have an orientation: we can not look at two pictures of an unmarked coffee mug and decide if they are different. But two mugs with graphics on opposite sides are distinguishable.

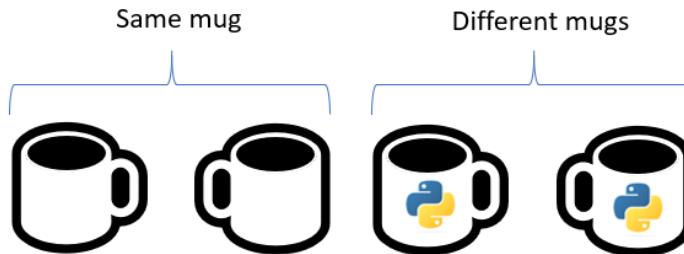


Figure 3.38 A mug with no image is the same object as its mirror image. A mug with an image on one side is *not* the same as its mirror image.

The readily-available object most mathematicians use to detect orientation is a hand. Our hands are oriented objects, so we can right hands from left hands even if they were unluckily detached from our bodies. Can you tell if this hand is a right or left hand?



Figure 3.39 Is this a right or left hand?

Clearly it's a right hand: we don't have fingernails on our left-hand fingertips! Mathematicians use their hands to distinguish the two possible orientations of coordinate axes, and they call the two possibilities "right-handed" and "left-handed" orientations. Here's the rule: if you point your right index finger along the positive x axis and curl your remaining fingers toward the positive y axis, your thumb tells you the direction of the positive z axis.

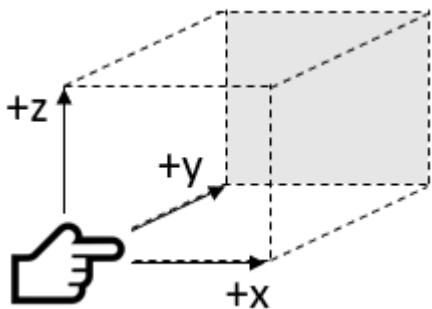


Figure 3.40 The right-hand rule helps us remember the orientation we've chosen.

This is called the "right-hand rule", and if it agrees with your axes then you are (correctly!) using the right-handed orientation.

Orientation matters! If you are writing a program to steer a drone or control a laparoscopic surgery robot, you need to keep your ups, downs, lefts, rights, forwards, and backwards consistent. The cross product is an oriented machine; if we're careful choosing the orientation of its output, it can help us keep track of orientation throughout all of our computations.

3.4.2 Finding the direction of the cross product

Again, before I tell you how to compute the cross product I want to show you what it looks like. Given two input vectors, the cross product outputs a result that is perpendicular to both of them. For instance, if $u = (1,0,0)$ and $v = (0,1,0)$ then it happens that the cross product $u \times v$ is $(0,0,1)$.

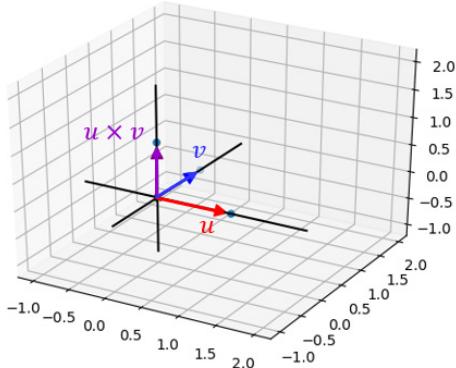


Figure 3.41 The cross product of $u = (1,0,0)$ and $v = (0,1,0)$.

In fact, any two vectors in the x,y plane will have a cross product that lies along the z axis.

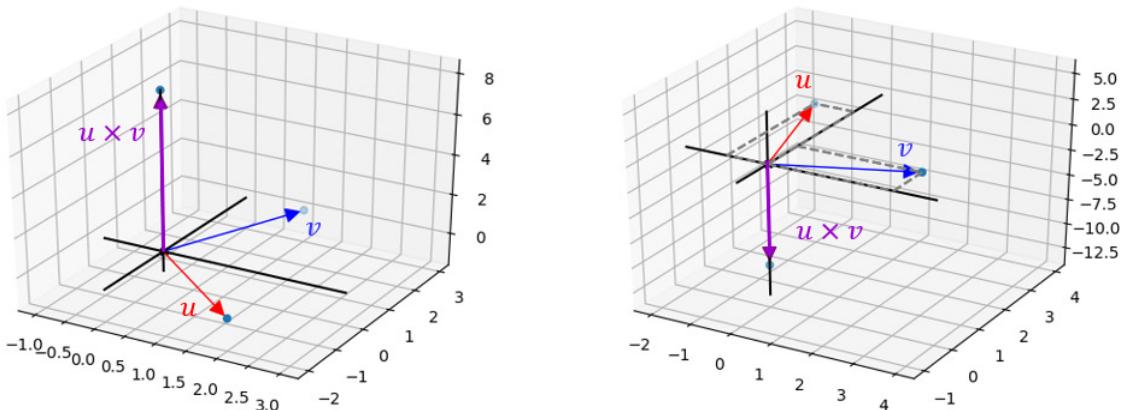


Figure 3.42 The cross product of any two vectors in the x,y plane lies on the z axis.

This makes it clear why the cross product doesn't work in 2D: it returns vectors that lie outside of the plane of its inputs. We can see the output of the cross product is perpendicular to both inputs even if they don't lie in the x,y plane.

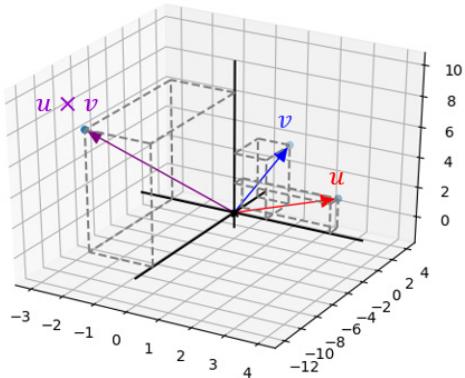


Figure 3.43 The cross product always returns a vector which is perpendicular to both inputs.

But there are two possible perpendicular directions, and the cross product selects only one. For instance, the result of $(1,0,0) \times (0,1,0)$ happens to be $(0,0,1)$, pointing in the positive z direction. Any vector on the z axis, positive or negative, would be perpendicular to both of these inputs. Why does the result point in the positive direction?

Here's where orientation comes in: the cross product obeys the right-hand rule as well. Once you've found the direction perpendicular to two input vectors u and v , the cross product $u \times v$ lies in a direction that puts the three vectors u , v , and $u \times v$ in a right-handed configuration. That is, we can point our right index finger in the direction of u , curl our other fingers toward v , and our thumb will point in the direction of $u \times v$.

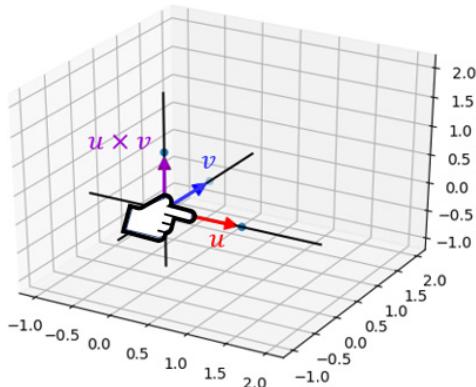


Figure 3.44 The right-hand rule tells us *which* perpendicular direction the cross product points toward.

When input vectors lie on two coordinate axes, it's not too hard to find the exact direction their cross product will point: it will be one of the two directions along the remaining axis. In

general, it's hard to come up with an exact perpendicular direction to two vectors without computing the cross product. This is one of the features that will make it so useful once we've seen how to compute it. But a vector doesn't just specify a direction; it also specifies a length. The length of the cross product encodes useful information as well.

3.4.3 Finding the length of the cross product

Like the dot product, the length of the cross product is a number that gives us information about the relative position of the input vectors. Instead of measuring how aligned two vectors are, it tells us something closer to "how perpendicular they are." More precisely, it tells us how big of an area its two inputs span.

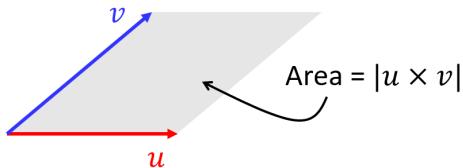


Figure 3.45 The length of the cross product is equal to the area of a parallelogram.

The parallelogram bounded by u and v as above has an area which is the same as the length of the cross product $u \times v$. For any pair of lengths that two input vectors could have, they will span the most area if they are perpendicular. On the other hand, if u and v are in the same direction, they don't span any area; the cross product will have zero length. This is convenient: we couldn't choose a unique perpendicular direction if the two input vectors were parallel.

Paired with the direction of the result, the length of the result gives us an exact vector. Two vectors in the plane are guaranteed to have a cross product pointing in the $+z$ or $-z$ direction. We can see that the bigger the parallelogram that the plane vectors span, the longer their cross product will be.

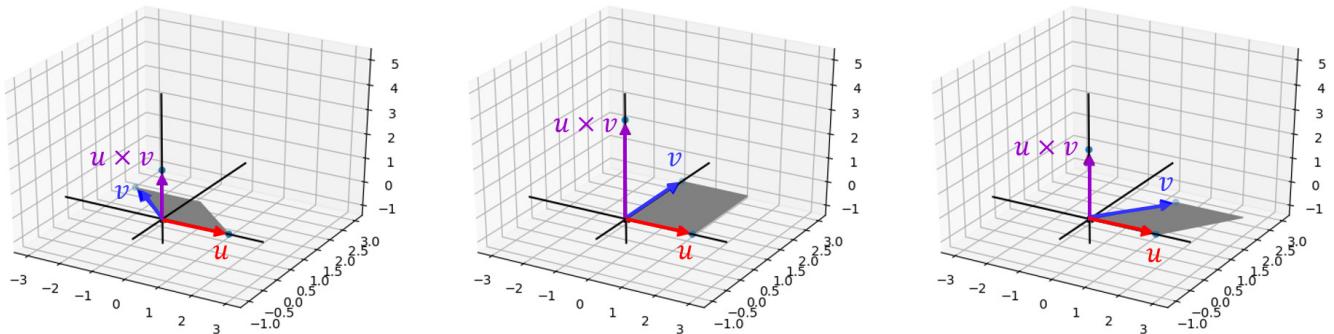


Figure 3.46 Pairs of vectors in the x,y plane have cross products of different sizes based on the area of the parallelogram they span.

There's a trigonometric formula for the area of this parallelogram. If u and v are separated by an angle θ , the area is $|u| \cdot |v| \cdot \sin(\theta)$.

We can put the length and direction together to see some simple cross products. For instance, what is the cross product of $(0,2,0)$ and $(0,0,-2)$? These vectors lie on the y and z axes respectively, so to be perpendicular to both, the cross product must lie on the x axis. Let's find the direction of the result using the right hand rule. Pointing in the direction of the first vector with our index finger (the positive y direction) and bending our fingers in the direction of the second vector (the negative z direction), we find our thumb is in the negative x direction. The magnitude of the cross product will be $2 * 2 * \sin(90^\circ)$ since the y and z axes meet at a 90 degree angle. (The parallelogram happens to be a square in this case, having side length two). This comes out to four, so the result is $(-4,0,0)$, a vector of length 4 in the $-x$ direction.

It's nice to convince ourselves that the cross product is a well-defined operation by computing it geometrically. But that's not practical in general, when vectors don't always lie on an axis and it's not obvious what coordinates you need to find a perpendicular result. Fortunately, there's an explicit formula for the coordinates of the cross product in terms of the coordinates of its inputs.

3.4.4 Computing the cross product of 3D vectors

The formula for the cross product looks hairy at first glance, but we can quickly wrap it in a Python function and compute it with no sweat. Let's start with coordinates for two vectors u and v . We could name the coordinates $u = (a,b,c)$ and $v = (d,e,f)$ but it's clearer if we use better symbols: $u = (u_x, u_y, u_z)$ and $v = (v_x, v_y, v_z)$. It's easier to remember that the number called v_x is the x coordinate of v than if we called it the arbitrary letter "d." In terms of these coordinates, the formula for the cross product is

$$u \times v = (u_y v_z - u_z v_y, u_z v_x - v_z u_x, u_x v_y - u_y v_x)$$

Or, in Python:

```
def cross(u, v):
    ux,uy,uz = u
    vx,vy,vz = v
    return (uy*vz - uz*vy, uz*vx - vx*u_z, ux*vy - uy*vx)
```

You can test-drive this formula in the exercises. Note that in contrast to most of the formulas we've used so far, this one doesn't appear to generalize well to other dimensions; it requires that the input vectors have exactly three components.

This algebraic procedure agrees with the geometric description we built up in this chapter. Because it tells us area and direction, the cross product will help us decide whether an occupant of 3D space would see a polygon floating in space with them. For instance, an observer standing on the x axis would not see the parallelogram spanned by $u = (1,1,0)$ and $v = (-2,1,0)$:

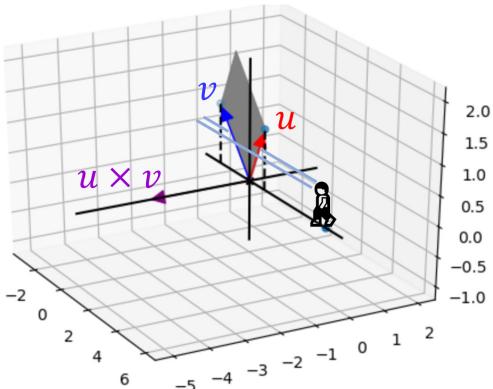


Figure 3.47 The cross product can indicate whether a polygon is visible to an observer.

In other words, this polygon is parallel to the observer's line of sight. Using the cross product, we could tell this without drawing the picture. Because the cross product is perpendicular to the person's line of sight, none of the polygon will be visible.

Now it's time for our culminating project: building a 3D object out of polygons and drawing it on a 2D canvas. We'll use all of the vector operations we've seen so far. In particular, the cross product will help us decide which polygons are visible.

3.4.5 Exercises

EXERCISE

Each of the following diagrams show three mutually perpendicular arrows indicating positive x, y, and z directions. A 3D box is shown for perspective, with the back of the box colored gray. Which of the four are compatible with the one we chose? That is, which show the x, y, and z axes as we've been drawing them, even if from a different perspective?

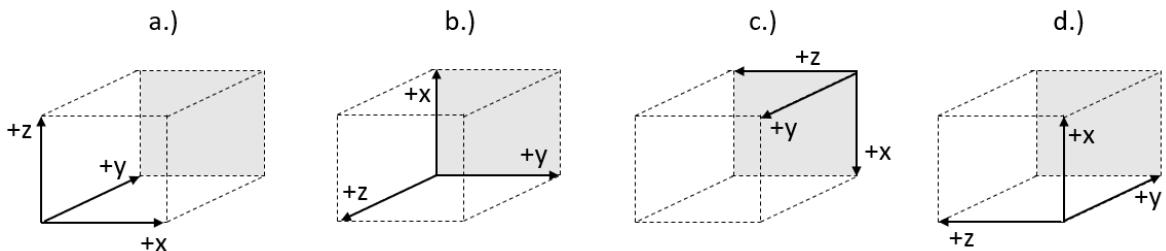


Figure 3.48 Which of these axes agrees with our orientation convention?

SOLUTION

Looking down on diagram a from above, we'd see the x and y axis as usual, with the z axis pointing toward us. So diagram a agrees with our orientation.

In diagram b the z axis is coming toward us, while the +y direction is 90 degrees clockwise from the +x direction. This does not agree with our orientation.

If we looked at diagram c from a point in the positive z direction (from the left side of the box) we would see the +y direction 90 degrees counterclockwise from the +x direction, so it agrees with our orientation.

Looking at diagram d from the left of the box, the +z direction would be toward us, and the +y direction would again be counterclockwise from the +x direction. This agrees with our orientation as well.

EXERCISE

If you held up three coordinate axes in front of a mirror, would the image in the mirror have the same orientation or a different one?

SOLUTION

The mirror image has reversed orientation. From this perspective, the z and y axes stay pointing the same direction. The x axis is clockwise from the y axis in the original, but in the mirror image it moves to counterclockwise.

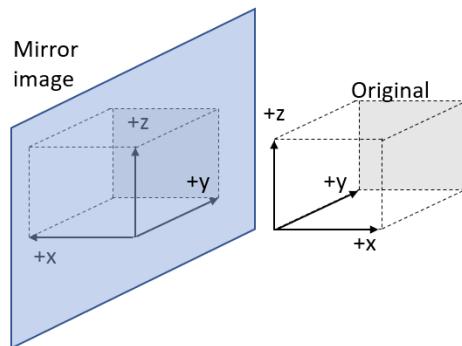


Figure 3.49 The x, y, and z axes and their mirror image.

EXERCISE

What direction does the result of $(0,0,3) \times (0,-2,0)$ point?

SOLUTION

If we point our right index finger in the direction of $(0,0,3)$, the positive z direction, and curl our other fingers in the direction of $(0,-2,0)$, the negative y direction, our thumb points in the positive x direction. Therefore, $(0,0,3) \times (0,-2,0)$ points in the positive x direction.

EXERCISE

What are the coordinates of the cross product of $(1,-2,1)$ and $(-6, 12, -6)$?

SOLUTION

As negative scalar multiples of one another, these vectors point in opposite directions and they don't span any area. The length of the cross product is therefore zero. The only vector of length zero is $(0,0,0)$, so that is the answer.

MINI PROJECT

The area of a parallelogram is equal to the length of its base times its height:

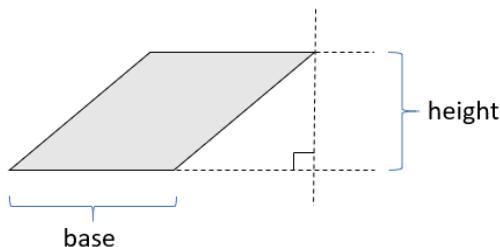


Figure 3.50 The area of a parallelogram is its base times its height.

Given that, explain why the formula $|u| \cdot |v| \cdot \sin(\theta)$ makes sense.

SOLUTION

In the diagram, the vector u defines the base, so the base length is $|u|$. From the tip of v to the base, we can draw a right triangle. The length of v is the hypotenuse, and the vertical leg of the triangle is the height we are looking for. By the definition of the sine function, the height is $|v| \cdot \sin(\theta)$.

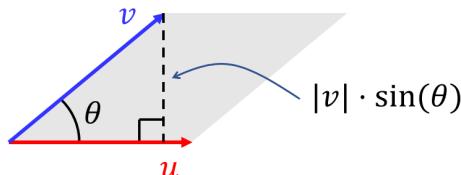


Figure 3.51 The formula for the area of a parallelogram in terms of the sine of one of its angles.

Since the base length is $|u|$ and the height is $|v| \cdot \sin(\theta)$, the area of the parallelogram is indeed $|u| \cdot |v| \cdot \sin(\theta)$.

EXERCISE

What is the result of the cross product $(1,0,1) \times (-1,0,0)$?

- a.) $(0,1,0)$
- b.) $(0,-1,0)$

c.) (0,-1,-1)

d.) (0,1,-1)

SOLUTION

These vectors lie in the x,z plane so their cross product lies on the y axis. Pointing our right index finger in the direction of (1,0,1) and curling our fingers toward (-1,0,0) requires our thumb to point in the -y direction.

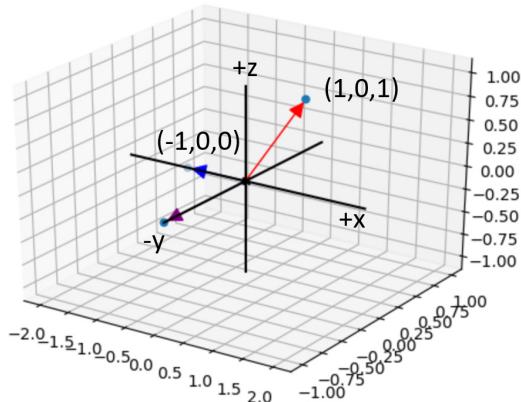


Figure 3.52 Computing the cross product of (1,0,1) and (-1,0,0) geometrically.

We could find the lengths of the vectors and the angle between them to get the size of the cross product, but we already have the base and height from the coordinates. They are both 1, so the length is 1. The cross product is therefore (0,-1,0), a vector of length 1 in the -y direction, and the answer is b.

EXERCISE

Use the Python `cross` function to compute $(0,0,1) \times v$ for a few different values of a second vector v . What is the z coordinate of each result, and why?

SOLUTION

No matter what vector v is chosen, the z coordinate will be zero.

```
>>> cross((0,0,1),(1,2,3))
(-2, 1, 0)
>>> cross((0,0,1),(-1,-1,0))
(1, -1, 0)
>>> cross((0,0,1),(1,-1,5))
(1, 1, 0)
```

Since $u = (0,0,1)$, both u_x and u_y are zero. This kills the term $u_x v_y - u_y v_x$ in the cross product formula regardless of the values v_x and v_y . Geometrically this makes sense: the cross product should be perpendicular to both inputs, and to be perpendicular to (0,0,1) the z-component must be zero.

MINI PROJECT

Show algebraically that $u \times v$ is perpendicular to both u and v regardless of the coordinates of u and v . (Hint: show that $(u \times v) \cdot u$ and $(u \times v) \cdot v$ by expanding them in coordinates).

SOLUTION

Let $u = (u_x, u_y, u_z)$ and $v = (v_x, v_y, v_z)$ below. We can write $(u \times v) \cdot u$ in terms of coordinates as follows:

$$(u \times v) \cdot u = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x) \cdot (u_x, u_y, u_z)$$

Figure 3.53 Expanding the dot product of a cross product.

After we expand the dot product, we see that there are 6 terms. Each of these cancels out with one of the others.

$$\begin{aligned} &= (u_y v_z - u_z v_y) u_x + (u_z v_x - u_x v_z) u_y + (u_x v_y - u_y v_x) u_z \\ &= u_y v_z u_x - u_z v_y u_x + u_z v_x u_y - u_x v_z u_y + u_x v_y u_z - u_y v_x u_z \end{aligned}$$

Figure 3.54 After fully expanding, all the terms cancel.

Since all of the terms cancel out, the result is zero. To save ink, I won't show the result of $(u \times v) \cdot v$ but the same thing happens: six terms appear and cancel each other out, resulting in zero. This means that $(u \times v)$ is perpendicular to both u and v .

3.5 Rendering a 3D object in 2D

Let's try using what we've learned to render a simple 3D shape called an octahedron. Whereas a cube has six faces, all of which are squares, an octahedron has eight faces, all of which are triangles. You can think of an octahedron as two four-sided pyramids stacked on top of each other. The skeleton of an octahedron looks like this:

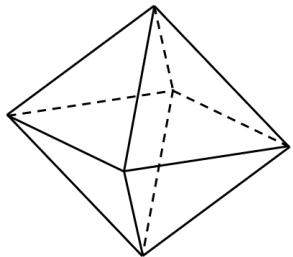


Figure 3.55 The skeleton of an octahedron, a shape with eight faces and six vertices.

The dotted lines show edges of the octahedron on the opposite side from us. If this were a solid, we wouldn't be able to see them. Instead, we'd see four of the eight triangular faces.

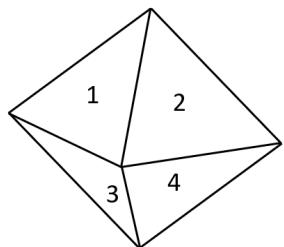


Figure 3.56 Four numbered faces of the octahedron that are visible to us in its current position.

Rendering the octahedron comes down to identifying the four triangles we need to show and shading them appropriately. Let's see how to do that.

3.5.1 Defining a 3D object with vectors

An octahedron is an easy example because it has only 6 corners, or vertices. We can give them simple coordinates: $(1,0,0)$, $(0,1,0)$, $(0,0,1)$ and their three opposite vectors.

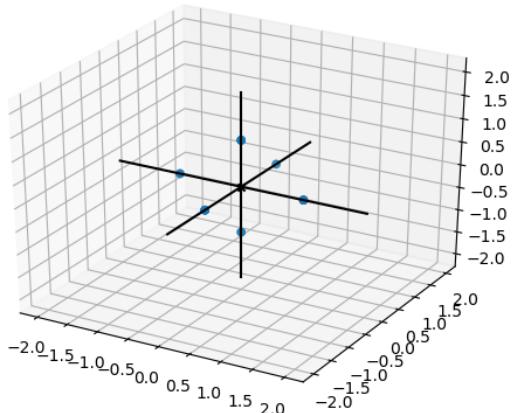


Figure 3.57 Vertices of an octahedron.

These six vectors define the boundaries of the shape, but not all the information we need to draw it. We'll also have to decide which of these vertices connect to form edges of the shape. For instance, the top point in the above diagram is $(0,0,1)$ and it connects by an edge to all four points in the x,y plane.

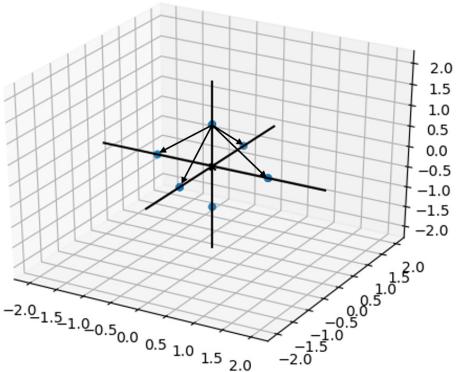


Figure 3.58 Four edges of the octahedron indicated by arrows.

These edges outline the top pyramid of the octahedron. Note that there is no edge from $(0,0,1)$ to $(0,0,-1)$ because that segment would lie within the octahedron, not on its outside. Each edge is defined by a pair of vectors: the start and end points of the edge as a line segment. For instance $(0,0,1)$ and $(1,0,0)$ define one of the edges.

Edges still aren't enough data to complete the drawing. We also need to know which triples of vertices and edges define triangular faces we want to fill with a solid, shaded color. Here's where orientation comes in: we will want to know not only which segments define faces of the octahedron, but also whether they face toward us or away from us.

Here's the strategy: we'll model a triangular face as three vectors, v_1 , v_2 , and v_3 defining its edges. Specifically, we'll order v_1 , v_2 , and v_3 such that $(v_2 - v_1) \times (v_3 - v_1)$ points outside the octahedron. If an outward-pointing vector is aimed toward us, it means the face is visible from our perspective. Otherwise, the face will be obscured, and we won't need to draw it.

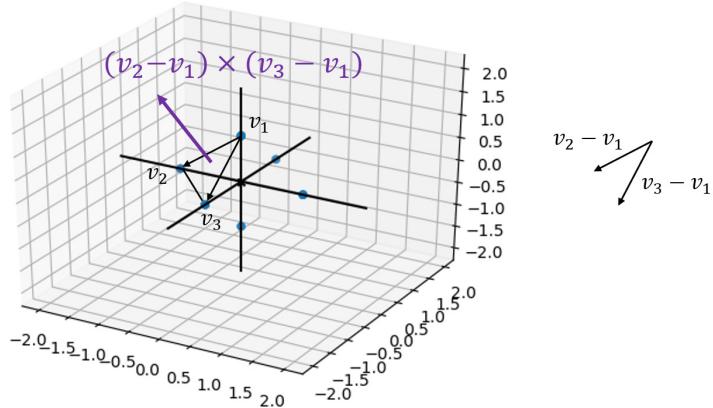


Figure 3.59 A face of the octahedron. The three points defining the face are ordered so that $(v_2 - v_1) \times (v_3 - v_1)$

points *outside* of the octahedron.

We can define the eight triangular faces as triples of three vectors v1, v2, and v3 as follows.

```
octahedron = [
    [(1,0,0), (0,1,0), (0,0,1)],
    [(1,0,0), (0,0,-1), (0,1,0)],
    [(1,0,0), (0,0,1), (0,-1,0)],
    [(1,0,0), (0,-1,0), (0,0,-1)],
    [(-1,0,0), (0,0,1), (0,1,0)],
    [(-1,0,0), (0,1,0), (0,0,-1)],
    [(-1,0,0), (0,-1,0), (0,0,1)],
    [(-1,0,0), (0,0,-1), (0,-1,0)],
]
```

The faces are actually the only data we need to render the shape; they contain the edges and vertices implicitly. For instance, we can get the vertices back from the faces with the following function:

```
def vertices(faces):
    return list(set([vertex for face in faces for vertex in face]))
```

3.5.2 Projecting to 2D

To turn 3D points into 2D points, we must choose what 3D direction we are observing from. Once we have two 3D vectors defining “up” and “right” from our perspective, we can *project* any 3D vector onto them and get two components instead of three. The component function below extracts the part of any 3D vector pointing in a given direction, using the dot product.

```
def component(v,direction):
    return (dot(v,direction) / length(direction))
```

With two directions hard-coded, in this case (1,0,0) and (0,1,0) we can establish a way to project from three coordinates down to two. This function takes a 3D vector, or a tuple of three numbers, and returns a 2D vector, or a tuple of two numbers.

```
def vector_to_2d(v):
    return (component(v,(1,0,0)), component(v,(0,1,0)))
```

We can picture this as “flattening” the 3D vector into the plane; deleting the z-component takes away any depth the vector had.

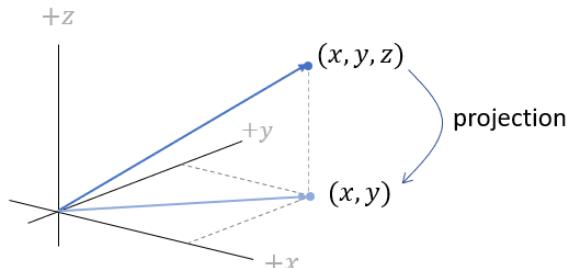


Figure 3.60 Deleting the z-component of a 3D vector flattens it into the x,y-plane.

Finally, to take a triangle from 3D to 2D, we need only apply this to all of the vertices defining a face.

```
def face_to_2d(face):
    return [vector_to_2d(vertex) for vertex in face]
```

3.5.3 Orienting faces and shading

To shade our 2D drawing, we pick a fixed color for each triangle according to how much it faces a given light source. Let's say our light source lies at a vector of $(1,2,3)$ from the origin. Then the brightness of a triangular face will be decided by "how perpendicular" it is to the light. Another way to measure this is by how aligned a perpendicular vector to the face is with the light source.

We don't have to think about computing colors: matplotlib has a built in library to do that for us. For instance:

```
blues = matplotlib.cm.get_cmap('Blues')
```

gives us a function called "blues" which maps numbers from 0 to 1 onto a spectrum of darker to brighter blue values. So our task is to find a number from 0 to 1 that indicates how bright a face should be.

With a vector perpendicular (or "normal") to each face and a vector pointing to the light source, their dot product will tell us how aligned they are. Moreover, since we're only considering directions, we can choose vectors with length 1. Then, if the face is pointing toward the light source at all, the dot product will lie between 0 and 1. If it is further than 90 degrees from the light source, it will not be illuminated at all.

This helper function takes a vector and returns another in the same direction but with length 1.

```
def unit(v):
    return scale(1./length(v), v)
```

This second helper function takes a face and gives us a vector perpendicular to it.

```
def normal(face):
    return(cross(subtract(face[1], face[0]), subtract(face[2], face[0])))
```

Putting it all together, we have a function that draws all the triangles we need to render a 3D shape using our `draw` function. (I've renamed `draw` to `draw2d`, and renamed classes accordingly to distinguish them from their 3D counterparts.)

```
def render(faces, light=(1,2,3), color_map=blues, lines=None):
    polygons = []
    for face in faces:
        unit_normal = unit(normal(face))
        if unit_normal[2] > 0:
            c = color_map(1 - dot(unit(normal(face)), unit(light)))
            p = Polygon2D(*face_to_2d(face), fill=c, color=lines)
            polygons.append(p)
    draw2d(*polygons, axes=False, origin=False, grid=None)
```

- ➊ For each face, compute a vector of length 1 perpendicular to it.
- ➋ Only proceed if the z-component of this vector is positive, or in other words if it points toward the viewer.
- ➌ The larger the dot product between the normal vector and the light source vector, the less shading.
- ➍ If desired, we can specify a `lines` for the edges of each triangle, revealing the skeleton of the shape we're drawing.

With this render function, it only takes a few lines of code to produce an octahedron.

```
screen = Plane((-2,-2),(2,2),"screen")
render(screen, octahedron, lines='k')
screen.show()
```

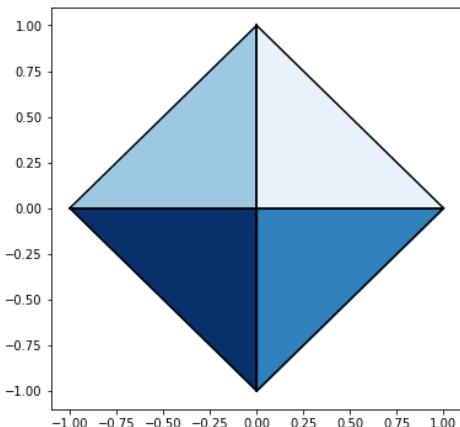


Figure 3.62 Four visible faces of the octahedron in shades of blue.

The shaded octahedron doesn't look that special from the side, but adding more faces we can tell that the shading is working. You can find pre-built shapes with more faces in the source code.

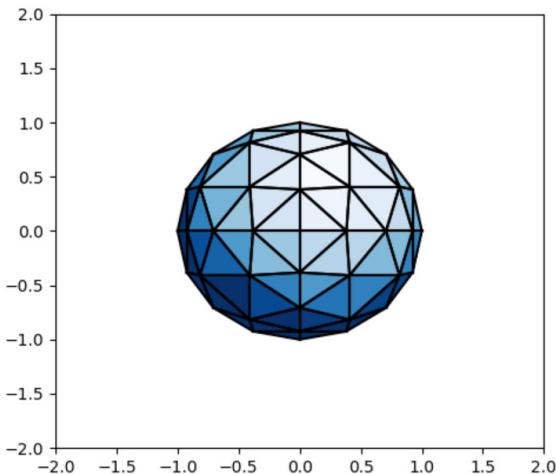


Figure 3.63 A 3D shape with many triangular sides. The effect of the shading is more apparent.

3.5.4 Exercises

MINI PROJECT

Find pairs of vectors defining each of the 12 edges of the octahedron, and draw all of the edges in Python.

SOLUTION

The “top” of the octahedron is (0,0,1). It connects to all four points in the x,y-plane via four edges. Likewise, the “bottom” of the octahedron is (0,0,-1), and it also connects to all four points in the x,y plane. Finally, the four points in the x,y-plane connect to each other in a square.

```
top = (0,0,1)
bottom = (0,0,-1)
xy_plane = [(1,0,0),(0,1,0),(-1,0,0),(0,-1,0)]
edges = [Segment3D(top,p) for p in xy_plane] + \
        [Segment3D(bottom, p) for p in xy_plane] + \
        [Segment3D(xy_plane[i],xy_plane[(i+1)%4]) for i in range(0,4)]
draw3d(*edges)
```

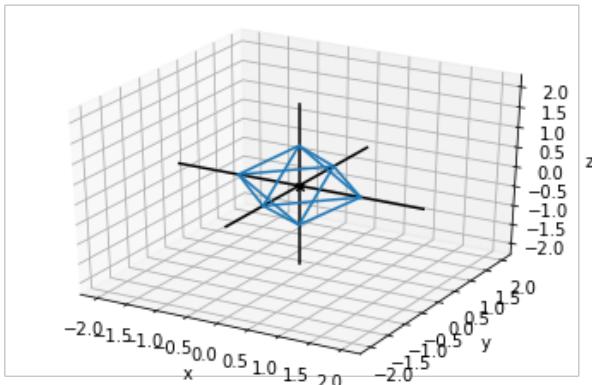


Figure 3.64 The resulting edges of the octahedron.

EXERCISE

The first face of the octahedron is $[(1,0,0), (0,1,0), (0,0,1)]$. Is that the only valid order to write the vertices for this face?

SOLUTION

No, for instance $[(0,1,0), (0,0,1), (1,0,0)]$ is the same set of three points, and the cross product still points in the same direction in this order.

3.6 Summary

In this chapter, you learned that

- Whereas vectors in 2D have lengths and widths, vectors in 3D also have depths.
- 3D vectors are defined with triples of numbers called x, y, and z-coordinates. They tell us how far from the origin we need to travel in each direction to get to a 3D point.
- As with 2D vectors, 3D vectors can be added, subtracted, and multiplied by scalars. We can find their lengths using a three-dimensional analogy of the pythagorean theorem.
- The dot product is a way to multiply two vectors and get a scalar. It measures how aligned two vectors are, and we can use its value to find the angle between two vectors.
- The cross product is a way to multiply two vectors and get a third vector which is perpendicular to both input vectors. The magnitude of the output of the cross product is the area of the parallelogram spanned by the two input vectors.
- We can represent the surface of any 3D object as a collection of triangles, where each triangle is respectively defined by three vectors representing its vertices.
- Using the cross product, we can decide what direction a triangle is visible from in 3D.

This can tell us whether a viewer will be able to see it, or how illuminated it will be by a given light source. By drawing and shading all of the triangles defining an object's surface, we can make it look three-dimensional.

4

Transforming Vectors and Graphics

This chapter covers

- Transforming and drawing 3D objects by applying mathematical transformations to the vectors that represent them
- Creating computer animations by applying successive transformations to vector graphics
- Identifying transformations which are *linear*, preserving lines and polygons
- Computing the effects of linear transformations on vectors and 3D models

You can follow the recipes from the last two chapters to render any 2D or 3D figure you can think of. Putting together line segments and polygons described by the vectors at their vertices, you can draw animated objects, characters, and worlds. But, there's still one thing standing in between you and your first feature-length computer animated film or life-like action video game: you need to be able to draw objects that *change* over time.

Animation works the same way for computer graphics as it does for film: you still render static images, but you display dozens of them every second. When we see that many snapshots of a moving object, it looks like the object is continuously changing. In chapters 2 and 3, we looked at a few mathematical operations that take in existing vectors and *transform* them geometrically to output new ones. Chaining together sequences of very small transformations, we'll be able to create the illusion of continuous motion.

As a mental model for this, you can keep in mind our examples of rotating 2D vectors. We saw we could write a Python function `rotate` that took in a 2D vector and rotated it by, say, 45 degrees in the counterclockwise direction. You can think of the `rotate` function as a machine that takes a vector in and outputs an appropriately transformed vector:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/math-for-programmers>

Licensed to Paul Olsztyn <polsztyn@gmail.com>

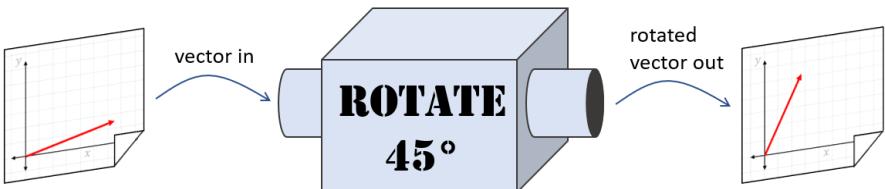


Figure 4.1 Picturing a vector function as a machine with an input slot and output slot.

If we apply a 3D analogy of this function to every vector of every polygon defining a 3D shape, we see the whole shape rotate. This 3D shape could be the octahedron from the previous chapter, or a more interesting one like a teapot. This rotation machine takes a teapot as an input and returns a rotated copy as its output.

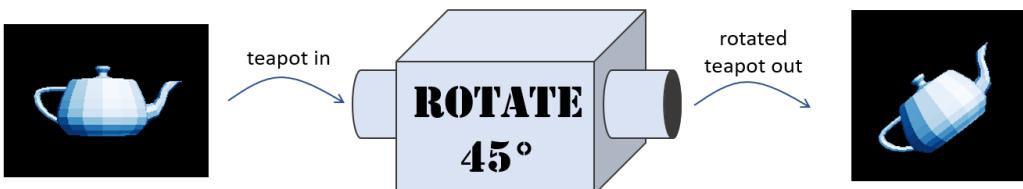


Figure 4.2 A transformation can be applied to every vector making up a 3D model, thereby transforming the whole model in the same geometric way.

If instead of rotating by 45 degrees once, we rotated by one degree 45 times, we could generate frames of a movie showing a rotating teapot.



Figure 4.3 Rotating the teapot by one degree at a time, 45 times in a row

Rotations turn out to be great examples to work with because when we rotate every point on a line segment by the same angle about the origin, we still have a line segment of the same length. As a result, when you rotate all the vectors outlining a 2D or 3D object, you can still recognize the object.

I'll introduce you to a broad class of vector transformations called *linear transformations* which, like rotations, send vectors lying on a straight line to new vectors that also lie on a straight line. Linear transformations have numerous applications in math, physics, and data analysis. It will be useful to know how to picture them when you meet them again in this book.

To visualize rotations, linear transformations, and other vector transformations in this chapter, we'll upgrade to more powerful drawing tools. We'll swap out Matplotlib for OpenGL, which is an industry standard library for high-performance graphics. Most OpenGL programming is done in C or C++, but we will use a friendly Python wrapper called PyOpenGL. We'll also use a video game development library in Python called PyGame. Specifically, we'll use the features in PyGame that make it easy to render successive images into an animation. The set-up for all of these new tools is covered in the appendix, so we can jump right in and focus on the math of transforming vectors.

4.1 Transforming 3D objects

Our main goal in this chapter is taking a 3D object like the teapot and changing it to create a new 3D object which is visually different. In Chapter 2, we already saw that we could translate or scale each vector in a 2D dinosaur and the whole dinosaur shape would move or change in size accordingly. We'll take the same approach here. Every transformation we look at will take a vector in and return a vector as output, something like this pseudocode:

```
def transform(v):
    old_x, old_y, old_z = v
    # ... do some computation here ...
    return (new_x, new_y, new_z)
```

Let's start by adapting the familiar examples of translation and scaling from 2D to 3D.

4.1.1 Drawing a transformed object

Let's start with a simple example: a function `scale2` which multiplies an input vector by the scalar 2.0.

```
from vectors import scale
def scale2(v):
    return scale(2.0, v)
```

This `scale2(v)` function has the same form as the `transform(v)` function above: when passed a 3D vector as input, it returns a new 3D vector as output. To execute this transformation on the teapot, we need to transform each of its vertices. We can do this

triangle-by-triangle: for each triangle that we used to build the teapot, we'll create a new triangle with the result of applying `scale2` to the original vertices.

```
original_triangles = load_triangles()
scaled_triangles = [
    [scale2(vertex) for vertex in triangle] ①
    for triangle in original_triangles ②
]
```

- ① Apply `scale2` to each vertex in a given triangle to get new vertices
- ② Do this for each triangle in the list of original triangles

Now that we've got a new set of triangles, we can draw them by calling `draw_model(scaled_triangles)`.

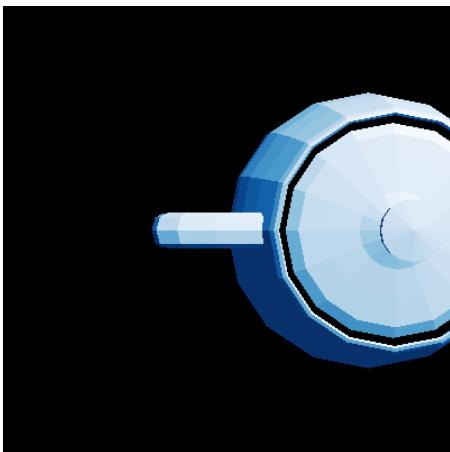


Figure 4.4 Applying `scale2` to each vertex of each triangle gives us a teapot which is twice as big.

This teapot looks about twice as big as the original: its spout is off the screen. Let's apply another transformation to each vector to re-center it: translation by the vector $(-1,0,0)$. Recall that "translating by a vector" is another way of saying "adding the vector," so what I'm really talking about is adding $(-1,0,0)$ to every vertex of the teapot. This should move the whole teapot one unit in the negative x direction, which is left from our perspective. This function will accomplish the translation for a single vertex:

```
from vectors import add
def translate1left(v):
    return add((-1,0,0), v)
```

Starting with the original triangles, we now want to scale each of their vertices as before and then apply the translation.

```
scaled_translated_triangles = [
    [translate1left(scale2(vertex)) for vertex in triangle]
```

```

        for triangle in original_triangles
    ]
draw_model(scaled_translated_triangles)

```

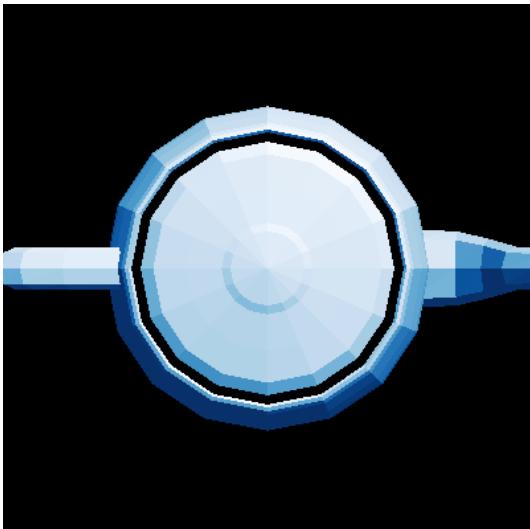


Figure 4.5 The teapot is bigger and moved to the left, as we hoped!

Different scalar multiples will change the size of the teapot by different factors, and different translation vectors will move the teapot to different positions in space. In the exercises you'll have a chance to try different scalar multiples and translations, but for now let's focus on combining and applying more transformations.

4.1.2 Composing vector transformations

Now you know how to scale and translate 3D objects, but what's more is that you can chain any number of these transformations to get a new transformation. In the case of scaling and then translating, we've built a new transformation that could be written as its own function:

```

def scale2_then_translate1left(v):
    return translate1left(scale2(v))

```

This is an important principle: since vector transformations take vectors as inputs and return vectors as outputs, we can combine as many of them as we want by *composition of functions*. If you haven't heard this term before, it means defining new functions by applying two or more existing ones in a specified order. If we picture `scale2` and `translate1left` as machines that take in 3D models and output new ones, we could combine them by passing the outputs of the first as inputs to the second.

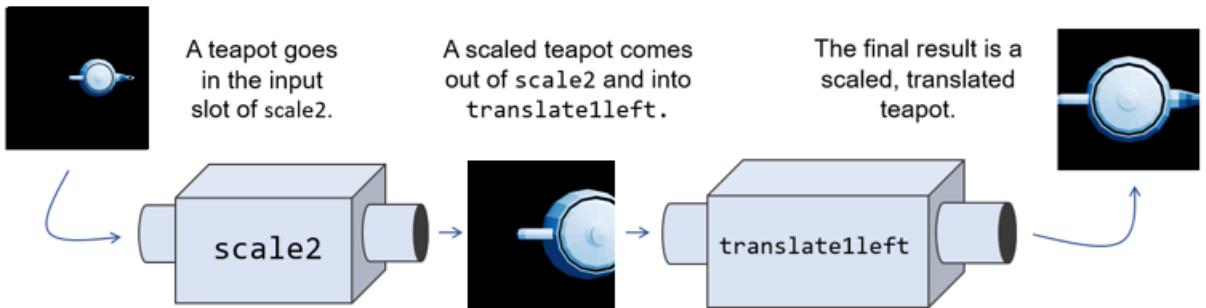


Figure 4.6 Picturing calling `scale2` and then `translate1left` on a teapot.

We could imagine hiding the intermediate step by welding the output slot of the first machine to the input slot of the second machine.

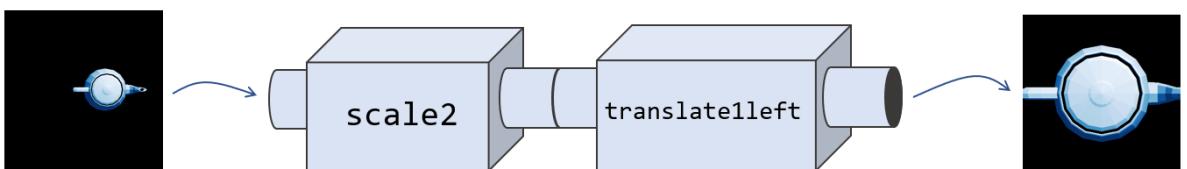


Figure 4.7 Welding" the two function machines together to get a new one, which performs both transformations in one step.

We can think of the result as a new machine altogether, which does the work of both of the original functions in one step. This "welding" of functions can be done in code as well. We can write a general purpose `compose` function which takes two Python functions (vector transformations, for instance) and returns a new function which is their composition:

```
def compose(f1,f2):
    def new_function(input):
        return f1(f2(input))
    return new_function
```

So instead of defining `scale2_then_translate1left` as its own function, we could write

```
scale2_then_translate1left = compose(translate1left, scale2)
```

You might have heard before that Python treats functions as "first-class" objects. What people usually mean by this is that Python functions can be assigned to variables, passed as inputs to other functions, or created on-the-fly and returned as output values. These are *functional programming* techniques, meaning that they help us build complex programs by building new functions out of old ones. There is some debate about whether functional programming should be encouraged in Python. I won't try to answer the general style question, but I will

say that it will be an invaluable tool for us while functions -- namely vector transformations -- are our central objects of study. You've already seen the compose function, now let me show you a few more functional recipes that justify this digression. Each of these is added in a new helper file called `transforms.py` in the source code.

Something we'll be doing repeatedly is taking a vector transformation and applying it to every vertex in every triangle defining a 3D model. We can write a reusable function for this, rather than writing a new list comprehension each time. The following `polygon_map` function takes a vector transformation and a list of polygons (usually they will be triangles) and applies the transformation to each vertex of each polygon, yielding a new list of new polygons.

```
def polygon_map(transformation, polygons):
    return [
        [transformation(vertex) for vertex in triangle]
        for triangle in polygons
    ]
```

With this helper function, we could apply `scale2` to the original teapot in one line:

```
draw_model(polygon_map(scale2, load_triangles()))
```

The `compose` and `polygon_map` functions both take vector transformations as arguments, but it's also useful to have functions that return vector transformations. For instance, it may have bothered you that we named a function "scale2" and hard-coded the number two into its definition. A replacement for this could be a `scale_by` function that returns a scaling transformation for a specified scalar.

```
def scale_by(scalar):
    def new_function(v):
        return scale(scalar, v)
    return new_function
```

With this function we could write `scale_by(2)` and the return value would be a new function which behaves identically to `scale2`. While we're picturing functions as machines with input and output slots, you can picture `scale_by` as a machine which takes numbers in its input slot and outputs new function machines from its output slot.

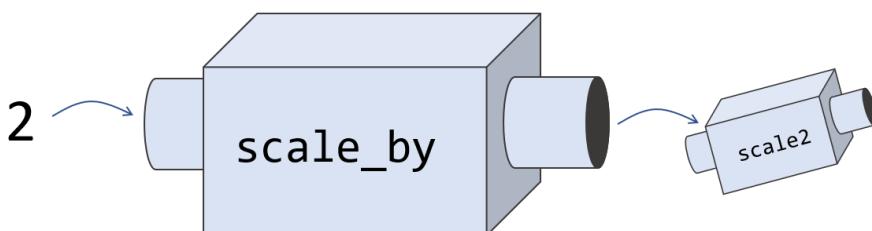


Figure 4.8 A function machine that takes numbers as inputs and produces new function machines as outputs.

As an exercise, you can write a similar `translate_by` function, which takes a translation vector as an input and returns a translation function as an output. In the terminology of functional programming, this process is called *currying*: taking a function that accepts multiple inputs and refactoring it to a function that returns another function. The result is a programmatic machine that behaves identically but is invoked differently; for instance `scale_by(s)(v)` will give the same result as `scale(s,v)` for any inputs `s` and `v`. The advantage is that `scale(...)` and `add(...)` accept different kinds of arguments, the resulting functions `scale_by(s)` and `translate_by(w)` will be interchangeable.

Next, we'll think similarly about rotations: for any given angle, we can produce a vector transformation that rotates our model by that angle.

4.1.3 Rotating an object about an axis

We've already seen how to do rotations in 2D in chapter 2: you can convert the cartesian coordinates to polar coordinates, increase or decrease the angle by the rotation factor, and then convert back. Even though this is a 2D trick, it is helpful in 3D because all 3D vector rotations are isolated to planes, in a sense. Picture, for instance, a single point in 3D being rotated about the z-axis. Its x and y coordinates will change, but its z coordinate will remain the same. If a given point is rotated around the z-axis, it will stay in a circle with a constant z-coordinate, regardless of the rotation angle.

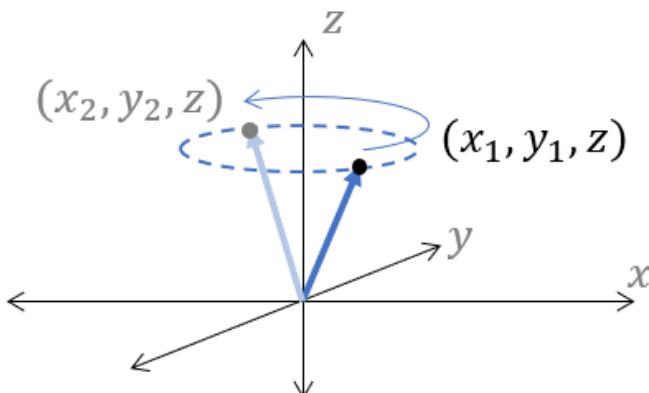


Figure 4.9

What this means is that we can rotate a 3D point around the z-axis by holding the z-coordinate constant and applying our 2D rotation function to the x and y coordinates only. Here's a 2D `rotate` function adapted from the strategy we used in chapter 2:

```
def rotate2d(angle, vector):
    l,a = to_polar(vector)
    return to_cartesian((l, a+angle))
```

This function takes an angle and a 2D vector and it returns a rotated 2D vector. Now, let's create a function `rotate_z` which applies this function only to the x and y components of a 3D vector.

```
def rotate_z(angle, vector):
    x,y,z = vector
    new_x, new_y = rotate2d(angle, (x,y))
    return new_x, new_y, z
```

Continuing to think in the functional programming paradigm, we can curry this function. Given any angle, the curried version produces a vector transformation that does the corresponding rotation.

```
def rotate_z_by(angle):
    def new_function(v):
        return rotate_z(angle,v)
    return new_function
```

Let's see it in action: the line

```
draw_model(polygon_map(rotate_z_by(pi/4.), load_triangles()))
```

yields the following teapot, which is rotated by $\pi/4$ or 45 degrees (remember we are looking from down on the teapot from the positive z axis).

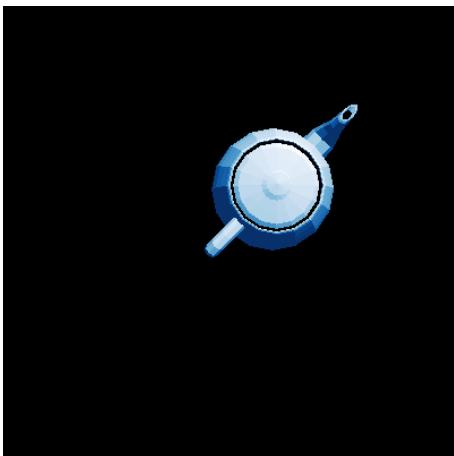


Figure 4.10 the teapot is rotated 45 degrees counterclockwise about the z-axis.

We can write a similar function to rotate the teapot about the x-axis, so we can finally admire its handle and spout.

```
def rotate_x(angle, vector):
    x,y,z = vector
    new_y, new_z = rotate2d(angle, (y,z))
    return x, new_y, new_z
```

```
def rotate_x_by(angle):
    def new_function(v):
        return rotate_x(angle,v)
    return new_function
```

In this function, a rotation about the x-axis is achieved by fixing the x coordinate and executing a 2D rotation in the y,z-plane. The following line draws us a 270 degree or $3\pi/2$ radian rotation (counterclockwise) about the x-axis, or an upright teapot.

```
draw_model(polygon_map(rotate_x_by(3*pi/2.), load_triangles()))
```



Figure 4.11 The teapot rotated by $3\pi/2$ about the x-axis.

The shading is finally consistent among these rotated teapots; their brightest polygons are toward the top-right of the figures, which is expected since the light source remains at (1,2,3). This is a good sign that we are successfully moving the teapot, and not just changing our OpenGL perspective as before.

We have plenty of other interesting transformations to explore, so let's put a bookmark in rotations for the time being. I'll leave you with good news and bad news. First, the good news: it is possible to get *any* rotation we want by composing rotations in the x and z directions. The bad news is that it is quite difficult to figure out exactly which angles we need to get to a given orientation. At the end of this chapter, I'll show you a convenient way to rotate by any angle about any axis, solving this problem.

4.1.4 Inventing your own geometric transformations

So far, I've focused on the vector transformations we've already seen in some way, shape, or form in the preceding chapters. Now, let's throw caution to the wind and see what other interesting transformations we can come up with. Remember: the only requirement for a 3D vector transformation is that it accept a single 3D vector as an input and return a new 3D vector as its output. Let's look at a few transformations that don't quite fall in any of the categories we've seen so far.

Let's try modifying one coordinate at a time. This function stretches vectors by a (hard-coded) factor of four, but only in the x-direction:

```
def stretch_x(vector):
    x,y,z = vector
    return (4.*x, y, z)
```

The result is a long, skinny teapot along the x axis, or in the "handle-to-spout" direction. Subsequent rotations and translations don't change the shape of this distorted teapot, so we can reposition it and see what it looks like.

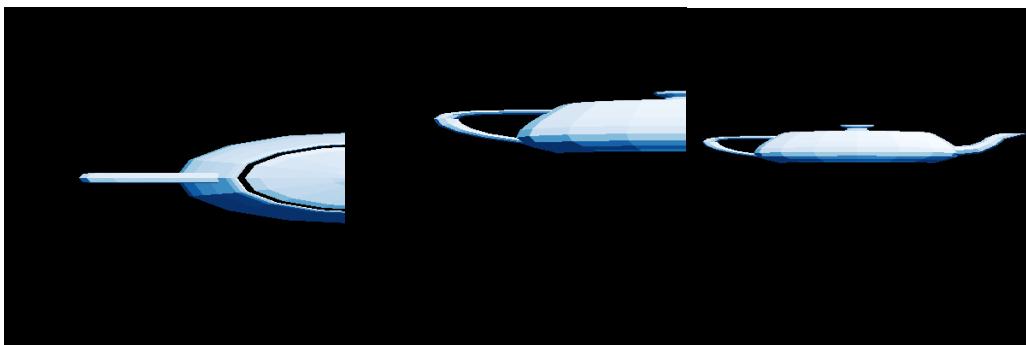


Figure 4.12 A stretched teapot. Applying `stretch_x` (left), then rotating by 270 degrees about the x axis (center), then translating by $(-2, 0, -2)$ to center it and move it further from the viewer (right).

A similar `stretch_z` function elongates the teapot from top-to-bottom. Suitable rotations and translations after applying `stretch_z` give us this teapot:

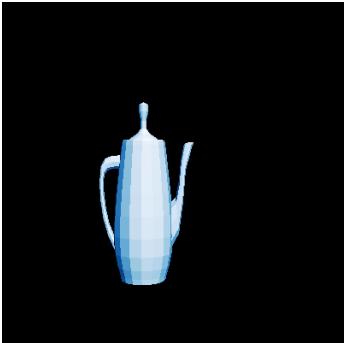


Figure 4.13 Stretching the teapot in the z-direction, then rotating it about the x-axis, then translating it to center it in the frame.

We can get even more creative, stretching the teapot by cubing the z coordinate rather than just multiplying it by a number. This transformation gives the teapot a disproportionately elongated lid.

```
def cube_stretch_z(vector):
    x,y,z = vector
    return (x, y, 2*z*z*z)
```

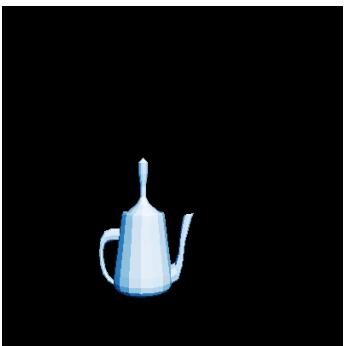


Figure 4.14 cubing the vertical dimension of the teapot.

If we selectively combine two of the three coordinates in the formula for the transformation, we can cause the teapot to slant between them:

```
def slant_xz(vector):
    x,y,z = vector
    return (x+z, y, z)
```



Figure 4.15 Adding the z coordinate to the existing x coordinate causes the teapot to slant in the x direction.

The point is not that any one of these transformations is important or useful, but that any mathematical transformation of the vectors constituting a 3D model will have *some* geometric consequence on the appearance of the model. It is possible to go too crazy with the transformation, at which point the model may become too distorted to recognize or even to draw successfully. Indeed, some transformations are “better-behaved” in general, and we’ll classify them in the next section.

4.1.5 Exercises

EXERCISE

Render the teapot translated up by 2 units in the positive z-direction. What does the resulting image look like?

SOLUTION

We can accomplish this by applying `translate_by((0,0,2))` to every vector of every polygon with `polygon_map`:

```
draw_model(polygon_map(translate_by((0,0,2)), load_triangles()))
```

Remember, we are looking at the teapot from five units up the z-axis. This transformation brings the teapot two units closer to us, so it looks larger.

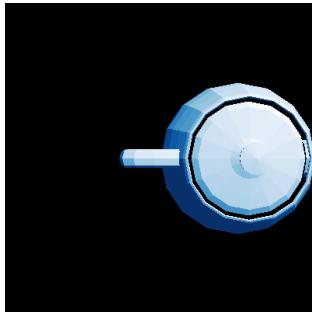


Figure 4.16 The teapot translated 2 units up the z-axis. It appears larger because it is closer to the viewpoint.

MINI-PROJECT

What happens to the teapot when you scale every vector by a scalar between 0 and 1? What happens when you scale it by a factor of -1?

SOLUTION

We can apply `scale_by(0.5)` and `scale_by(-1)` to see the results.

```
draw_model(polygon_map(scale_by(0.5), load_triangles()))
draw_model(polygon_map(scale_by(-1), load_triangles()))
```

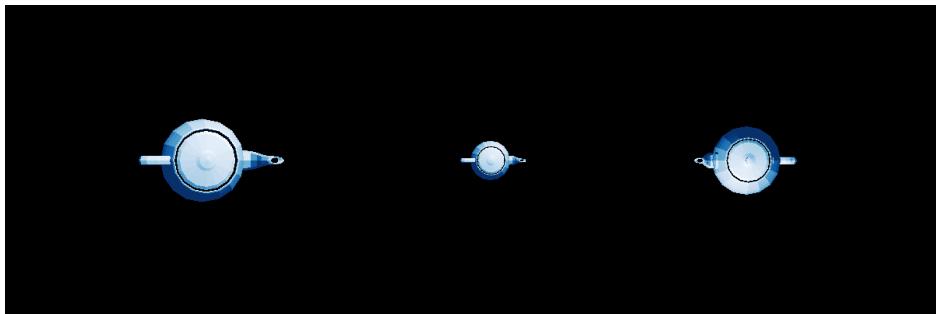


Figure 4.17 Left-to-right, the original teapot, the teapot scaled by 0.5, and the teapot scaled by -1.

As you can see, `scale_by(0.5)` shrinks the teapot to half its original size. The action of `scale_by(-1)` seems to rotate the teapot by 180° , but the situation is a bit more complicated. Looking at the teapot scaled by -1 from a different perspective, we can see it has been flipped upside-down!

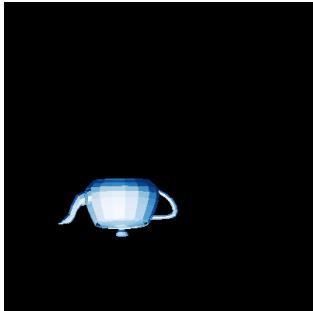


Figure 4.18 Applying `scale_by(-1)` rotated the teapot by 180° but also turned it upside-down.

If the teapot was upside down, why did we still see its top when we looked at it from above? The answer is that it was not just reflected, but also turned “inside-out.” Each triangle has been reflected, so each normal vector now points into the teapot rather than outward from its surface.

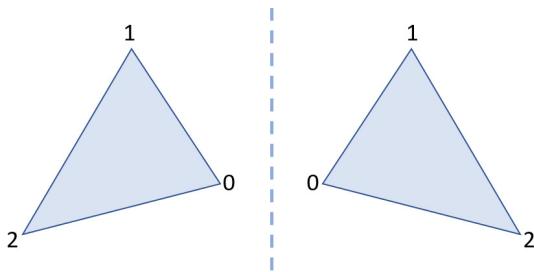


Figure 4.19 Reflection changes the orientation of a triangle. The indexed vertices are in counterclockwise order on the left, and clockwise order in the reflection on the right. The normal vectors to these triangles point in opposite directions.

Rotating the teapot, you can see that it is not quite rendering correctly as a result. We should be careful with reflections of our graphics for this reason!

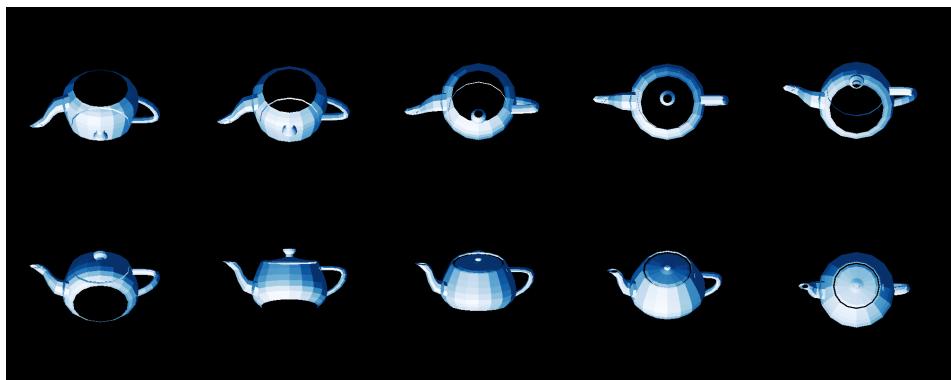


Figure 4.20 The rotated, reflected teapot does not look quite right; some features appear which should be concealed. For instance, we can see both the lid and the hollow bottom in the first frame.

EXERCISE

First apply `translateleft` to the teapot, and then apply `scale2` after. What is different than the opposite order of composition? Why?

SOLUTION

We can compose these two functions in the specified order and then apply them with `polygon_map`.

```
draw_model(polygon_map(compose(scale2, translateleft), load_triangles())))
```

The resulting is still twice as big as the original, but this one is translated further to the left. This is because when a scaling factor of two is applied after a translation, the distance of the translation is doubled as well.



Figure 4.21 Scaling after translating has the effect of scaling the translation as well. The teapot is moved twice as far to the left.

EXERCISE

What is the effect of the transformation `compose(scale_by(0.4), scale_by(1.5))`?

SOLUTION

Applying this to a vector will scale it by 1.5 and then by 0.4, for a net scaling factor of 0.6. The resulting figure will be 60% of the size of the original.

EXERCISE

Modify the `compose(f,g)` function to `compose(*args)`, which takes several functions as arguments and returns a new function which is their composition.

SOLUTION:

```
def compose(*args):
    def new_function(input):      1
        state = input            2
        for f in reversed(args):  3
            state = f(result)    4
        return state
```

```
    return new_function
```

- 1 Start defining the function that `compose` will return.
- 2 Set the current state equal to the input.
- 3 Iterate over the input functions in reverse order since the inner functions of a composition are applied first.
For example, `compose(f, g, h)(x)` should equal `f(g(h(x)))`; the first function to apply is `h`.
- 4 At each step, update the state by applying the next function. The final state will have had all the functions applied in the correct order.

To check our work, we can build some functions and compose them:

```
def prepend(string):
    def new_function(input):
        return string + input
    return new_function

f = compose(prepend("P"), prepend("y"), prepend("t"))
```

Then running `f("hon")` returns the string “Python”. In general, the constructed function `f` appends the string “`Pyt`” to whatever string it is given.

EXERCISE

Write a `curry2(f)` function that takes in a Python function `f(x, y)` with two arguments and returns a new function which is curried. For instance, once you write `g = curry2(f)`, the two expressions `f(x, y)` and `g(x)(y)` should return the same result.

SOLUTION

The return value should be a new function which in turn produces a new function when called.

```
def curry2(f):
    def g(x):
        def new_function(y):
            return f(x,y)
        return new_function
    return g
```

As an example use case, we could have built the `scale_by` function like this:

```
>>> scale_by = curry2(scale)
>>> g = scale_by(2)
>>> g((1,2,3))
(2, 4, 6)
```

EXERCISE

Write a function `rotate_y_by` in analogy to `rotate_x_by` and `rotate_z_by`. What direction does your new function rotate through the x-z plane?

SOLUTION

The implementation for `rotate_y` is the same as `rotate_x` or `rotate_z` except this time the y-coordinate is the one left unchanged. Then you can get `rotate_y_by` by currying `rotate_y`.

```
def rotate_y(angle, vector):
    x,y,z = vector
    new_x, new_z = rotate2d(angle(x,z))
    return new_x, y, new_z

def rotate_y_by(angle):
    return new_function(v):
        return rotate_y(angle,v)
    return new_function
```

Applying `rotate_y_by(pi/2)` to the teapot, we see it effects a $\pi/2$ rotation *clockwise* from the perspective of the positive y axis.

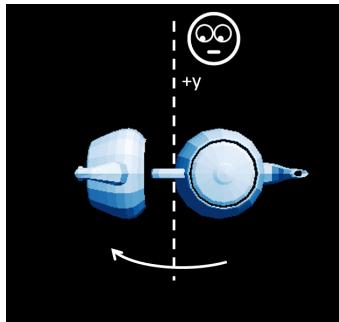


Figure 4.22 Looking at the teapot from somewhere up the positive y-axis, we would see it rotated in the clockwise direction.

This is different from `rotate_x_by` and `rotate_z_by`, which produce counterclockwise rotations as viewed from the positive x and z axes respectively. Can you figure out why?

EXERCISE

Without running it, what is the result of applying the transformation `compose(rotate_z_by(pi/2), rotate_x_by(pi/2))`? What about if you switch the order of the composition?

SOLUTION

This composition is equivalent to a clockwise rotation by $\pi/2$ about the y-axis. Reversing the order gives a counterclockwise rotation by $\pi/2$ about the y-axis.

EXERCISE

Write a function `stretch_x(scalar,vector)` that scales the target vector by the given factor, but only in the x-direction. Also write a curried version `stretch_x_by` so that `stretch_x_by(scalar)(vector)` returns the same result.

SOLUTION:

```
def stretch_x(scalar,vector):
    x,y,z = vector
    return (scalar*x, y, z)

def stretch_x_by(scalar):
    def new_function(vector):
        return stretch_x(scalar,vector)
    return new_function
```

4.2 Linear transformations

The “well-behaved” vector transformations we’re going to focus on are called *Linear Transformations*. Along with vectors, linear transformations are the other main objects of study in the mathematical subject of linear algebra. Linear transformations are special transformations where vector arithmetic looks the same before or after the transformation. Let me show you what I mean

4.2.1 Preserving vector arithmetic

The two most important arithmetic operations on vectors are addition and scalar multiplication. Let’s return to our 2D pictures of these operations and see how they look before and after a transformation is applied.

We can picture the sum of two vectors as the new vector we get when we place them tip-to-tail, or the vector to the tip of the parallelogram they define. For instance, the following picture represents the vector sum $u + v = w$.

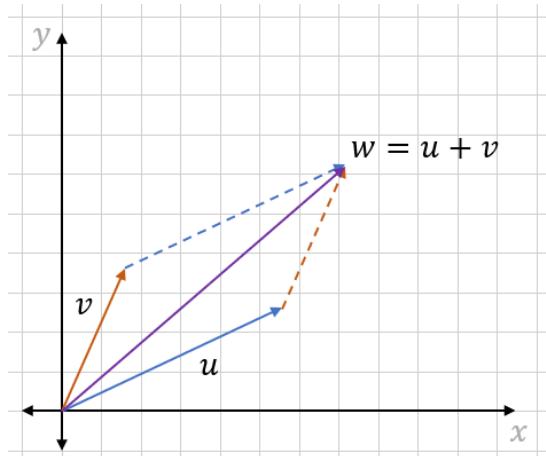


Figure 4.23 Geometric demonstration of the vector sum $u + v = w$.

Next, let's apply the same vector transformation to all three of these vectors. Specifically, let's use a rotation R which is a counterclockwise rotation transformation. Here are u , v , and w rotated by the same angle under the influence of the transformation R .

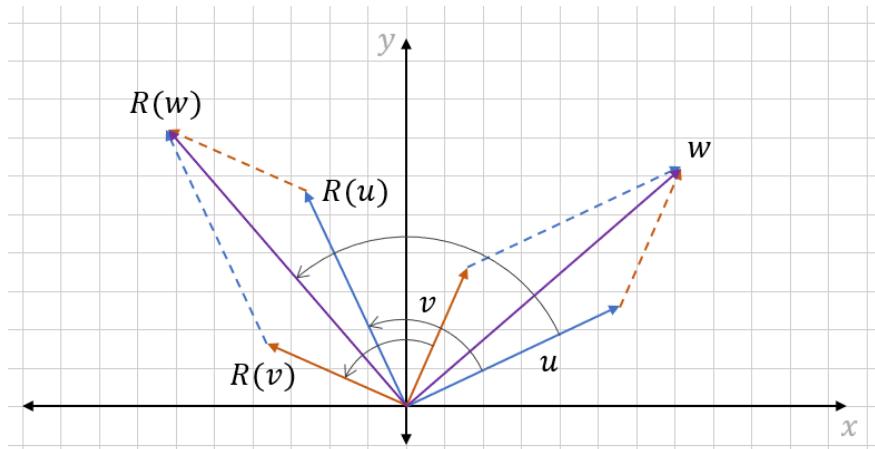


Figure 4.24 After rotating u , v , and w by the same rotation R , the sum still holds.

This picture tells us that for the selected vectors u , v , and w , the sum equality still holds *after* the rotation as well: $R(u) + R(v) = R(w)$. You can draw the picture for any three vectors u , v , and w , and as long as $u + v = w$ and you apply the same rotation transformation R to them you will find that $R(u) + R(v) = R(w)$ as well. To describe this property, we could say that any rotation *preserves* vector sums.

It's similarly true that rotations preserve scalar multiples. If v is a vector and sv is a multiple of v by a scalar s , then sv points in the same direction but is scaled by a factor of s .

If we rotate by v and sv by the same rotation R , we'll see that $R(sv)$ is a scalar multiple of $R(v)$ by the same factor, s .

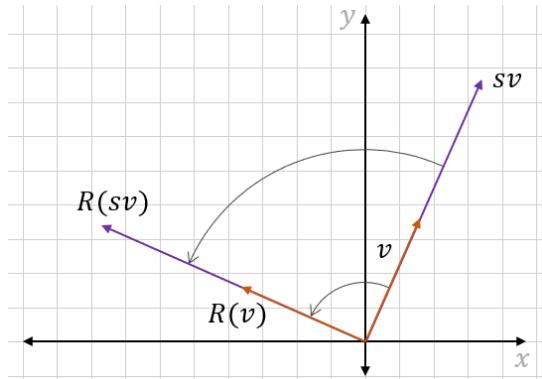


Figure 4.25 Scalar multiplication is preserved by rotation.

Again, this is only a visual example and not a proof, but you'll find that for any vector v , scalar s , and rotation R , the same picture holds. Rotations, or any other vector transformations that preserves vector sums and scalar multiples, are called *linear transformations*.

DEFINITION

A vector transformation T is said to be a *linear transformation* if it preserves vector addition and scalar multiplication. That is, for any input vectors u and v , we have

$$T(u) + T(v) = T(u+v)$$

and for any pair of a scalar s and a vector v we have

$$T(sv) = sT(v).$$

This definition is so important, it's worth looking at several more examples to get the picture in your head.

4.2.2 Picturing linear transformations

First, let's look at a counterexample: a vector transformation which is *not* linear. Such an example is a transformation $S(v)$ that takes a vector $v = (x,y)$ and outputs a vector with both coordinates squared: $S(v) = (x^2, y^2)$. As an example, let's look at the sum of $u = (2,3)$ and $v = (1,-1)$. The sum is $(2,3) + (1,-1) = (3,2)$.

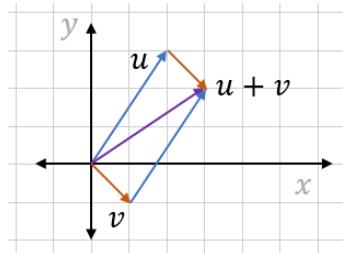


Figure 4.26 Picturing the vector sum of $u = (2,3)$ and $v = (1,-1)$; $u+v = (3,2)$.

Now let's apply S to each of these:

$$S(u) = (4,9),$$

$$S(v) = (1,1),$$

and

$$S(u+v) = (9,4).$$

Clearly $S(u) + S(v)$ does *not* agree with $S(u+v)$.

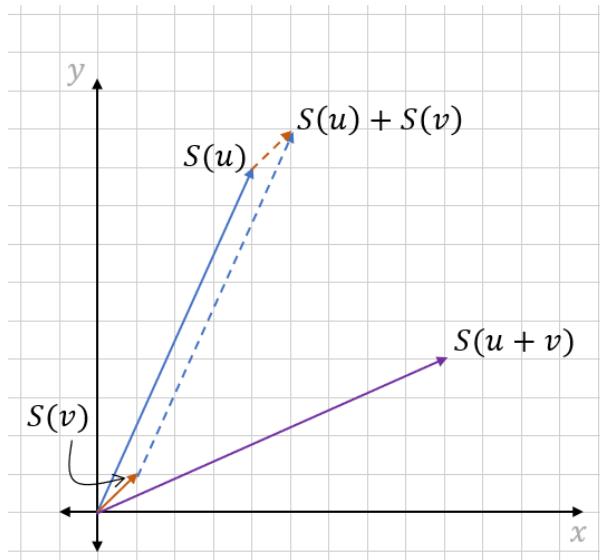


Figure 4.27 S does not respect sums; $S(u) + S(v)$ is far from $S(u+v)$.

As an exercise, you can find a counterexample demonstrating that S does not preserve scalar multiples either.

Let's examine another transformation. Let $D(v)$ be the vector transformation that scales the input vector by a factor of 2. In other words, $D(v) = 2v$. This *does* preserve vector sums: if $u + v = w$, then $2u + 2v = 2w$ as well. Here's a visual example.

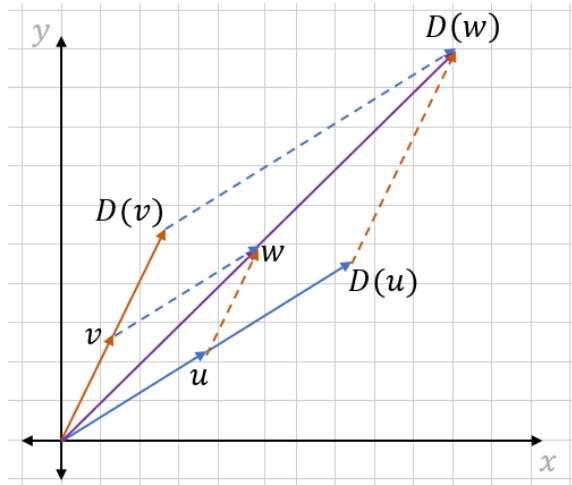


Figure 4.28 Doubling the lengths of vectors preserves their sums. If $u + v = w$, then $D(u) + D(v) = D(w)$.

Likewise, $D(v)$ preserves scalar multiplication. This is a bit harder to draw, but you can see it algebraically. For any scalar s , $D(sv) = 2(sv) = s(2v) = sD(v)$.

How about translation? Suppose $B(v)$ translates any input vector v by $(7, 0)$. Surprisingly, this is *not* a linear transformation. Here's a visual counterexample where $u + v = w$ but $B(v) + B(w)$ is not the same as $B(v+w)$.

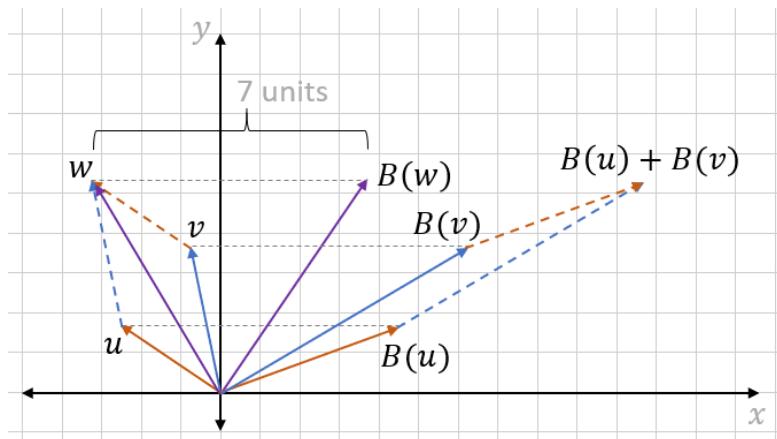


Figure 4.29 The translation transformation B does not preserve a vector sum, since $B(u)+B(v)$ is not equal to $B(u+v)$.

It turns out that for a transformation to be linear, it must not move the origin (see why as an exercise!). Translation by any non-zero vector will transform the origin to end up at a different point, so it cannot be linear.

Other examples of linear transformations include reflection, projection, and shearing, and any 3D analogy of the preceding linear transformations. These are defined in the exercises section, and you should convince yourself with several examples that each of these transformations preserves vector addition and scalar multiplication. With practice, you can recognize which transformations are linear and which are not. It's clear from the definition that linear transformations have special properties; now I'll show you why these properties are useful.

4.2.3 Why linear transformations?

Because linear transformations preserve vector sums and scalar multiples, they preserve a broader class of vector arithmetic operations as well. The most general operation is called a *linear combination*. A linear combination of a collection of vectors is a sum of scalar multiples of them. For instance, one linear combination of two vectors u and v would be $3u - 2v$. A linear combination of three vectors u , v , and w could be $0.5u - v + 6w$. The two defining properties of linear transformations ensure that they preserve all linear combinations.

A general way to say this is if you have a collection of n vectors, v_1, v_2, \dots, v_n , as well as any choice of n scalars $s_1, s_2, s_3, \dots, s_n$, a linear transformation T preserves the linear combination:

$$T(s_1v_1 + s_2v_2 + s_3v_3 + \dots + s_nv_n) = s_1T(v_1) + s_2T(v_2) + s_3T(v_3) + \dots + s_nT(v_n).$$

There are some very important linear combinations we've been relying on. One is the linear combination $\frac{1}{2} u + \frac{1}{2} v$, which is equivalent to $\frac{1}{2} (u + v)$. This linear combination of two vectors gives us the midpoint of the line segment connecting them.

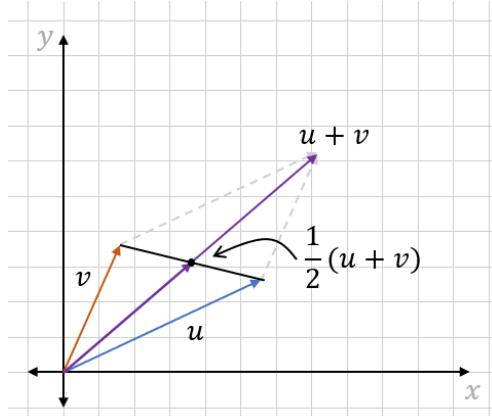


Figure 4.30 The midpoint between the tips of two vectors u and v can be found as the linear combination $\frac{1}{2} u + \frac{1}{2} v = \frac{1}{2} (u+v)$.

This means linear transformations send midpoints to other midpoints: $T(\frac{1}{2}u + \frac{1}{2}v) = \frac{1}{2}T(u) + \frac{1}{2}T(v)$, which is the midpoint of the segment connecting $T(u)$ and $T(v)$.

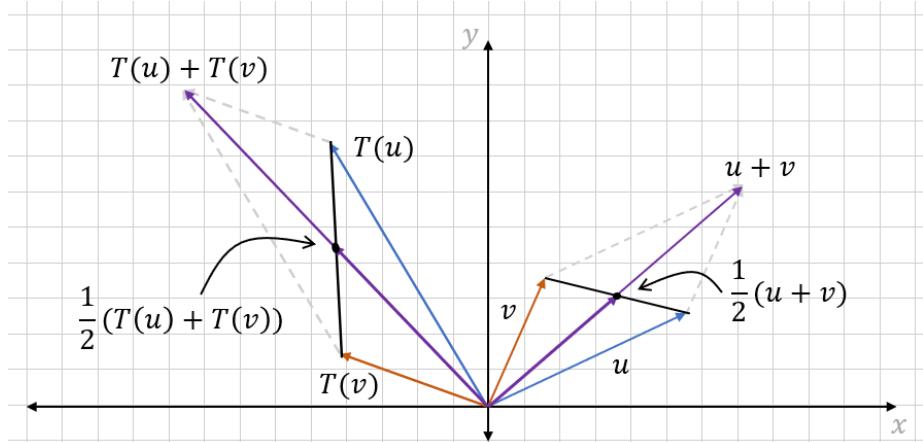


Figure 4.31 Because the midpoint between two vectors is a linear combination of the vectors, the linear transformation T sends the midpoint between u and v to the midpoint between $T(u)$ and $T(v)$.

It's less obvious, but a linear combination like $0.25u + 0.75v$ also lies on the line segment between u and v . Specifically, this is the point 75% of the way from u to v . Likewise, $0.6u + 0.4v$ is 40% of the way from u to v , and so on.

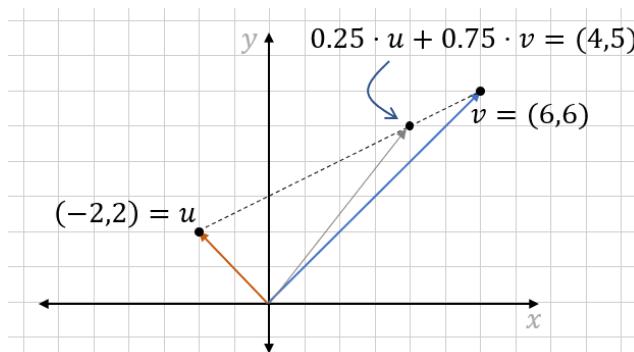


Figure 4.32 The point $0.25u + 0.75v$ lies on the line segment connecting u and v , 75% of the way from u to v . You can see this concretely with $u = (-2,2)$ and $v = (6,6)$.

In fact, every point on the line segment between two vectors is a "weighted average" like this, having the form $su + (1-s)v$ for some number s between 0 and 1. To convince you, here are vectors $su + (1-s)v$ for $u = (-1,1)$ and $v = (3,4)$ for 10 values of s between 0 and 1, and then for 100 values of s between 0 and 1:

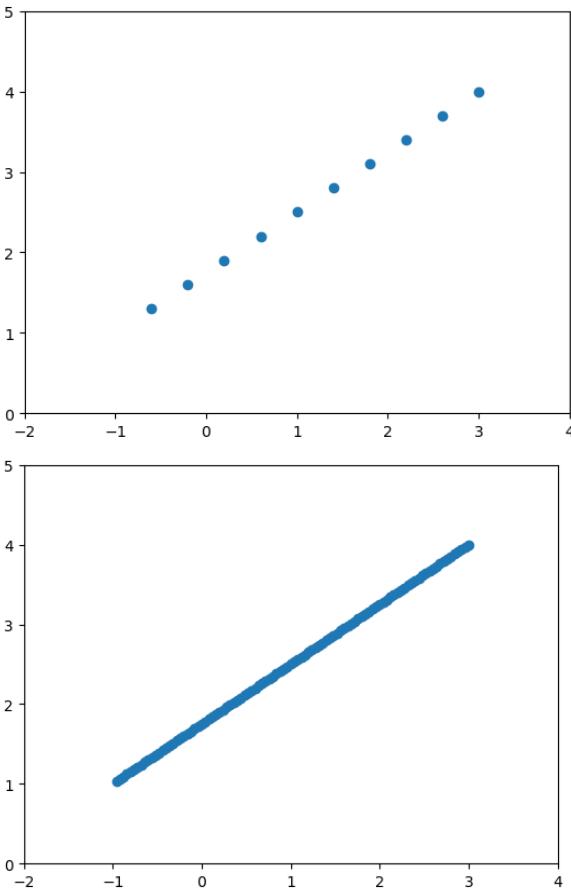


Figure 4.33 Plotting various weighted averages of $(-1, 1)$ and $(3, 4)$ with 10 values of s between 0 and 1 (left) and 100 values of s between 0 and 1 (right)

The key idea here is that every point on a line segment connecting two vectors u and v is a weighted average and therefore a linear combination of points u and v . This lets us reason about the effect of a linear transformation on a line segment. Any point on the line segment connecting u and v is a weighted average of u and v , so it has the form $s \cdot u + (1-s) \cdot v$ for some value of s . A linear transformation, T , transforms u and v to some new vectors $T(u)$ and $T(v)$. The point on the line segment is transformed to some new point $T(s \cdot u + (1-s) \cdot v)$ or $s \cdot T(u) + (1-s) \cdot T(v)$. This is, in turn, a weighted average of $T(u)$ and $T(v)$, so it is a point that lies on the segment connecting $T(u)$ and $T(v)$.

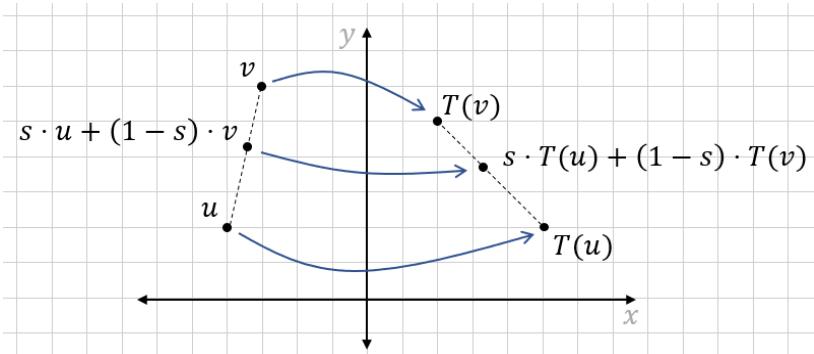


Figure 4.34 A linear transformation T transforms a weighted average of u and v to a weighted average of $T(u)$ and $T(v)$. The original weighted average lies on the segment connecting u and v , and the transformed one lies on the segment connecting $T(u)$ and $T(v)$.

Because of this, a linear transformation T takes every point on the line segment connecting u and v to a point on the line segment connecting $T(u)$ and $T(v)$. This is a key property of linear transformations: they send every existing line segment to a new line segment. Since our 3D models are made up of polygons, and polygons are outlined by line segments, linear transformations can be expected to preserve the structure of our 3D models to some extent.

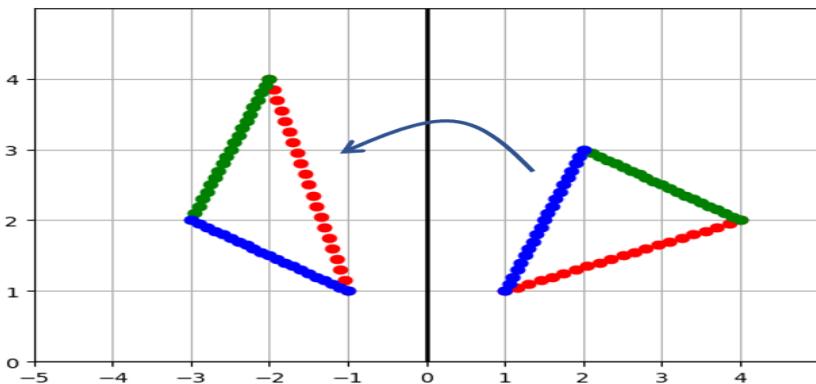


Figure 4.35 Applying a linear transformation (rotation by 60 degrees) to points making up a triangle; the result is a rotated triangle.

By contrast, if we use the non-linear transformation $S(v)$ sending $v = (x, y)$ to (x^2, y^2) , we can see that line segments are distorted. This means that a triangle defined by vectors u , v , and w is not really sent to another triangle defined by $S(u)$, $S(v)$, and $S(w)$.

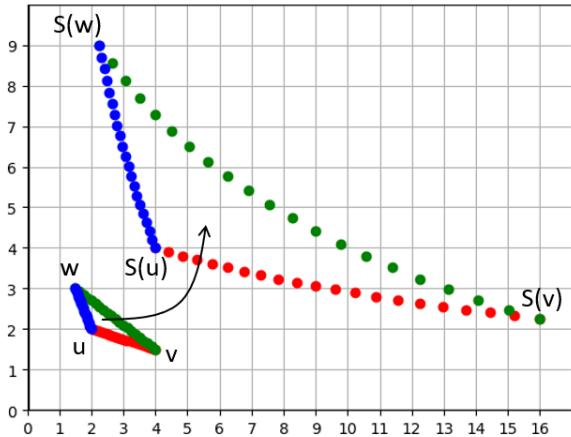


Figure 4.36 Applying the non-linear transformation S does *not* preserve straightness of edges of the triangle.

In summary, linear transformations respect the algebraic properties of vectors, preserving sums, scalar multiples, and linear combinations. They also respect the geometric properties of collections of vectors, sending line segments and polygons defined by vectors to new ones defined by the transformed vectors. Finally, we'll see that all linear transformations can be expressed in a formulaic way, which makes them very easy to compute with.

4.2.4 Computing linear transformations

In chapters 2 and 3, we saw how to break 2D and 3D vectors into components. For instance, the vector $(4,3,5)$ can be decomposed as a sum $(4,0,0) + (0,3,0) + (0,0,5)$. This makes it easy to picture how far the vector extends in each of the three dimensions of the space we're in. It may seem pedantic, but we can decompose this even further into a linear combination:

$$(4,3,5) = 4 \cdot (1,0,0) + 3 \cdot (0,1,0) + 5 \cdot (0,0,1)$$

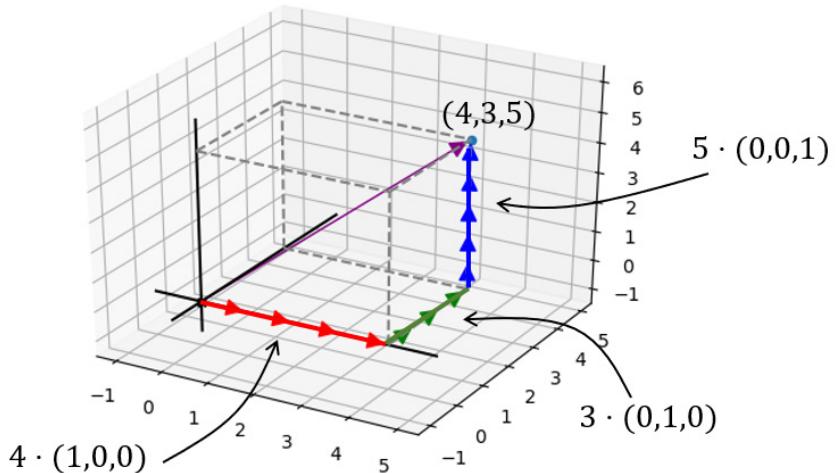


Figure 4.37 $(4,3,5)$ as linear combination of $(1,0,0)$ $(0,1,0)$ and $(0,0,1)$.

This may seem like a boring fact, but it's one of the profound insights from linear algebra: any 3D vector can be decomposed into a linear combination of three vectors $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$. The scalars appearing in this decomposition for a vector v are exactly the coordinates of v . The three vectors $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ are called the *standard basis* for three-dimensional space. They are denoted e_1 , e_2 , and e_3 , so we could write the linear combination above as $(3,4,5) = 3 e_1 + 4 e_2 + 5 e_3$. When we're working in 2D space, we call $e_1 = (1,0)$ and $e_2 = (0,1)$, so for example $(7,-4) = 7 e_1 - 4 e_2$. (When we say e_1 , we could mean $(1,0)$ or $(1,0,0)$ but usually it's clear which one we mean because we've established whether we're working in two or three dimensions.)

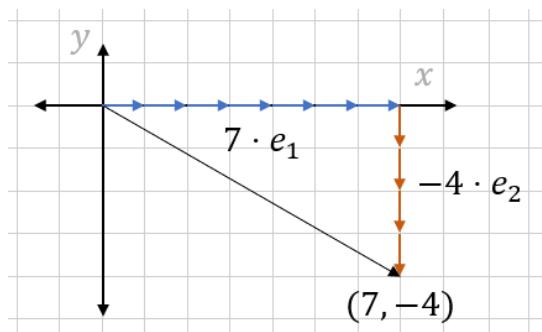


Figure 4.38 The 2D vector $(7,-4)$ as a linear combination of standard basis vectors e_1 and e_2 .

So, what? We've only written the same vectors in a slightly different way, but it turns out this change in perspective makes it very easy to compute linear transformations. Because linear

transformations respect linear combinations, all we need to know to compute a linear transformation is how it affects standard basis vectors.

Let's look at a visual example. Say we know nothing about a 2D vector transformation T except that it is linear and we know what $T(e_1)$ and $T(e_2)$ are.

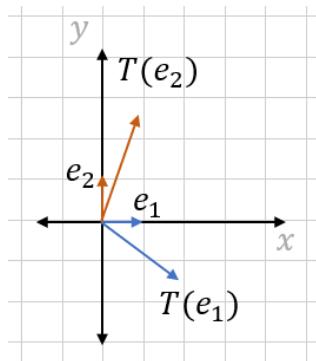


Figure 4.39 When a linear transformation acts on the two standard basis vectors in 2D, we get two new vectors as a result.

For any other vector v , we automatically know where $T(v)$ will end up. Say $v = (3,2)$. Then we can assert:

$$T(v) = T(3e_1 + 2e_2) = 3T(e_1) + 2T(e_2)$$

Since we already know where $T(e_1)$ and $T(e_2)$ are, we can locate $T(v)$:

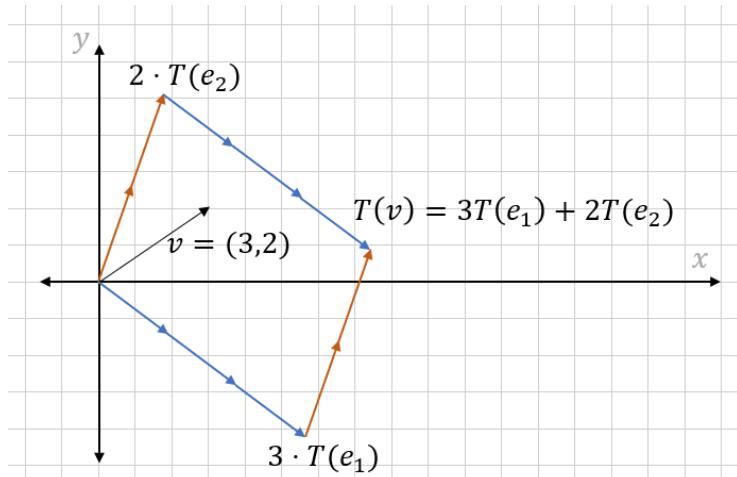


Figure 4.40 We can compute $T(v)$ for any vector v as a linear combination of $T(e_1)$ and $T(e_2)$.

Let's do a 3D example with all numbers included to hammer this home. Say A is a linear transformation, and $A(e_1) = (1,1,1)$, $A(e_2) = (1,0,-1)$, and $A(e_3) = (0,1,1)$. If $v = (-1,2,2)$, what is $A(v)$? Well, first we can expand v as a linear combination of the three standard basis vectors. Since $v = (-1,2,2) = -e_1 + 2e_2 + 2e_3$, we can make the substitution:

$$A(v) = A(-e_1 + 2e_2 + 2e_3)$$

Next, we can use the fact that A is linear and preserves linear combinations:

$$= -A(e_1) + 2A(e_2) + 2A(e_3)$$

Finally, we can substitute in the known values of $A(e_1)$, $A(e_2)$, and $A(e_3)$ and simplify:

$$\begin{aligned} &= -(1,1,1) + 2*(1,0,-1) + 2*(0,1,1) \\ &= (1, 1, -1). \end{aligned}$$

As proof we really know how A works, we can apply it to the teapot:

```
Ae1 = (1,1,1)      ①
Ae2 = (1,0,-1)
Ae3 = (0,1,1)

def apply_A(v):  ②
    return add( ③
        scale(v[0], Ae1),
        scale(v[1], Ae2),
        scale(v[2], Ae3)
    )

draw_model(polygon_map(apply_A, load_triangles())) ④
```

- ① First write down the results of applying A to the standard basis vectors.
- ② Build a function `apply_A(v)` which returns the result of A on the input vector v.
- ③ The result should be a linear combination of these vectors, where the scalars are taken to be the coordinates of the target vector v.
- ④ Finally, use `polygon_map` to apply A to every vector of every triangle in the teapot.

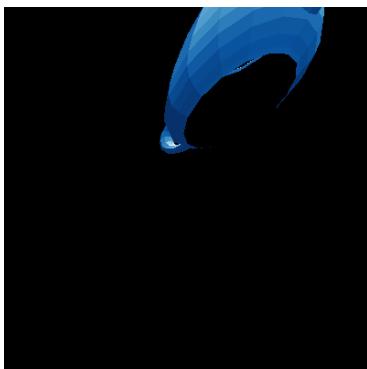


Figure 4.41 The transformed teapot – in this rotated, skewed configuration, the teapot shows us it does not have a bottom!

The takeaway here is that a 2D linear transformation T is defined completely by the values of $T(e_1)$ and $T(e_2)$: two vectors or four numbers in total. Likewise, a 3D linear transformation T is defined completely by the values of $T(e_1)$, $T(e_2)$, and $T(e_3)$: three vectors or nine numbers in total. In any number of dimensions, the behavior of a linear transformation is specified by a list of vectors, or an array-of-arrays of numbers. Such an array-of-arrays is called a *matrix*, and we'll see how to use matrices in the next chapter.

4.2.5 Exercises

EXERCISE

Considering S again, the vector transformation that squares all coordinates, show algebraically that $S(sv) = sS(v)$ does not hold for all choices of scalars s and 2D vectors v .

SOLUTION

Let $v = (x,y)$. Then $sv = (sx,sy)$ and $S(sv) = (s^2x^2, s^2y^2) = s^2(x^2,y^2) = s^2S(v)$. For most values of s and most vectors v , $S(sv) = s^2S(v)$ won't equal $sS(v)$. A specific counterexample is $s=2$ and $v=(1,1,1)$, where $S(sv) = (4,4,4)$ while $sS(v) = (2,2,2)$. This counterexample shows that S is not linear.

EXERCISE

Suppose T is a vector transformation and $T(0) \neq 0$, where 0 here represents the vector with all coordinates equal to zero. Why can T not be linear according to the definition?

SOLUTION

For any vector v , $v + 0 = v$. For T to preserve vector addition, it should be that $T(v+0) = T(v) + T(0)$. But, since $T(v+0) = T(v)$ this would require that $T(v) = T(v) + T(0)$, or $0 = T(0)$. Given that this is not the case, T cannot be linear.

EXERCISE

The identity transformation is the vector transformation that returns the same vector it is passed. It is denoted with a capital I , so we could write its definition as $I(v) = v$ for all vectors v . Explain why I is a linear transformation.

SOLUTION

For any vectors v and w , $I(v+w) = v+w = I(v) + I(w)$ and for any scalar s , $I(sv) = sv = sI(v)$. These equalities show that the identity transformation preserves vector sums and scalar multiples.

EXERCISE

What is the midpoint between $(5,3)$ and $(-2, 1)$? Plot all three of these points to see that you are correct.

SOLUTION

The midpoint is $\frac{1}{2}(5,3) + \frac{1}{2}(-2,1)$ or $(5/2, 3/2) + (-1, 1/2)$ which equals $(3/2, 2)$. This looks right when drawn to scale in the diagram below.

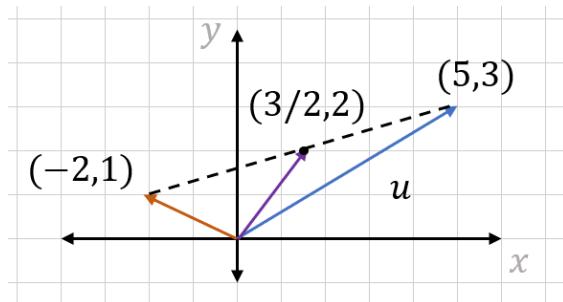


Figure 4.42 The midpoint of the segment connecting $(5,3)$ and $(-2,1)$ is $(3/2,2)$.

EXERCISE

Plot all 36 vectors v with integer coordinates 0 to 5 as points using the drawing code from chapter 2. and then plot $S(v)$ for each of them. What happens geometrically to vectors under the action of S ?

SOLUTION

The space between points is uniform to begin with, but in the transformed picture the spacing increases in the horizontal and vertical directions as the x and y -coordinates increase, respectively.

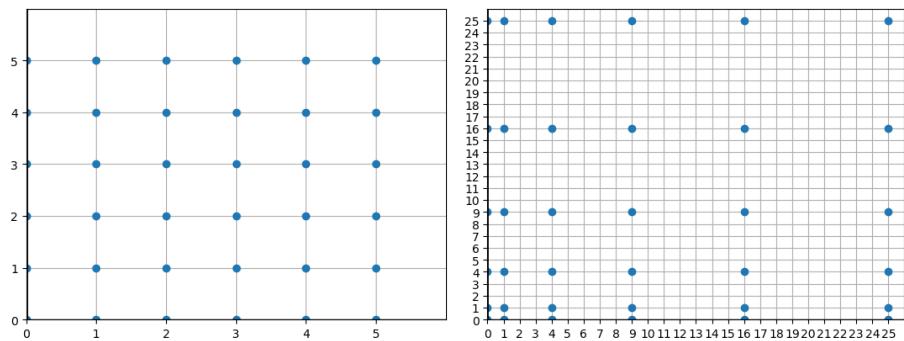


Figure 4.43 The grid of points is initially uniformly spaced, but after applying the transformation S , the spacing varies between points, even on the same lines.

MINI-PROJECT

Property-based testing is a type of unit testing that involves inventing arbitrary input data for a program and then checking that the outputs satisfy desired conditions. There are popular Python libraries like Hypothesis (available through pip) that make it easy to set this up. Using your library of choice, implement property-based tests that check if a vector transformation is linear. Specifically, given a vector transformation T implemented

as a python function, generate a large number of pairs of random vectors and assert for all of them that their sum is preserved by T . Then, do the same thing for pairs of a scalar and a vector, and ensure that T preserves scalar multiples. You should find that linear transformations like `rotate_x_by(pi/2)` will pass the test, but non-linear transformations like the coordinate-squaring transformation will not pass.

EXERCISES

One 2D vector transformation is *reflection* across the x -axis. This transformation takes a vector and returns another one which is the mirror image with respect to the x -axis: its x -coordinate should be unchanged, and its y -coordinate should change sign. Denoting this transformation S_x , here is an image of a vector $v = (3,2)$ and the transformed vector $S_x(v)$.

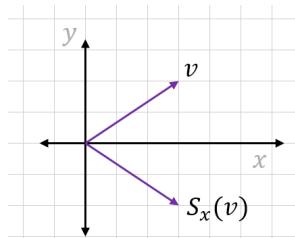


Figure 4.44 A vector $v = (3,2)$ and its reflection over the x -axis $(3,-2)$.

Draw two vectors and their sum, as well as the reflection of these three vectors, to demonstrate that this transformation preserves vector addition. Draw another diagram to show similarly that scalar multiplication is preserved, thereby demonstrating both criteria for linearity.

SOLUTION

Here's an example of reflection over the x -axis preserving a vector sum.

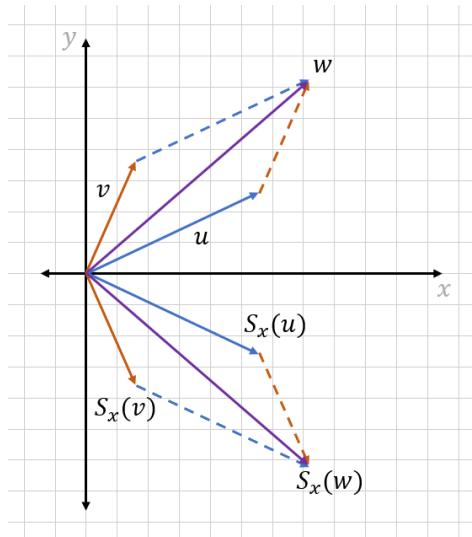


Figure 4.45 For $u + v = w$ as shown, reflection over the x axis preserves the sum. That is, $S_x(u) + S_x(v) = S_x(w)$ as well.

And here's an example showing reflection preserving a scalar multiple: $S_x(sv)$ lies where $sS_x(v)$ is expected to be.

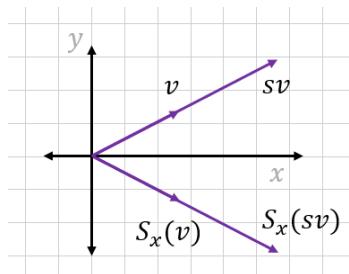


Figure 4.46 Reflection across the x axis preserves this scalar multiple.

To prove that S_x is linear, you would need to show that these pictures hold for every vector sum and every scalar multiple. There are infinitely many of these, so it's better to use an algebraic proof (can you figure out how to show these two facts algebraically?).

MINI PROJECT

Suppose S and T are both linear transformations. Explain why the composition of S and T is also linear.

SOLUTION

The composition $S(T(v))$ is linear if for any vector sum $u + v = w$ we have $S(T(u)) + S(T(v)) = S(T(w))$ and for any scalar multiple sv we have $S(T(sv)) = sS(T(v))$. This is only a statement of the definition that must be satisfied.

Now let's see why it's true. Suppose first that $u + v = w$ for any given input vectors u and v . Then by the linearity of T , we also know that $T(u) + T(v) = T(w)$. Since this sum holds, linearity of S tells us that the sum is preserved under S : $S(T(u)) + S(T(v)) = S(T(w))$. That means that $S(T(v))$ preserves vector sums.

Similarly for any scalar multiple sv , linearity of T tells us that $sT(v) = T(sv)$. By linearity of S , $sS(T(v)) = S(T(sv))$ as well. This means $S(T(v))$ preserves scalar multiplication, and therefore that $S(T(v))$ satisfies the full definition of linearity above. We can conclude that the composition of two linear transformations is linear.

EXERCISE

For `rotate_x_by(pi/2)`, what are $T(e_1)$, $T(e_2)$, and $T(e_3)$?

SOLUTION

Any rotation about an axis leaves points on the axis unaffected, so since $T(e_1)$ is on the x-axis, $T(e_1) = e_1 = (1,0,0)$. A counterclockwise rotation of $e_2 = (0,1,0)$ in the y,z-plane takes this vector from the point one unit in the positive y-direction to the point one unit in the positive z-direction, so $T(e_2) = e_3 = (0,0,1)$. Likewise, e_3 is rotated counterclockwise from the positive z-direction to the negative y-direction. $T(e_3)$ still has length one in this direction, so it is $-e_2$ or $(0,-1,0)$.

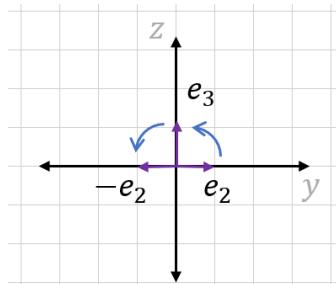


Figure 4.47 A quarter-turn counterclockwise in the y,z plane send e_2 to e_3 and e_3 to $-e_2$.

EXERCISE

Write a `linear_combination(scalars, *vectors)` that takes an iterable of scalars and the same number of vectors and returns a single vector. For example, `linear_combination([1,2,3], (1,0,0), (0,1,0), (0,0,1))` should return $1(1,0,0) + 2(0,1,0) + 3(0,0,1)$ or $(1,2,3)$.

SOLUTION:

```
from vectors import *
def linear_combination(scalars,*vectors):
    scaled = [scale(s,v) for s,v in zip(scalars,vectors)]
    return add(*scaled)
```

We can confirm this gives the expected result from above.

```
>>> linear_combination([1,2,3], (1,0,0), (0,1,0), (0,0,1))
(1, 2, 3)
```

EXERCISE

write a function `transform_standard_basis(transform)` that takes a 3D vector transformation as an input and outputs the effect it has on the standard basis: it should output a tuple of 3 vectors which are the results of `transform` acting on e_1 , e_2 , and e_3 respectively.

SOLUTION

This is as simple as translating the definition above to Python.

```
def transform_standard_basis(transform):
    return transform((1,0,0)), transform((0,1,0)), transform((0,0,1))
```

It confirms (within floating point error) our solution to a previous exercise where we sought this output for `rotate_x_by(pi/2)`.

```
>>> from math import *
>>> transform_standard_basis(rotate_x_by(pi/2))
((1, 0.0, 0.0), (0, 6.123233995736766e-17, 1.0), (0, -1.0,
1.2246467991473532e-16))
```

EXERCISE

Suppose B is a linear transformation, with $B(e_1) = (0,0,1)$, $B(e_2) = (2,1,0)$, and $B(e_3) = (-1,0,-1)$ and $v = (-1,1,2)$. What is $B(v)$?

SOLUTION

Since $v = (-1,1,2) = -e_1 + e_2 + 2e_3$, $B(v) = B(-e_1 + e_2 + 2e_3)$. Since B is linear, it preserves this linear combination: $B(v) = -B(e_1) + B(e_2) + 2B(e_3)$. We have all the information we need from above now: $B(v) = -(0,0,1) + (2,1,0) + 2(-1,0,-1) = (0, 1, -3)$.

EXERCISE

suppose A and B are both linear transformations, with $A(e_1) = (1,1,1)$, $A(e_2) = (1,0,-1)$, and $A(e_3) = (0,1,1)$ and $B(e_1) = (0,0,1)$, $B(e_2) = (2,1,0)$, and $B(e_3) = (-1,0,-1)$. What is $A(B(e_1))$, $A(B(e_2))$, and $A(B(e_3))$?

SOLUTION

$A(B(e_1))$ is A applied to $B(e_1) = (0,0,1) = e_3$. We already know $A(e_3) = (0,1,1)$ so $B(A(e_1)) = (0,1,1)$.

$A(B(e_2))$ is A applied to $B(e_2) = (2,1,0)$. This is a linear combination of $A(e_1)$, $A(e_2)$, and $A(e_3)$ with scalars $(2,1,0)$: $2 \cdot (1,1,1) + 1 \cdot (1,0,-1) + 0 \cdot (0,1,1) = (3,2,1)$.

Finally, $A(B(e_3))$ is A applied to $B(e_3) = (-1,0,-1)$. This is the linear combination $-1(1,1,1) + 0 \cdot (1,0,-1) + -1 \cdot (0,1,1) = (-1,-2,-2)$.

Note that now we know the result of the composition of A and B for all of the standard basis vectors, so we can calculate $A(B(v))$ for any vector v .

4.3 Summary

In this chapter, you learned

- That *vector transformations* are functions which take vectors as inputs and return vectors as outputs. We've seen that vector transformations can operate on 2D or 3D vectors.
- How to apply a vector transformation to every vertex of every polygon of a 3D model, thereby effecting a geometric transformation of the model.
- You can combine existing vector transformations by *composition of functions* to create new transformations, which are equivalent to applying the existing vector transformations sequentially.
- *Functional programming* is a programming paradigm which emphasizes composing and otherwise manipulating functions. The functional operation of *currying* turns a function which takes multiple arguments into a function which takes one argument and returns a new function. Currying allowed us to turn existing functions like `scale` and `add` into vector transformations.
- *Linear transformations* are vector transformations which preserve vector sums and scalar multiples. They are well-behaved since these properties mean they will keep

straight lines straight.

- A *linear combination* is the most general combination of scalar multiplication and vector addition. Every 3D vector is a linear combination of the 3D standard basis vectors, which are denoted $e_1 = (1,0,0)$, $e_2 = (0,1,0)$, and $e_3 = (0,0,1)$. Likewise, every 2D vector is a linear combination of the 2D standard basis vectors, which are $e_1 = (1,0)$ and $e_2 = (0,1)$.
- Once we know how a given linear transformation acts on the standard basis vectors, we can figure out how it acts on *any* vector by writing the vector as a linear combination of the standard basis and using the fact that linear combinations are preserved. In 3D, three vectors or nine total numbers specify a linear transformation. In 2D, two vectors or four total numbers do the same.

This last point is critical: linear transformations are both well behaved and easy to compute with because they can be specified with so little data. We'll explore this in depth in the next chapter, and you'll master computing linear transformations using the notation of *matrices*.

5

Computing Transformations with Matrices

This chapter covers

- Writing a linear transformation as a matrix.
- Multiplying matrices to compose and apply linear transformations.
- Understanding linear functions accepting and outputting vectors of different dimensions.
- Translating vectors in 2D or 3D with matrices.

In the culmination of Chapter 4, you learned quite a big idea: *any* linear transformation in 3D can be specified by just three vectors, or nine numbers total. Rotation by any angle about any axis, reflection across any plane, projection onto any plane, or scaling by any factor in any direction: any of these can be achieved by the right selection of nine numbers.

We could describe a linear transformation as “rotation counterclockwise by 90 degrees about the z-axis,” or we could describe it using the right nine numbers. Either way, we get an imaginary “machine” that behaves the same way: taking 3D vectors as inputs and producing rotated 3D vectors as outputs. The implementations might be different, but the “machines” would produce indistinguishable results.

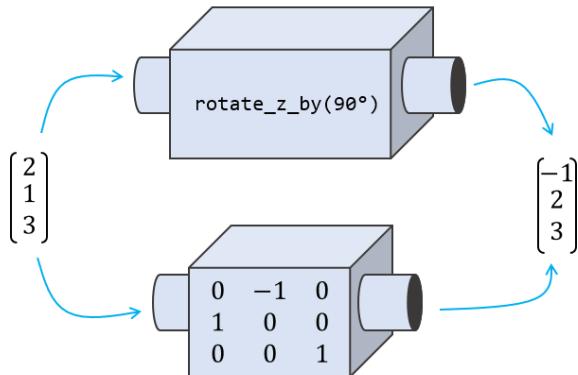


Figure 5.1 Two machines which do the same linear transformation. One is powered by geometric reasoning, the other is powered by nine numbers.

When arranged appropriately in a grid, the numbers that tell us how to execute a linear transformation are called a *matrix*. This chapter is focused on using these grids of numbers as computational tools, so there will be more number-crunching in this chapter than the previous ones. Don't let this intimidate you! All of the mental models remain the same: we're still dealing with linear transformations of vectors.

In this chapter, we're simply focusing in on computing linear transformations using the data of how they transform standard basis vectors. All of the notation serves to organize that process, and not to introduce any new ideas. I know it can feel like a pain to learn a new and complicated notation, but I promise it will pay off. We are better off being able to think of vectors as geometric objects *or* as tuples of numbers, and likewise we'll open new doors when we can think of linear transformations as matrices of numbers.

5.1 Representing linear transformations with matrices

Let's return to a concrete example of the "nine numbers" that specify a 3D linear transformation. Suppose A is a linear transformation and we know $A(\mathbf{e}_1) = (1,1,1)$, $A(\mathbf{e}_2) = (1,0,-1)$, and $A(\mathbf{e}_3) = (0,1,1)$. These three vectors having nine components in total contain all of the information required to specify the linear transformation, A . Since we'll reuse this concept over and over, it warrants a special notation. We'll adopt a new notation called *matrix notation* to work with these nine numbers as a representation of A .

5.1.1 Writing vectors and linear transformations as matrices

Matrices are rectangular grids of numbers, and their shapes tell us how to interpret them. For instance, a matrix which is a single column of numbers is interpreted as a vector, with its entries being the coordinates, ordered top to bottom. In this form, the vectors are called

column vectors. For example, the standard basis for three dimensions can be written as three column vectors:

$$e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad e_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad e_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

This notation conveys the same three facts as $e_1 = (1,0,0)$ $e_2 = (0,1,0)$ and $e_3 = (0,0,1)$. We can indicate how A transforms standard basis vectors in this notation as well.

$$A(e_1) = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad A(e_2) = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \quad A(e_3) = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}.$$

The matrix representing the linear transformation A is the 3-by-3 grid consisting of these vectors squashed together side by side:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix}$$

For 2D vectors, the column vectors will have two entries and the transformations will have four total entries. We can look at the linear transformation D that scales input vectors by a multiple of 2. First we can write down how it works on basis vectors:

$$D(e_1) = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \quad D(e_2) = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

and then the matrix for D is obtained by putting these columns next to each other.

$$D = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}.$$

Matrices can come in other shapes and sizes, but these are the ones we'll focus on for now: the single column matrices representing vectors, and the square matrices representing linear transformations.

Remember, there are new concepts here, only a new way of writing the core idea from the previous section: a linear transformation is defined by its results acting on the standard basis vectors. The recipe to get a matrix from a linear transformation is finding the vectors it produces from all of the standard basis vectors, and then combining the results side-by-side. Now, we'll look at the opposite problem: how to evaluate a linear transformation given its matrix.

5.1.2 Multiplying a matrix with a vector

If a linear transformation B is represented as a matrix, and a vector v is also represented as a matrix (a column vector), we have all of the numbers present required to evaluate $B(v)$. For instance, if B and v are given by

$$B = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix}, \quad v = \begin{pmatrix} 3 \\ -2 \\ 5 \end{pmatrix}$$

then the vectors $B(e_1)$, $B(e_2)$, and $B(e_3)$ can be read off of B as the columns of its matrix. From that point, we use the same procedure as before. Since $v = 3e_1 - 2e_2 + 5e_3$, it follows that $B(v) = 3B(e_1) - 2B(e_2) + 5B(e_3)$. Expanding this, we get:

$$B(v) = 3 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} - 2 \cdot \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} + 5 \cdot \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix} + \begin{pmatrix} -4 \\ -2 \\ 0 \end{pmatrix} + \begin{pmatrix} 5 \\ 0 \\ -5 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ -2 \end{pmatrix}.$$

and the result is the vector $(1, -2, -2)$.

Treating a square matrix as a function that operates on a column vector is a special case of an operation called *matrix multiplication*. Again, this has an impact on our notation and terminology, but we are doing the same thing: applying a linear transformation to a vector. Written as a multiplication, it would look like this:

$$Bv = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 3 \\ -2 \\ 5 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ -2 \end{pmatrix}.$$

As opposed to multiplying numbers, the order matters when you multiply matrices by vectors. In this case Bv is a valid product but vB is not. Shortly, we'll see how to multiply matrices of various shapes, and there will be a general rule for what order matrices can be multiplied. For now, take my word for it, and continue to think of this multiplication as valid because it means applying a 3D linear operator to a 3D vector.

We can write Python code that multiplies a matrix by a vector. Let's say we encode the matrix B as a tuple-of-tuples and the vector v as a tuple as usual:

```
B = (
    (0, 2, 1),
    (0, 1, 0),
    (1, 0, -1)
)
v = (3, -2, -5)
```

This is a little bit different from how we originally thought about the matrix B : we created it by combining three columns, but here B is created as a sequence of rows. The advantage of defining a matrix in Python as a tuple of rows is that the numbers are laid out in the same order as we'd write them on paper. We can get the columns any time we want using Python's `zip` function:

```
>>> zip(*B)
[(0, 0, 1), (2, 1, 0), (1, 0, -1)]
```

The first entry of this list is $(0, 0, 1)$, which is the first column of B , and so on. What we want is the linear combination of these vectors, where the scalars are the coordinates of v . To get this, we can use the `linear_combination` function from a preceding exercise. The first argument to `linear_combination` should be v , which serves as the list of scalars, and the subsequent arguments should be the columns of B . Here's the complete function

```
def multiply_matrix_vector(matrix, vector):
    return linear_combination(vector, *zip(*matrix))
```

It confirms the calculation we did by hand with B and v .

```
>>> multiply_matrix_vector(B,v)
(1, -2, -2)
```

There are two other mnemonic recipes for multiplying a matrix by a vector, both of which give the same results. To see them, let's write out a prototypical matrix multiplication:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

the result of which is the linear combination of the columns of the matrix with the coordinates x , y , and z as the scalars.

$$= x \cdot \begin{pmatrix} a \\ d \\ g \end{pmatrix} + y \cdot \begin{pmatrix} b \\ e \\ h \end{pmatrix} + z \cdot \begin{pmatrix} c \\ f \\ i \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix}$$

This is an explicit formula for the product of a 3-by-3 matrix with a 3D vector, and you could write a similar one for a 2D vector:

$$\begin{pmatrix} j & k \\ l & m \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = x \cdot \begin{pmatrix} j \\ l \end{pmatrix} + y \cdot \begin{pmatrix} k \\ m \end{pmatrix} = \begin{pmatrix} jx + ky \\ lx + my \end{pmatrix}$$

The first mnemonic is that each coordinate of the output vector is a function of the all coordinates of the input vector. For instance, the first coordinate of the 3D output is a function $f(x, y, z) = ax + by + cz$. Moreover, this is a *linear* function in a sense you used the word

in high school algebra; it is a sum of a number times each variable. “Linear transformation” was already a good term, since linear transformations preserve lines, but this gives us another reason to use that term: a linear transformation is a collection of linear functions on the input coordinates that give the respective output coordinates.

The second mnemonic puts the same formula in other words: the coordinates of the output vector are dot products of rows of the matrix with the target vector. For instance, the first row of the 3-by-3 matrix is (a,b,c) and the multiplied vector is (x,y,z) , so the first coordinate of the output is $(a,b,c) \cdot (x,y,z) = ax+by+cz$. We can combine our two notations to state this fact in a formula.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} (a,b,c) \cdot (x,y,z) \\ (d,e,f) \cdot (x,y,z) \\ (g,h,i) \cdot (x,y,z) \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix}$$

If your eyes are starting to glaze over from looking at so many letters and numbers in arrays, don’t worry. The notation can be overwhelming at first, and it takes some time to connect it to your intuition. I’ll give you more examples of matrices in this chapter, and we’ll review and practice everything we’ve covered in the next chapter as well.

5.1.3 Composing linear transformations by matrix multiplication

Among the examples of linear transformations I gave you were rotations, reflections, scalings, and other geometric transformations. What’s more is that any number of linear transformations chained together give us a new linear transformation. In math jargon, the *composition* of any number of linear transformations is also a linear transformation.

Since any linear transformation can be represented by a matrix, any two composed linear transformations can be as well. In fact, if you want to compose linear transformations to build new ones, matrices are the best tools for the job.

NOTE

Let me take off my mathematician hat and put on my programmer hat for a moment. Suppose you wanted to compute the result of, say, 1000 composed linear transformations operating on a vector. This could come up if you are animating an object by applying additional, small transformations with every frame of the animation. In Python, it would be very computationally expensive to apply 1000 sequential functions, since there is overhead associated with every function call. However, if you were to find a matrix representing the composition of 1000 linear transformations, you would boil the whole process down to a handful of numbers and a handful of computations.

Let’s look at a composition of two linear transformations: $A(B(v))$ where the matrix representations of A and B are known to be the following.

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix}$$

Here's how the composition works step by step. First, the transformation B is applied to v, yielding a new vector B(v), or Bv if we're writing it as a multiplication. Second, this vector becomes the input to the transformation A, yielding a final 3D vector as a result: A(Bv). Once again, we'll drop the parentheses and write A(Bv) as the product ABv. Writing this product out for v = (x,y,z) would give us a formula that looks like this:

$$ABv = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

If we work right-to-left, we know how to evaluate this. Now I'm going to claim that we can work left-to-right as well and get the same result. Specifically we can ascribe meaning to the product matrix "AB" on its own; it will be a new matrix (to be discovered) representing the composition of the linear transformations A and B.

$$AB = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{pmatrix}$$

Now, what should the entries of this new matrix be? Its purpose is to represent the composition of the transformations A and B, which give us a new linear transformation, AB. As we've seen, the columns of a matrix are the results of applying its transformation to standard basis vectors. The columns of the matrix AB will be the result of applying the transformation AB to each of e₁, e₂, and e₃.

The columns of AB are therefore AB(e₁), AB(e₂) and AB(e₃). Let's look at the first column for instance, which should be AB(e₁), or A applied to the vector B(e₁). In other words, to get the first column of AB, we do the operation of multiplying a matrix by a vector that we already practiced:

$$AB = \begin{array}{c|ccc} & A & B(e_1) & AB(e_1) \\ \hline AB = & \left(\begin{array}{ccc} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{array} \right) & \left(\begin{array}{ccc} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{array} \right) & = \left(\begin{array}{ccc} 0 & ? & ? \\ 1 & ? & ? \\ 1 & ? & ? \end{array} \right) \end{array}$$

Similarly, we can find that AB(e₂) = (3,2,1) and AB(e₃) = (1,0,0) which are the second and third columns of AB.

$$AB = \begin{pmatrix} 0 & 3 & 1 \\ 1 & 2 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

That's how matrix multiplication is done, and you can see there's nothing to it besides carefully composing linear operators. Similarly, you can use mnemonics instead of reasoning through this process each time. Since multiplying a 3-by-3 matrix by a column vector is the same as doing three dot products, multiplying two 3-by-3 matrices together is the same as doing *nine* dot products -- all possible dot products of rows of the first with columns of the second.

$$\begin{aligned} B &= \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \\ A &= \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 3 & 1 \\ 1 & 2 & 0 \\ 1 & 1 & 0 \end{pmatrix} = AB \\ (1, 0, 1) \cdot (2, 1, 0) &= 1 \cdot 2 + 0 \cdot 1 + 1 \cdot 0 = 2 \end{aligned}$$

Figure 5.2 Each entry of a product matrix is a dot product of a row of the first matrix with a column of the second matrix.

Everything we've said about 3-by-3 matrix multiplication applies to 2-by-2 matrices as well. For instance, to find the product of the following 2-by-2 matrices

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

we can take the dot products of rows of the first with columns of the second. The dot product of the first row of the first matrix with the first column of the second matrix is $(1, 2) \cdot (0, 1) = 2$. This tells us that the entry in the first row and first column of the result matrix is 2.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & ? \\ ? & ? \end{pmatrix}$$

Repeating this procedure, we can find all the entries of the product matrix:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & -1 \\ 4 & -3 \end{pmatrix}$$

You can do some matrix multiplication as an exercise to get a hang of it, but you'll quickly prefer your computer doing the work for you. Let's implement matrix multiplication in Python to make this possible.

5.1.4 Implementing matrix multiplication

There are a few ways we could write our matrix multiplication function, but I prefer using the dot product trick. Since the result of matrix multiplication should be a tuple of tuples, we can write it as a nested comprehension. It will take in two nested tuples as well, representing our input matrices `a` and `b`. The input matrix `a` is already a tuple of rows of the first matrix, and we can pair these up with `zip(*b)` which is a tuple of columns of the second matrix. Finally, for each pair we should take the dot product and yield it in the inner comprehension. Here's the implementation:

```
from vectors import *

def matrix_multiply(a,b):
    return tuple(
        tuple(dot(row,col) for col in zip(*b))
        for row in a
    )
```

The outer comprehension builds the rows of the result, and the inner one builds the entries of each row. Since the output rows are formed by the various dot products with rows of `a`, the outer comprehension iterates over `a`.

Fortunately, our `matrix_multiply` function doesn't have any hard-coded dimensions. We can successfully use it to do the matrix multiplications from both preceding examples in an interactive session:

```
>>> from matrices import *
>>> a = ((1,1,0),(1,0,1),(1,-1,1))
>>> b = ((0,2,1),(0,1,0),(1,0,-1))
>>> matrix_multiply(a,b)
((0, 3, 1), (1, 2, 0), (1, 1, 0))
>>> c = ((1,2),(3,4))
>>> d = ((0,-1),(1,0))
>>> matrix_multiply(c,d)
((2, -1), (4, -3))
```

Equipped with the computational tool of matrix multiplication, we can now do some easy manipulations of our 3D graphics.

5.1.5 3D Animation with matrix transformations

To animate a 3D model, we'll redraw a transformed version of the original model each frame. To make the model appear to move or change over time, we'll need to use different transformations as time progresses. If these transformations are linear transformations specified by matrices, we need a new matrix for every new frame of the animation.

Since PyGame's built-in clock keeps track of time (in milliseconds), one thing we can do is generate matrices whose entries depend on time. In other words, instead of thinking of every entry of a matrix as a number, we can think of it as a function which takes the current time, t , and returns a number.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \rightarrow \begin{pmatrix} a(t) & b(t) & c(t) \\ d(t) & e(t) & f(t) \\ g(t) & h(t) & i(t) \end{pmatrix}$$

Figure 5.3 Thinking of entries of a matrix as functions of time, allowing the overall matrix to change as time passes.

For instance, we could use these nine expressions:

$$\begin{pmatrix} \cos(t) & 0 & -\sin(t) \\ 0 & 1 & 0 \\ \sin(t) & 0 & \cos(t) \end{pmatrix}$$

As we covered in chapter 2, cosine and sine are both functions that take a number and return another number as a result. The other five entries happen to not change over time, but if you crave consistency you can think of them as constant functions, as in $f(t) = 0$. Given any value of t , this matrix happens to represent the same linear transformation as `rotate_y_by(t)`. Time moves forward and the value of t increases, so if we apply this matrix transformation each frame we'll get a bigger rotation each time.

Let's give our `draw_model` function a `get_matrix` keyword argument, where the value passed to `get_matrix` will be a function that takes time in milliseconds and returns the transformation matrix that should be applied at that time. For instance, we might want to call it like this to animate the rotating teapot:

```
from teapot import load_triangles
from draw_model import draw_model
from math import *

def get_rotation_matrix(t):
    seconds = t/1000
    return (
        (cos(seconds),0,-sin(seconds)),
        (0,1,0),
        (sin(seconds),0,cos(seconds))
    )
```

1
2

```
draw_model(load_triangles(), get_matrix=get_rotation_matrix) ③
```

- ① A function which generates a new transformation matrix for any numeric input representing time.
- ② Let's convert the time to seconds so the transformation doesn't happen too fast.
- ③ The function is passed as a keyword argument to draw_model.

Now, `draw_model` is passed the data required to transform the underlying teapot model over time, but we need to use it in the function body. Before iterating over the faces, we execute the appropriate matrix transformation:

```
def draw_model(faces, color_map=blues, light=(1,2,3),
              camera=Camera("default_camera", []),
              glRotatefArgs=None,
              get_matrix=None):
    #...
    def do_matrix_transform(v):
        if get_matrix:
            m = get_matrix(pygame.time.get_ticks())
            return multiply_matrix_vector(m, v)
        else:
            return v
    transformed_faces = polygon_map(do_matrix_transform, faces)
    for face in transformed_faces:
        #... #6
```

- ① Since most of the function body is unchanged, we don't print it here.
- ② Inside the main "while" loop, create a new function which will apply the matrix for this frame.
- ③ Use the elapsed milliseconds, given by `pygame.time.get_ticks()`, as well as the provided `get_matrix` function to compute a matrix for this frame
- ④ If no `get_matrix` was specified, don't carry out any transformation and return the vector unchanged.
- ⑤ Finally, apply the function to every polygon using `polygon_map`.

With these changes, you can run the code and see the teapot rotate.



Figure 5.4 The teapot is transformed by a new matrix every frame, depending on the elapsed time when the frame is drawn.

Hopefully I've convinced you with the preceding examples that matrices are entirely interchangeable with linear transformations: we've managed to transform and animate the teapot the same way, using only nine numbers to specify each transformation. You can practice your matrix skills some more in the exercises, and then I'll show you there's even more to learn from the `matrix_multiply` function we've already implemented.

5.1.6 Exercises

EXERCISE

write a function `infer_matrix(n, transformation)` that takes in a dimension (like 2 or 3) and a function which is a vector transformation assumed to be linear. It should return an n-by-n square matrix, i.e. an n-tuple of n-tuples of numbers which is the matrix representing the linear transformation. Of course, the output will only meaningful if the input transformation is linear. Otherwise it is meaningless!

SOLUTION:

```
def infer_matrix(n, transformation):
    def standard_basis_vector(i):
        return tuple(1 if i==j else 0 for j in range(1,n+1)) #1
    standard_basis = [standard_basis_vector(i) for i in range(1,n+1)] #2
    cols = [transformation(v) for v in standard_basis] #3
    return tuple(zip(*cols)) #4
```

- 1) This function creates the i th standard basis vector as a tuple containing a one in the i th coordinate and zeroes in all other coordinates.
- 2) The standard basis is created as a list of n vectors.
- 3) We defined the columns of a matrix to be the result of applying the corresponding linear transformation to the standard basis vectors.
- 4) Reshape the matrix to be a tuple of rows instead of a list of columns, following our convention.

We can test this on a linear transformation, like `rotate_z_by(pi/2)`:

```
>>> from transforms import rotate_z_by
>>> from math import pi
>>> infer_matrix(3,rotate_z_by(pi/2))
((6.123233995736766e-17, -1.0, 0.0), (1.0, 1.2246467991473532e-16, 0.0), (0, 0, 1))
```

EXERCISE

What is the result of the following product of a 2-by-2 matrix with a 2D vector?

$$\begin{pmatrix} 1.3 & -0.7 \\ 6.5 & 3.2 \end{pmatrix} \begin{pmatrix} -2.5 \\ 0.3 \end{pmatrix}$$

SOLUTION

The dot product of the vector with the first row of the matrix is $-2.5 \cdot 1.3 + 0.3 \cdot -0.7 = -3.46$. The dot product of the vector with the second row of the matrix is $-2.5 \cdot 6.5 + 0.3 \cdot 3.2 = -15.29$. These are the coordinates of the output vector, so the result is:

$$\begin{pmatrix} 1.3 & -0.7 \\ 6.5 & 3.2 \end{pmatrix} \begin{pmatrix} -2.5 \\ 0.3 \end{pmatrix} = \begin{pmatrix} -3.46 \\ -15.29 \end{pmatrix}$$

MINI PROJECT

Write a `random_matrix` function that generates matrices of a specified size with random whole-number entries. Use it to generate five pairs of 3-by-3 matrices. Multiply each of the pairs together by hand for practice, and then check your work with the `matrix_multiply` function.

SOLUTION

I gave the `random_matrix` function arguments to specify the number of rows, the number of columns, and the minimum and maximum values for entries.

```
def random_matrix(rows,cols,min=-2,max=2):
    return tuple(
        tuple(
            randint(min,max) for j in range(0,cols))
        for i in range(0,rows)
    )
```

So I could generate a random 3-by-3 matrix with entries between 0 and 10 as follows:

```
>>> random_matrix(3,3,0,10)
((3, 4, 9), (7, 10, 2), (0, 7, 4))
```

EXERCISE

For each of your pairs of matrices from the previous exercise, try multiplying them in the opposite order. Do you get the same result?

SOLUTION

Unless you get very lucky, your results will all be different. Most pairs of matrices give different results when multiplied in different orders. In math jargon, we say an operation is *commutative* if it gives the same result regardless of the order of inputs. For instance, multiplying numbers is a commutative operation since $xy = yx$ for any choice of numbers x and y . However, matrix multiplication is *not* commutative since for two square matrices A and B , AB does not always equal BA .

EXERCISE

In either 2D or 3D, there is a boring but important vector transformation that takes in a vector and returns the same vector as an output. This transformation is linear, because it takes any input vector sum, scalar multiple, or linear combination and gives the exact same thing back as an output. What are the matrices representing the identity transformation in 2D and 3D respectively?

SOLUTION

In 2D or 3D, the identity transformation acts on the standard basis vectors and leaves them unchanged. Therefore, in either dimension, the matrix for this transformation has the standard basis vectors as its columns, in order. In 2D and 3D these *identity matrices* are denoted I_2 and I_3 respectively, and they look like this:

$$I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

EXERCISE

apply the matrix $((2,1,1), (1,2,1), (1,1,2))$ to all the vectors defining the teapot. What happens to the teapot and why?

SOLUTION

Applying the matrix as follows,

```
def transform(v):
    m = ((2,1,1),(1,2,1),(1,1,2))
    return multiply_matrix_vector(m,v)
draw_model(polygon_map(transform, load_triangles()))
```

we see that the front of the teapot is stretched out into the region where x , y , and z are all positive.

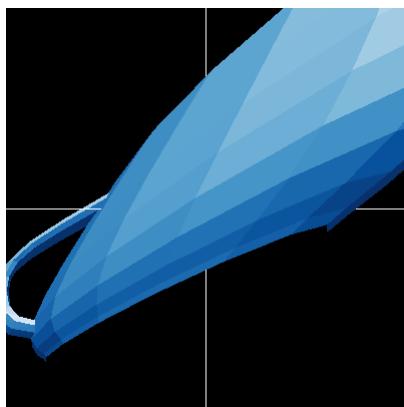


Figure 5.5 Applying the given matrix to all vertices of the teapot.

This is because all of the standard basis vectors are transformed to vectors with positive coordinates: $(2,1,1)$, $(1,2,1)$, and $(1,1,2)$ respectively.

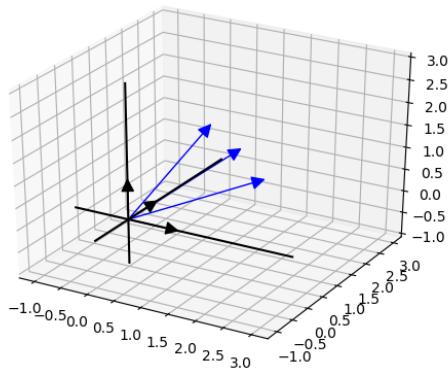


Figure5.6: How the linear transformation defined by this matrix affects the standard basis vectors.

A linear combination of these new vectors with positive scalars will be stretched further in the +x, +y, and +z directions than the same linear combination of the standard basis.

EXERCISE

Implement `multiply_matrix_vector` a different way by using two nested comprehensions: one traversing the rows of the matrix and one traversing the entries of each row

SOLUTION:

```
def multiply_matrix_vector(matrix,vector):
    return tuple(
        sum(vector_entry * matrix_entry
            for vector_entry, matrix_entry in zip(row,vector))
        for row in matrix
    )
```

EXERCISE

Implement `multiply_matrix_vector` yet another way using the fact that the output coordinates are dot products of the input matrix rows with the input vector.

SOLUTION

This is a simplified version of the previous exercise solution.

```
def multiply_matrix_vector(matrix,vector):
    return tuple(
        dot(row,vector)
        for row in matrix
    )
```

MINI PROJECT

I first told you what a linear transformation was, and then showed you that any linear transformation can be represented by a matrix. Let's prove the *converse* fact now, that all matrices represent linear transformations.

Starting with the explicit formulas for multiplying a 2D vector by a 2-by-2 matrix or multiplying a 3D vector by a 3-by-3 matrix, prove that bot. That is, show that matrix multiplication preserves sums and scalar multiples.

SOLUTION

I'll show the proof for 2D, and the 3D proof has the same structure but with a bit more writing. Suppose we have a 2-by-2 matrix called A with *any* four numbers a, b, c , and d as its entries. Let's see how A operates on two vectors u and v .

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}, \quad v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

You can do the matrix multiplications explicitly to find Au and Av :

$$Au = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} au_1 + bu_2 \\ cu_1 + du_2 \end{pmatrix}$$

$$Av = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} av_1 + bv_2 \\ cv_1 + dv_2 \end{pmatrix}$$

And then we can compute $Au + Av$ and $A(u+v)$ and see that they match.

$$Au + Av = \begin{pmatrix} au_1 + bu_2 \\ cu_1 + du_2 \end{pmatrix} + \begin{pmatrix} av_1 + bv_2 \\ cv_1 + dv_2 \end{pmatrix} = \begin{pmatrix} au_1 + bu_2 + av_1 + bv_2 \\ cu_1 + du_2 + cv_1 + dv_2 \end{pmatrix}$$

$$A(u+v) = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} u_1 + v_1 \\ u_2 + v_2 \end{pmatrix} = \begin{pmatrix} a(u_1 + v_1) + b(u_2 + v_2) \\ c(u_1 + v_1) + d(u_2 + v_2) \end{pmatrix} = \begin{pmatrix} au_1 + av_1 + bu_2 + bv_2 \\ cu_1 + cv_1 + du_2 + dv_2 \end{pmatrix}$$

This tells us that the 2D vector transformation defined by multiplying *any* 2-by-2 matrix preserves vector sums. Likewise, for any number s , we have:

$$sv = \begin{pmatrix} sv_1 \\ sv_2 \end{pmatrix}$$

$$s(Av) = \begin{pmatrix} s(av_1 + bv_2) \\ s(cv_1 + dv_2) \end{pmatrix} = \begin{pmatrix} sav_1 + sbv_2 \\ scv_1 + sdv_2 \end{pmatrix}$$

$$A(sv) = \begin{pmatrix} a(sv_1) + b(sv_2) \\ c(sv_1) + d(sv_2) \end{pmatrix} = \begin{pmatrix} sav_1 + sbv_2 \\ scv_1 + sdv_2 \end{pmatrix}$$

So $s(Av)$ and $A(sv)$ give the same results, and we see that multiplying by the matrix A preserves scalar multiples as well. These two facts mean that multiplying by any 2-by-2 matrix is a linear transformation of 2D vectors.

EXERCISE

Given the matrices A and B from the preceding section, write a function `compose_a_b` that executes the composition of the linear transformation for A and the linear transformation for B . Then, use the `infer_matrix` function from a preceding exercise to show that `infer_matrix(3, compose_a_b)` is the same as the matrix product AB .

SOLUTION:

First, we implement two functions `transform_a` and `transform_b` that do the linear transformations defined by matrices A and B . Then, we compose them together using our `compose` function

```
from matrices import *
from transforms import *

a = ((1,1,0),(1,0,1),(1,-1,1))
b = ((0,2,1),(0,1,0),(1,0,-1))

def transform_a(v):
    return multiply_matrix_vector(a,v)

def transform_b(v):
    return multiply_matrix_vector(b,v)

compose_a_b = compose(transform_a, transform_b)
```

Then, we can use our `infer_matrix` function to find the matrix corresponding to this composition of linear transformations, and compare it to the matrix product AB .

```
>>> infer_matrix(3, compose_a_b)
((0, 3, 1), (1, 2, 0), (1, 1, 0))
>>> matrix_multiply(a,b)
((0, 3, 1), (1, 2, 0), (1, 1, 0))
```

MINI PROJECT

Find two 2-by-2 matrices, neither of which is the identity matrix I_2 , but whose product *is* the identity matrix.

SOLUTION

One way to do this is to write down two matrices and play with their entries until you get the identity matrix as a product. Another way is to think of the problem in terms of linear transformations. If two matrices multiply together to produce the identity matrix, then the composition of their corresponding linear transformations should produce the identity transformation.

With that in mind, what are two 2D linear transformations whose composition is the identity transformation? When applied in sequence to a given 2D vector, these linear transformations should return the original vector as output. One such pair of transformations is rotation by 90 degrees clockwise and rotation by 270 degrees clockwise. Applying both of these executes a 360 degree rotation, which brings any vector back to its original

position. The matrices for a 270 degree rotation and a 90 degree rotation are the following, and their product is the identity matrix.

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

EXERCISE

We can multiply a square matrix by itself any number of times. We can think of successive matrix multiplications as “raising a matrix to a power.” For a square matrix A, A·A could be written A^2 , A·A·A could be written A^3 , and so on. Write a `matrix_power(power,matrix)` function that raises a matrix to the specified (whole number) power.

SOLUTION

Here is an implementation that will work for whole number powers greater than or equal to 1.

```
def matrix_power(power,matrix):
    result = matrix
    for _ in range(1,power):
        result = matrix_multiply(result,matrix)
    return result
```

5.2 Interpreting matrices of different shapes

The `matrix_multiply` function doesn’t hard-code the size of the input matrices, so we can use it to multiply either two-by-two or three-by-three matrices together. As it turns out, it can also handle matrices of other sizes as well. For instance, it can handle these two five-by-five matrices:

```
>>> a = ((-1, 0, -1, -2, -2), (0, 0, 2, -2, 1), (-2, -1, -2, 0, 1), (0, 2, -2, -1,
          0), (1, 1, -1, -1, 0))
>>> b = ((-1, 0, -1, -2, -2), (0, 0, 2, -2, 1), (-2, -1, -2, 0, 1), (0, 2, -2, -1,
          0), (1, 1, -1, -1, 0))
>>> matrix_multiply(a,b)
((-10, -1, 2, -7, 4), (-2, 5, 5, 4, -6), (-1, 1, -4, 2, -2), (-4, -5, -5, -9, 4), (-1,
     -2, -2, -6, 4))
```

There’s no reason we shouldn’t take this result seriously -- our functions for vector addition, scalar multiplication, dot products, and therefore matrix multiplication don’t depend on the dimension of the vectors we use. Even though we can’t picture a five-dimensional vector, we can do all the same algebra on five-tuples of numbers that we did on pairs and triples of numbers in 2D and 3D respectively. In this “5D” product, the entries of the resulting matrix are still dot products of rows of the first matrix with columns of the second.

$$\begin{array}{c}
 \text{second row} \\
 \left(\begin{array}{ccccc} -1 & 0 & -1 & -2 & -2 \\ 0 & 0 & 2 & -2 & 1 \\ -2 & -1 & -2 & 0 & 1 \\ 0 & 2 & -2 & -1 & 0 \\ 1 & 1 & -1 & -1 & 0 \end{array} \right) \left(\begin{array}{ccccc} 2 & 0 & 0 & -1 & 2 \\ -1 & -2 & -1 & -2 & 0 \\ 0 & 1 & 2 & 2 & -2 \\ 2 & -1 & -1 & 1 & 0 \\ 2 & 1 & -1 & 2 & -2 \end{array} \right) = \left(\begin{array}{ccccc} -10 & -1 & 2 & -7 & 4 \\ -2 & 5 & 5 & 4 & -6 \\ -1 & 1 & -4 & 2 & -2 \\ -4 & -5 & -5 & -9 & 4 \\ -1 & -2 & -2 & -6 & 4 \end{array} \right)
 \end{array}$$

$(0,0,2,-2,1) \cdot (-1,-2,2,1,2) = 4$

Figure 5.7 The dot product of a row of the first matrix with a column of the second matrix produces one entry of the matrix product.

You can't visualize it in the same way, but you can show algebraically that a five-by-five matrix specifies a linear transformation of five-dimensional vectors. We'll spend time talking about what kind of objects live in four, five, or more dimensions in the next chapter.

5.2.1 Column vectors as matrices

Let's return to the example of multiplying a matrix by a column vector. I already showed you how to do a multiplication like this, but we treated it as its own case with the `multiply_matrix_vector` function. It turns out `matrix_multiply` is capable of doing these products as well, but we have to write the column vector as a matrix.

As an example, let's try to pass the following square matrix and single-column matrix to our `matrix_multiply` function.

$$C = \begin{pmatrix} -1 & -1 & 0 \\ -2 & 1 & 2 \\ 1 & 0 & -1 \end{pmatrix}, \quad D = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

I claimed before that you can think of a vector and a single-column matrix interchangeably, so we might encode D as a vector $(1, 1, 1)$. But this time let's force ourselves to think of it as a matrix, having three rows with one entry each:

```
>>> c = ((-1, -1, 0), (-2, 1, 2), (1, 0, -1))
>>> d = ((1),(1),(1))
>>> matrix_multiply(c,d)
((-2,), (1,), (0,))
```

The result has three rows with one entry each, so it is a single-column matrix as well. Here's what this product looks like in matrix notation:

$$\begin{pmatrix} -1 & -1 & 0 \\ -2 & 1 & 2 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}$$

Our `multiply_matrix_vector` function can evaluate the same product but in a different format:

```
>>> multiply_matrix_vector(c,(1,1,1))
(-2, 1, 0)
```

This demonstrates that multiplying a matrix and a column vector is a special case of matrix multiplication. We don't need a separate function `multiply_matrix_vector` after all. We can further see that the entries of the output are dot products of the rows of the first matrix with the single column of the second.

$$\begin{pmatrix} -1 & -1 & 0 \\ -2 & 1 & 2 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}$$

Figure 5.8 An entry of the resulting vector, computed as a dot product.

On paper, you'll see vectors represented interchangeably as tuples (with commas) or as column vectors. But for the Python functions we've written, the distinction is critical. The tuple `(-2, 1, 0)` can't be used interchangeably with the tuple-of-tuples `((-2,), (1,), (0,))`. Yet another way of writing the same vector would be as a *row vector*, or a matrix with one row. Here are the three notations for comparison:

Representation	In math notation	In Python
Ordered triple (ordered tuple)	$v = (-2, 1, 0)$	<code>v = (-2, 1, 0)</code>
Column vector	$v = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}$	<code>v = ((-2,), (1,), (0,))</code>
Row vector	$v = \begin{pmatrix} -2 & 1 & 0 \end{pmatrix}$	<code>v = ((-2, 1, 0),)</code>

If you've saw this comparison in math class before, you may have thought it was a pedantic notational distinction. Once we represent them in Python however, we see that they are really three different objects that need to be treated differently. While they all represent the same geometric data, which is a 3D arrow or point in space, only one of them, the column vector, can be multiplied by a three-by-three matrix. The row vector doesn't work since we can't take the dot product of a row of the first matrix with a column of the second:

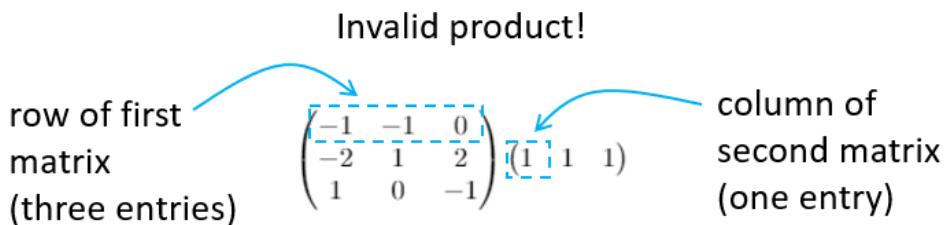


Figure 5.9 Two matrices which cannot be multiplied together.

So, for our definition of matrix multiplication to be consistent, we can only multiply a matrix on the left of a *column* vector. This prompts the general question:

5.2.2 What pairs of matrices can be multiplied?

We can make grids of numbers of any dimensions. When can our matrix multiplication formula work, and what does it mean when it does?

The answer is that the number of columns of the first matrix has to match the number of rows of the second. This is clear when we do the matrix multiplication in terms of dot products. For instance, we can multiply any matrix with three columns by a second matrix with three rows. This means that rows of the first matrix and columns of the second will each have three entries, so we can take their dot products. Here, the dot product of the first row of the first matrix with the first column of the second matrix give us an entry of the product matrix:

$$\begin{matrix} \text{first row} & \begin{pmatrix} 1 & -2 & 0 \end{pmatrix} \end{matrix} \begin{pmatrix} 2 & 0 & -1 & 2 \\ 0 & -2 & 2 & -2 \\ -1 & -1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & ? & ? & ? \end{pmatrix} \text{ first row, first column}$$

Figure 5.10 Finding the first entry of the product matrix.

We can complete this matrix product by taking the remaining seven dot products, another one of which is shown here:

$$\begin{pmatrix} 1 & -2 & 0 \\ -1 & -2 & 2 \end{pmatrix} \begin{pmatrix} 2 & 0 & -1 & 2 \\ 0 & -2 & 2 & -2 \\ -1 & -1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 3 & 6 \\ -4 & 2 & 1 & 4 \end{pmatrix}$$

Second row, third column

Figure 5.11 Finding another entry of the product matrix.

This constraint also makes sense in terms of our original definition of matrix multiplication: the columns of the output are each linear combinations of the columns of the first matrix, with scalars given by a row of the second matrix.

$$\begin{pmatrix} 1 & -2 & 0 \\ -1 & -2 & 2 \end{pmatrix} \begin{pmatrix} 2 & 0 & -1 & 2 \\ 0 & -2 & 2 & -2 \\ -1 & -1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 3 & 6 \\ -4 & 2 & 1 & 4 \end{pmatrix}$$

Three column vectors

Linear combination

Three scalars

Result column

$$\begin{pmatrix} 1 \\ -1 \end{pmatrix} \begin{pmatrix} -2 \\ -2 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \end{pmatrix} \rightarrow -1 \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} + 2 \cdot \begin{pmatrix} -2 \\ -2 \end{pmatrix} + 2 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

Figure 5.12 Each column of the result is a linear combination of the columns of the first matrix.

I was calling the square matrices above “two-by-two” and “three-by-three” matrices. This last example was the product of a “two-by-three” and “three-by-four” matrix. Whenever we describe the *dimensions of a matrix* like this, we say the number of rows first, and then the number of columns. For instance, a 3D column vector would be a “three-by-one” matrix. Sometimes you’ll see matrix dimensions written with a multiplication sign, as in a “3 x 3 matrix” or a “3 x 1 matrix.” In this language, we can make a very general statement about the shapes of matrices that can be multiplied.

FACT

You can only multiply an $n \times m$ matrix by a $p \times q$ matrix if $m = p$. When that is true, the resulting matrix will be a $n \times q$ matrix.

For instance, a 17×9 matrix cannot be multiplied by a 6×11 matrix. However, a 5×8 matrix can be multiplied by a 8×10 matrix. The result of the latter will be a 5×10 matrix.

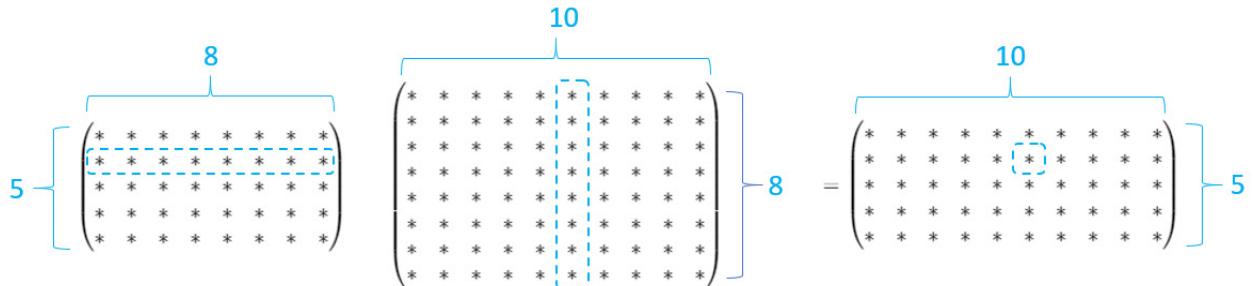


Figure 5.13 Each of the five rows of the first matrix can be paired with one of the ten columns of the second to produce one of the $5 \times 10 = 50$ entries of the product matrix. I used stars instead of numbers to show you that *any* matrices of these sizes are compatible.

By contrast, you couldn't multiply these matrices in the opposite order: a 10×8 matrix can't be multiplied by a 5×8 matrix.

Alright -- enough of this mystical numerology. Our matrix multiplication algorithm happens to work on various sized matrices, but can we learn anything from this? It turns out we can: *all* matrices represent vector functions, and all valid matrix products can be interpreted as composition of these functions. Let's take a look.

5.2.3 Viewing square and non-square matrices as vector functions

We can think of a 2-by-2 matrix as the data required to do a given linear transformation of a 2D vector. Pictured as a machine, this transformation takes a 2D vector into its input slot and produces a 2D vector out of its output slot as a result.

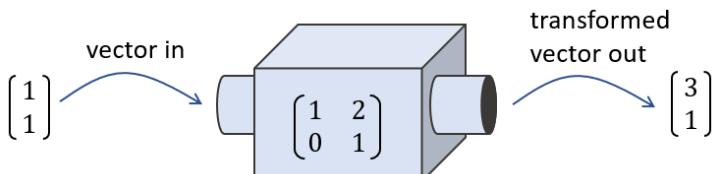


Figure 5.14 Thinking of a matrix as a machine that takes in vectors and produces vectors.

Under the hood, it's doing this matrix multiplication:

$$\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

So it's fair to think of matrices as machines take vectors as input and produce vectors as output. This matrix can't take *any* vector as input though: it is a 2×2 matrix so it does a linear transformation of 2D vectors. Correspondingly, this matrix can only be multiplied by a column

vector with two entries. Let's bifurcate the machine's input and output slots to suggest that they take and produce 2D vectors, or pairs of numbers:

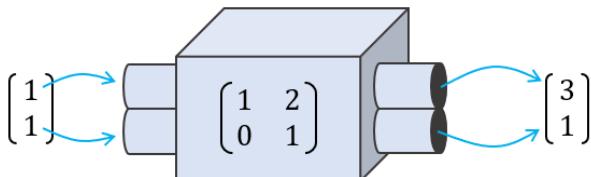


Figure 5.15 Refining our mental model; re-drawing the machine's input and output slots to indicate that its inputs and outputs are pairs of numbers.

Likewise, a linear transformation machine powered by a 3×3 matrix can only take in 3D vectors and produce 3D vectors as a result:

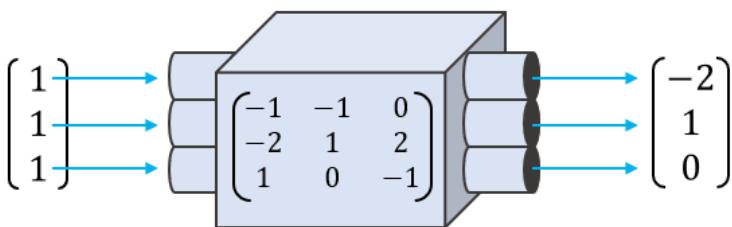


Figure 5.16 A “linear transformation machine” powered by a 3×3 matrix takes in 3D vectors and outputs 3D vectors.

Now we can ask ourselves, what would a machine look like if it were powered by a non-square matrix? As a specific example, what kinds of vectors could this 2×3 matrix act on?

$$\begin{pmatrix} -2 & -1 & -1 \\ 2 & -2 & 1 \end{pmatrix}$$

If we're going to multiply this matrix with a column vector, the column vector will have to have three entries to match the size of the rows of this matrix. Multiplying our 2×3 matrix by a 3×1 column vector will give us a 2×1 matrix as a result, or a 2D column vector. For example:

$$\begin{pmatrix} -2 & -1 & -1 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$$

This tells us that this 2×3 matrix represents a function taking 3D vectors to 2D vectors. If we were to draw it as a machine, it would accept 3D vectors in its input slot and produce 2D vectors from its output slot.

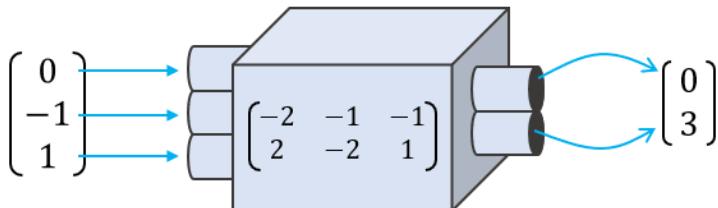


Figure 5.17 A machine which takes in 3D vectors and outputs 2D vectors, powered by a 2×3 matrix.

In general, an $m \times n$ matrix defines a function taking n -dimensional vectors as inputs and returning m -dimensional vectors as outputs. Any such function is *linear* in the sense that it preserves vector sums and scalar multiples. It's not a "transformation" since it doesn't just modify its input, it returns an entirely different *kind* of output: a vector living in a different number of dimensions. For this reason, we'll use more general terminology -- we'll call it a *linear function* or a *linear map*. Let's do an in-depth example of a familiar *linear map* from 3D to 2D.

5.2.4 Projection as a linear map from 3D to 2D

We already saw a vector function which accepts 3D vectors and produces 2D vectors: projection of a 3D vector onto the x,y -plane. This transformation (we can call it P) takes vectors of the form (x,y,z) and returns them with their z -component deleted: (x,y) . Let's spend time carefully showing why this is a linear map and how it preserves vector addition and scalar multiplication.

First of all, let's try to write P as a matrix. To accept 3D vectors and return 2D vectors, it should be a 2×3 matrix. Let's follow our trusty recipe for finding a matrix by testing the action of P on standard basis vectors. When we project $e_1 = (1,0,0)$, $e_2 = (0,1,0)$, and $e_3 = (0,0,1)$ we get $(1,0)$, $(0,1)$, and $(0,0)$ respectively. We can write these as column vectors:

$$P(e_1) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad P(e_2) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad P(e_3) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

and then stick them together side-by-side to get a matrix.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

To check this, let's multiply it by a test vector (a,b,c) . The dot product of (a,b,c) with $(1,0,0)$ is a , so that's the first entry of the result. The second entry is the dot product of (a,b,c) with $(0,1,0)$, or b . You can picture this matrix as grabbing a and b from (a,b,c) and ignoring c .

$$\begin{aligned} & 1 \cdot a + 0 \cdot b + 0 \cdot c \\ \left(\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array} \right) \left(\begin{array}{c} a \\ b \\ c \end{array} \right) &= \left(\begin{array}{c} a \\ b \end{array} \right) \\ & 0 \cdot a + 1 \cdot b + 1 \cdot c \end{aligned}$$

Figure 5.18 Only $1 \cdot a$ contributes to the first entry of the product, and only $1 \cdot b$ contributes to the second entry. The other entries are zeroed out.

This matrix does what we wanted: it deletes the third coordinate of a 3D vector, leaving us with the first two coordinates only. It's good news that we can write this projection as a matrix, but let's also give an algebraic *proof* that this is a linear map. To do this, we have to show the two key conditions of linearity are satisfied.

PROVING PROJECTION PRESERVES VECTOR SUMS

If P is linear, any vector sum $u + v = w$ should be respected by P . That is $P(u) + P(v)$ should equal $P(w)$ as well. Let's confirm this, using

$$u = (u_1, u_2, u_3) \text{ and } v = (v_1, v_2, v_3).$$

Then $w = u + v$ so

$$w = (u_1 + v_1, u_2 + v_2, u_3 + v_3).$$

Executing P on all of these vectors is simple, since we only need to remove the third coordinates:

$$P(u) = (u_1, u_2),$$

$$P(v) = (v_1, v_2),$$

and

$$P(w) = (u_1 + v_1, u_2 + v_2).$$

Adding $P(u)$ and $P(v)$ we get $(u_1 + v_1, u_2 + v_2)$ which is the same as $P(w)$. Therefore for any three 3D vectors $u + v = w$, we will also have $P(u) + P(v) = P(w)$. This checks our first box.

PROVING PROJECTION PRESERVES SCALAR MULTIPLES

The second thing we need to show is that P preserves scalar multiples. Letting s stand for *any* real number, and letting $u = (u_1, u_2, u_3)$ we want to demonstrate that $P(su)$ is the same as $sP(u)$. This is also true; deleting the third coordinate and doing the scalar multiplication give the same result regardless of which order they're carried out. The result of su is (su_1, su_2, su_3) so $P(su) = (su_1, su_2)$. The result of $P(u)$ is (u_1, u_2) , so $sP(u) = (su_1, su_2)$. This checks the second box, and confirms that P satisfies the definition of linearity.

These kinds of proofs are usually easier to do than to follow, so I've given you another one as an exercise. In the exercise, you can check that a function from 2D to 3D specified by a given matrix is linear using the same approach.

More illustrative than an algebraic proof is an example. What does it look like when we project a 3D vector sum down to 2D? We can see it in three steps. First we can draw a vector sum of two vectors u and v in 3D.

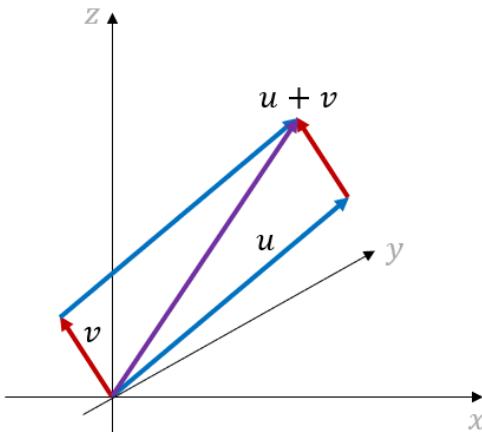


Figure 5.19 A vector sum of two arbitrary vectors u and v in 3D.

Then, we can trace a line from each of them to the x,y -plane to show where they end up after projection.

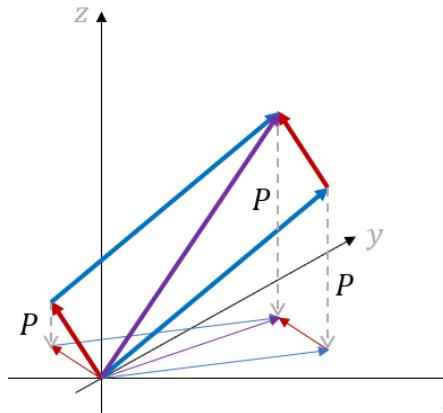


Figure 5.20 Visualizing where u , v , and $u+v$ end up after projection to the x,y -plane.

Finally, we can look at these new vectors and see that they *still* constitute a vector sum.

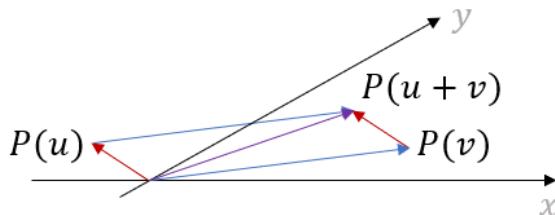


Figure 5.21 The projected vectors form a sum: $P(u) + P(v) = P(u+v)$.

In other words, if three vectors u , v , and w form a vector sum $u+v = w$, then their “shadows” in the x,y -plane also form a vector sum. Now that you’ve got some intuition for a linear transformation from 3D to 2D and a matrix that represents it, let’s return to our discussion of linear maps in general.

5.2.5 Composing linear maps

The beauty of matrices is that they store all of the data required to evaluate a linear function on a given vector. What’s more is that the dimensions of a matrix tell us the dimensions of input vectors and output vectors for that underlying function. We captured that visually by drawing “machines” for matrices of varying dimensions, whose input and output slots have different shapes. Here are four examples we’ve seen, labeled with letters so we can refer back to them.

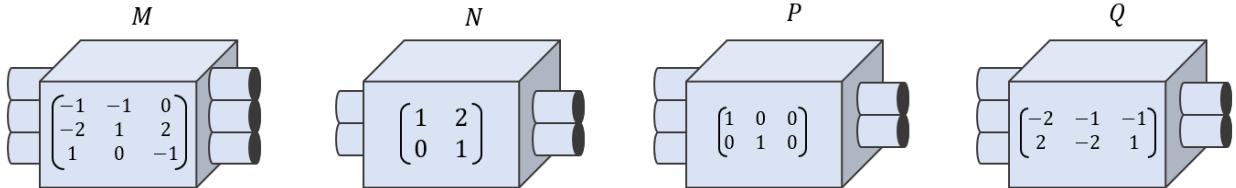


Figure 5.22 Four linear functions represented as machines with input and output slots. The shape of a slot tells us what dimension of vector it accepts or produces.

Drawn like this, it's easy to pick out which pairs of linear function machines could be "welded" together to build a new one. For instance, the output slot of M has the same shape as the input slot of P, so we could make the composition $P(M(v))$ -- the output of M is a three dimensional vector, which can be passed right along into the input slot of P.

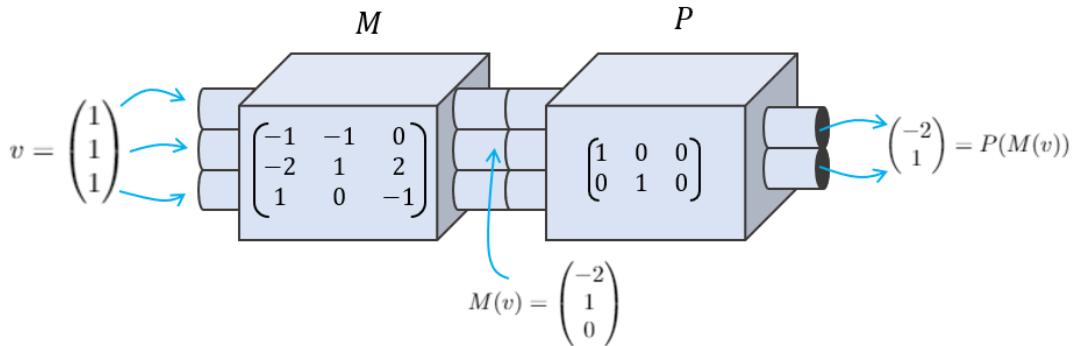


Figure 5.23 The composition of P and M. A vector is passed in to the input slot of M, the output $M(v)$ passes invisibly through the plumbing and into P, and finally the output $P(M(v))$ emerges from the other end.

By contrast, we can't compose N and M because N doesn't have enough output slots to fill every input of M.

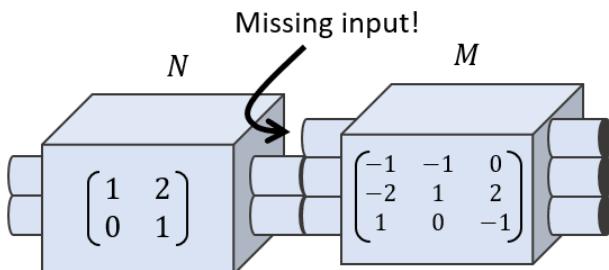


Figure 5.21 The composition of N and M is **not valid** since outputs of N are 2D vectors while inputs to M are 3D vectors.

I'm making this idea visual now by talking about "slots", but hidden underneath is the same reasoning we used to decide if two matrices could be multiplied together. The count of the columns of the first matrix has to match the count of rows of the second. When the dimensions match in this way, so do the slots, and we can compose the linear functions and multiply their matrices.

Thinking of P and M as matrices, the composition of P and M is written PM as a matrix product. (Remember, if PM acts on a vector v as "PMv", M is applied first and then P.) When $v = (1, 1, 1)$, the product PMv is a product of two matrices and a column vector, and it can be simplified into a single matrix times a column vector if we evaluate PM.

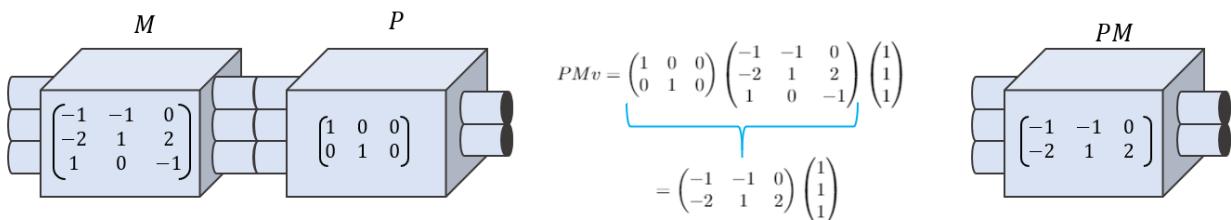


Figure 5.22 Applying M and then P is equivalent to applying the composition PM. We consolidate the composition into a single matrix by doing the matrix multiplication.

As a programmer, you're used to thinking of functions in terms of the types of data they consume and produce. I've given you a lot of notation and terminology to digest thus far in this chapter, but as long as you hang on to this core concept you'll get the hang of it eventually. I strongly encourage you to work the following exercises to make sure you've got the hang of the language of matrices. For the rest of this chapter and the next, there won't be many big new concepts, only applications of what we've seen so far. These applications will give you even more practice with matrix and vector computations.

5.2.6 Exercises

EXERCISE

What are the dimensions of this matrix?

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

- a.) 5x3
- b.) 3x5

SOLUTION

This is a 3x5 matrix, since it has three rows and five columns.

EXERCISE

What are the dimensions of a 2D column vector considered as a matrix? What about a 2D row vector? A 3D column vector? A 3D row vector?

SOLUTION

A 2D column vector has two rows and one column, so it is a 2x1 matrix. A 2D row vector has one row with two columns, so it is a 1x2 matrix. Likewise, 3D column and row vectors have dimensions 3x1 and 1x3 as matrices, respectively.

MINI PROJECT

Many of our vector and matrix operations make use of the Python `zip` function. When given input lists of different sizes, this function truncates the longer of the two rather than failing. This means that when we pass invalid inputs, we get meaningless results back. For instance, there is no such thing as a dot product between a 2D vector and a 3D vector, but our `dot` function returns something anyway:

```
>>> from vectors import dot
>>> dot((1,1),(1,1,1))
2
```

Add guards to all of the vector arithmetic functions so that they throw exceptions rather than returning values for vectors of invalid sizes. Once you've done that, show that `matrix_multiply` can no longer accept a product of a 3x2 and a 4x5 matrix.

EXERCISE

Which of the following are valid matrix products? For those that are valid, what dimension is the product matrix?

- A) $\begin{pmatrix} 10 & 0 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 8 & 2 & 3 & 6 \\ 7 & 8 & 9 & 4 \\ 5 & 7 & 0 & 9 \\ 3 & 3 & 0 & 2 \end{pmatrix}$
- B) $\begin{pmatrix} 0 & 2 & 1 & -2 \\ -2 & 1 & -2 & -1 \end{pmatrix} \begin{pmatrix} -3 & -5 \\ 1 & -4 \\ -4 & -4 \\ -2 & -4 \end{pmatrix}$
- C) $\begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} (3 \ 3 \ 5 \ 1 \ 3 \ 0 \ 5 \ 1)$
- D) $\begin{pmatrix} 9 & 2 & 3 \\ 0 & 6 & 8 \\ 7 & 7 & 9 \end{pmatrix} \begin{pmatrix} 7 & 8 & 9 \\ 10 & 7 & 8 \end{pmatrix}$

- A) This product of a 2×2 matrix and a 4×4 matrix is not valid; the first matrix has two columns but the second matrix has four rows.
- B) This product of a 2×4 matrix and a 4×2 matrix *is* valid; the four columns of the first matrix match the four rows of the second matrix. The result is a 2×2 matrix.
- C) This product of a 3×1 matrix and a 1×8 matrix *is* valid; the single column of the first matrix matches the single row of the second. The result is a 3×8 matrix.
- D) This product of a 3×3 matrix and a 2×3 matrix is not valid; the three columns of the first matrix do not match the two rows of the second.

EXERCISE

A matrix with 15 total entries is multiplied by a matrix with 6 total entries. What are the dimensions of the two matrices and what is the dimension of the product matrix?

SOLUTION

Let's call the dimensions of the matrices $m \times n$ and $n \times k$, since the number of columns of the first matrix has to match the number of rows of the second. Then $mn = 15$ and $nk = 6$. There are actually two possibilities.

The first is that $m = 5$, $n = 3$, and $k = 2$. Then this would be a 5×3 matrix multiplied by a 3×2 matrix resulting in a 5×2 matrix.

The second possibility is that $m = 15$, $n = 1$, and $k = 6$. Then this would be a 15×1 matrix times a 1×6 matrix, resulting in a 15×6 matrix.

EXERCISE

Write a function that turns a column vector into a row vector, or vice versa. Flipping a matrix on its side like this is called *transposition*.

SOLUTION:

```
def transpose(matrix):
    return tuple(zip(*matrix))
```

The call `zip(*matrix)` returns a list of columns of the matrix, and then we tuple them. This has the effect of swapping rows and columns of any input matrix, specifically turning column vectors into row vectors and vice versa.

```
>>> transpose(((1,), (2,), (3,)))
((1, 2, 3),
>>> transpose(((1, 2, 3),))
((1,), (2,), (3,))
```

EXERCISE

Draw a picture that shows that a 10×8 and a 5×8 matrix can't be multiplied in that order.

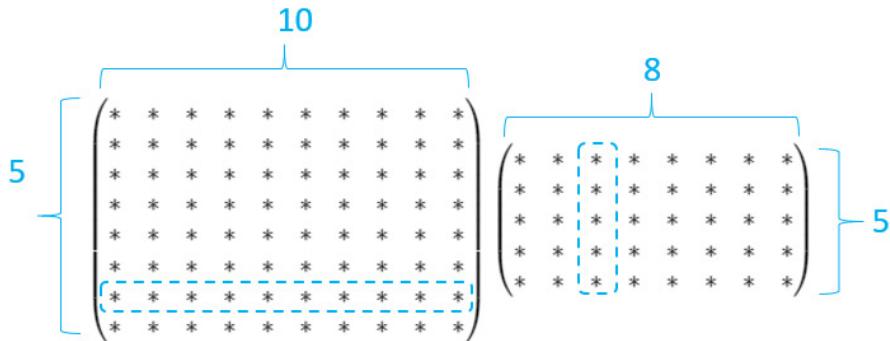
SOLUTION:

Figure 5.23 The rows of the first matrix have ten entries but the columns of the second have five, meaning we can't evaluate this matrix product.

EXERCISE

Want to multiply three matrices together: A is 5×7 , B is 2×3 and C is 3×5 . What order can they be multiplied in and what is the size of the result?

SOLUTION

One valid product is BC, a 2×3 times a 3×5 matrix yielding a 2×5 matrix. Another is CA, a 3×5 matrix times a 5×7 matrix yielding a 3×7 matrix. The product of three matrices BCA is valid, regardless of the order you do it: (BC)A is a 2×5 matrix times a 5×7 matrix, while B(CA) is a 2×3 matrix times a 3×7 matrix. Each yields the same 2×7 matrix as a result.

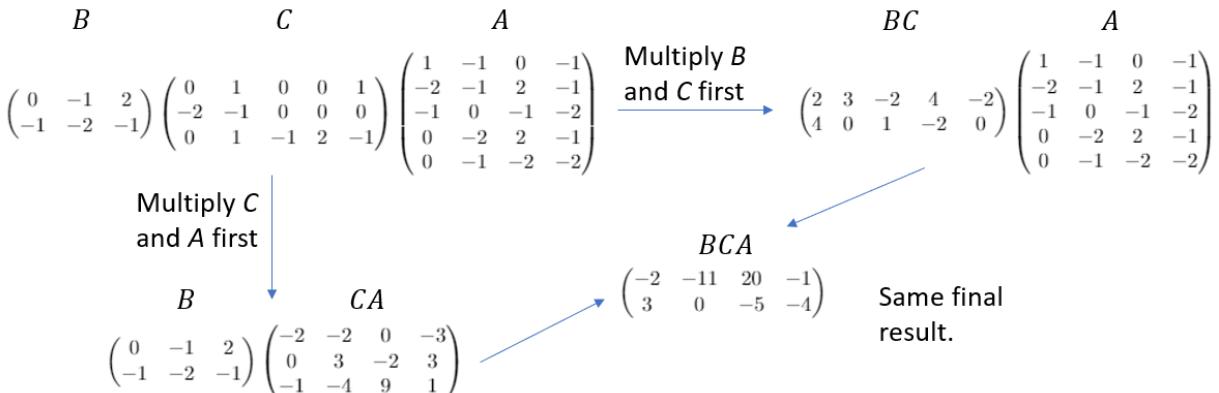


Figure 5.24 Multiplying three matrices in different orders.

EXERCISE

Projection onto the y,z -plane and onto the x,z plane are also linear maps from 3D to 2D. What are their matrices?

SOLUTION

Projection onto the y,z -plane deletes the x -coordinate. The matrix for this operation is:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Likewise, projection onto the x,z -plane deletes the y -coordinate:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

For example:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ z \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} y \\ z \end{pmatrix}.$$

EXERCISE

Show by example that the `infer_matrix` function from a previous exercise can create matrices for linear functions whose inputs and outputs have different dimensions.

SOLUTION

One function we could test would be projection onto the x,y-plane, which takes in 3D vectors and returns 2D vectors. We can implement this linear transformation as a python function and then infer its 2×3 matrix:

```
>>> def project_xy(v):
...     x,y,z = v
...     return (x,y)
...
>>> infer_matrix(3,project_xy)
((1, 0, 0), (0, 1, 0))
```

Note that we had to supply the dimension of *input* vectors as an argument, so that we could build the correct standard basis vectors to test under the action of `project_xy`. Once `project_xy` is passed the 3D standard basis vectors, it automatically outputs 2D vectors to supply the columns of the matrix.

EXERCISE

Write a 4×5 matrix which acts on a 5-dimensional vector by deleting the third of its five entries, thereby producing a 4-dimensional vector. For instance, multiplying it with the column vector form of $(1,2,3,4,5)$ should return $(1,2,4,5)$.

SOLUTION

The matrix is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

You can see that the first, second, fourth, and fifth coordinates of an input vector form the four coordinates of the output vector.

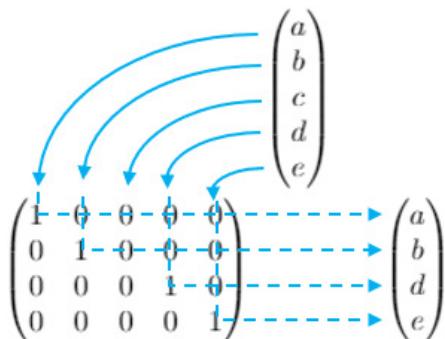


Figure 5.25 The ones in the matrix indicate where coordinates of the input vector will end up in the output vector.

MINI PROJECT

Consider the vector of six variables (l, e, m, o, n, s) . Find the matrix for the linear transformation that acts on this vector to produce the vector (s, o, l, e, m, n) as a result.

(Hint: the third coordinate of the output equals the first coordinate of the input, so the transformation must send the standard basis vector $(1, 0, 0, 0, 0, 0)$ to $(0, 0, 1, 0, 0, 0)$.)

SOLUTION:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} l \\ e \\ m \\ o \\ n \\ s \end{pmatrix} = \begin{pmatrix} 0 + 0 + 0 + 0 + 0 + s \\ 0 + o + 0 + 0 + 0 + 0 \\ l + 0 + 0 + 0 + 0 + 0 \\ 0 + e + 0 + 0 + 0 + 0 \\ 0 + 0 + m + 0 + 0 + 0 \\ 0 + 0 + 0 + 0 + n + 0 \end{pmatrix} = \begin{pmatrix} s \\ o \\ l \\ e \\ m \\ n \end{pmatrix}$$

Figure 5.26 A matrix which re-orders the entries of a 6D vector in a specified way.

EXERCISE

What valid products can be made from the matrices M, N, P, and Q from the last subsection? Include in your consideration the products of matrices with themselves. For those products that are valid, what are the dimensions of the matrix products?

SOLUTION

M is 3x3, N is 2x2, and P and Q are both 2x3. The product of M with itself, $MM = M^2$ is valid and a 3x3 matrix, so is $NN = N^2$ which is a 2x2 matrix. Apart from that, PM , QM , NP , and NQ are all 3x2 matrices.

5.3 Translating vectors with matrices

One advantage of matrices is that computations look the same in any number of dimensions. We don't need to worry about picturing the configurations of vectors in 2D or 3D; we can simply plug them into the formulas for matrix multiplication or use them as inputs to our Python `matrix_multiply`. This is especially useful when we want to do computations in more than three dimensions. The human brain isn't wired to picture vectors in four or five dimensions, let alone 100, but we have already seen we can do computations with vectors in higher dimensions. In this section, we'll cover a computation that *requires* doing computation in higher dimensions: translating vectors using a matrix.

5.3.1 Making plane translations linear

In the last chapter, we showed that translations are *not* linear transformations. When we move every point in the plane by a given vector, the origin moves and vector sums are not preserved. So how can we hope to execute a 2D transformation with a matrix if it is not a linear transformation?

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/math-for-programmers>

The trick is that we'll think of our 2D points to translate as living in 3D. Let's return to our dinosaur figure from chapter 2. The dinosaur was composed of 21 points, and we could connect them in order to create the outline of the figure.

```
dino_vectors = [(6,4), (3,1), (1,2), (-1,5), (-2,5), (-3,4), (-4,4),
    (-5,3), (-5,2), (-2,2), (-5,1), (-4,0), (-2,1), (-1,0), (0,-3),
    (-1,-4), (1,-4), (2,-3), (1,-2), (3,-1), (5,1)
]

p = Plane((-7,-7),(7,7),"dino-plane")
p.draw_vectors(*dino_vectors)
count = len(dino_vectors)
for i in range(0,count):
    p.draw_segment(dino_vectors[i],dino_vectors[(i+1)%count], color='blue')
p.show()
```

The result is the familiar 2D dinosaur:

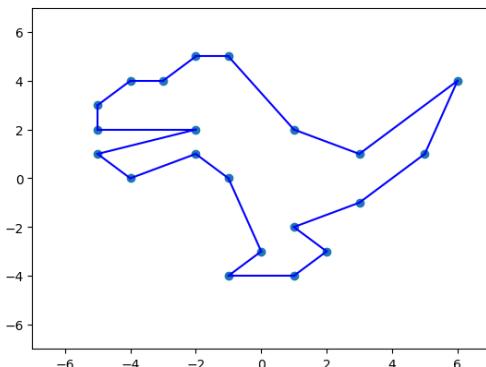


Figure 5.27 The familiar, 2D dinosaur from chapter 2.

If we wanted to translate the dinosaur to the right by 3 units and up by 1 unit, we could simply add the vector $(3,1)$ to each of the dinosaur's vertices. But this isn't a linear map, so we can't produce a 2×2 matrix that will do this translation. If we think of the dinosaur as an inhabitant of 3D space instead of the 2D plane, it turns out we *can* formulate the translation as a matrix.

Bear with me for a moment while I show you the trick, and I'll explain how it works shortly. Let's give every point of the dinosaur a z -coordinate of 1. Then we can plot it in 3D and see that it lies on the plane where $z = 1$.

```
dino_vectors_3d = [(x,y,1) for x,y in dino_vectors]

s = Space((-7,-7,-1), (7,7,3), "dino-space")
s.draw_vectors(*dino_vectors_3d)
s.draw_axes()
for i in range(0,count):
    s.draw_segment(
        dino_vectors_3d[i],
```

```
dino_vectors_3d[(i+1)%count], color='blue')
s.show()
```

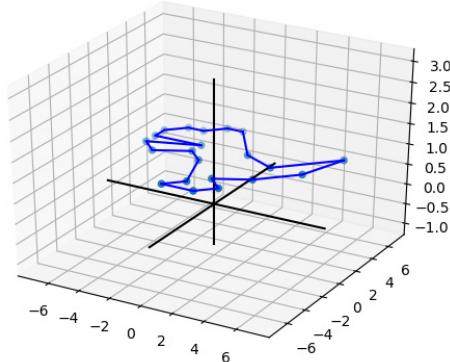


Figure 5.28 The same dinosaur with each of its vertices given a z-coordinate of 1.

Now, here's a matrix that "skews" 3D space, so that the origin stays put, but the plane where $z = 1$ is translated as desired. Trust me for now! I've highlighted the numbers relating to the translation that you should pay attention to.

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 5.28 A magic matrix which moves the plane $z=1$ by +3 in the x-direction and +1 in the y-direction.

We can apply this matrix to each vertex of the dinosaur, and then voila! The dinosaur is translated by $(3,1)$ in its plane.

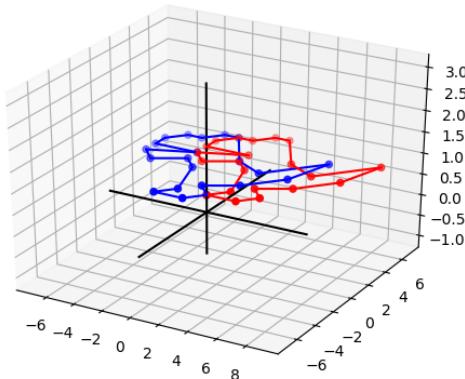


Figure 5.29 Applying the matrix to every point keeps the dinosaur in the same plane, but translates it within the plane by $(3,1)$.

```

magic_matrix = (
    (1,0,3),
    (0,1,1),
    (0,0,1))

translated = [multiply_matrix_vector(magic_matrix, v) for v in dino_vectors_3d]

```

For clarity, we could then delete the z-coordinates again and show the translated dinosaur in the plane with the original one.

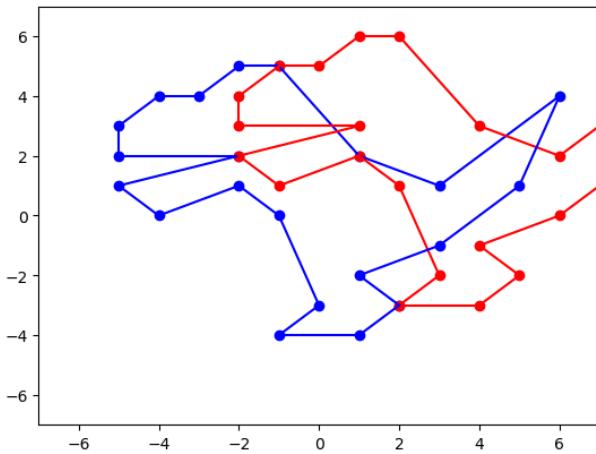


Figure 5.30 Dropping the translated dinosaur back into 2D.

You can reproduce the code and check the coordinates to see that the dinosaur was indeed translated by $(3,1)$ in the final picture. Now let me show you how the trick works.

5.3.2 Finding a 3D matrix for a 2D translation

The columns of our “magic” matrix, like the columns of any matrix, tell us where the standard basis vectors end up after being transformed. Calling this matrix T , the vectors e_1 , e_2 , and e_3 would be transformed into the vectors $Te_1 = (1,0,0)$, $Te_2 = (0,1,0)$, and $Te_3 = (3,1,1)$. This means e_1 and e_2 are unaffected, and e_3 changes its x and y-components only.

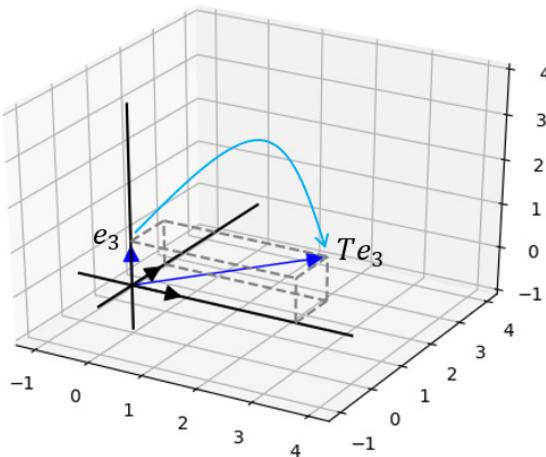


Figure 5.31 This matrix doesn't move e_1 or e_2 , but it does move e_3 .

Any point in 3D, and therefore any point on the dinosaur, is built as a linear combination of e_1 , e_2 , and e_3 . For instance, the tip of the dinosaur's tail is at $(6,4,1)$ which is $6e_1 + 4e_2 + e_3$. Since T doesn't move e_1 or e_2 only the effect on e_3 will move the point. $T(e_3) = e_3 + (3,1,0)$ so the point is translated by $+3$ in the x -direction and $+1$ in the y -direction.

You can also see this algebraically. Any vector $(x,y,1)$ will be translated by $(3,1,0)$ by this matrix:

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot x + 0 \cdot y + 3 \cdot 1 \\ 0 \cdot x + 1 \cdot y + 1 \cdot 1 \\ 0 \cdot x + 0 \cdot y + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} x+3 \\ y+1 \\ 1 \end{pmatrix}$$

Figure 5.32 Showing algebraically that the matrix translates a vector by $(3,1)$.

If you want to translate a collection of 2D vectors by some vector (a,b) , the general recipe is:

1. Move the 2D vectors into the plane in 3D space where $z = 1$. Namely, give them each a z -coordinate of 1.
2. Multiply them by the matrix:

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}$$

with your given choices of a and b plugged in.

3. Delete the z -coordinate of all of the vectors so you are left with 2D vectors as a result.

Now that we can do translations with matrices, we can creatively combine them with other linear transformations.

5.3.3 Combining translation with other linear transformations

In the example matrix above, the first two columns are exactly e_1 and e_2 , meaning that only the change in e_3 will move a figure. We don't want $T(e_1)$ or $T(e_2)$ to have any z -component, because that would move the figure out of the plane $z = 1$. But we could modify or interchange the other components. Here's the idea:

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 5.33 Let's see what happens when we move $T(e_1)$ and $T(e_2)$ in the x,y plane.

It turns out you can put any 2×2 matrix in the top left here, doing the corresponding linear transformation *in addition* to the translation specified in the third column. For instance, the matrix

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

produces a 90 degree counterclockwise rotation. So inserting it in the translation matrix, we'd get a new matrix which would rotate the x,y -plane by 90 degrees and *then* translate by $(3,1)$.

$$\begin{pmatrix} 0 & -1 & 3 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 5.34 A matrix which rotates e_1 and e_2 by 90 degrees **and** translates e_3 by $(3,1)$. Any figure in the plane where $z = 1$ will feel both transformations.

To prove it, we can carry out this transformation on all of the 3D dinosaur vertices in Python.

```
rotate_and_translate = ((0, -1, 3), (1, 0, 1), (0, 0, 1))
rotated_translated_dino = [
    multiply_matrix_vector(rotate_and_translate, v)
    for v in dino_vectors_3d]
```

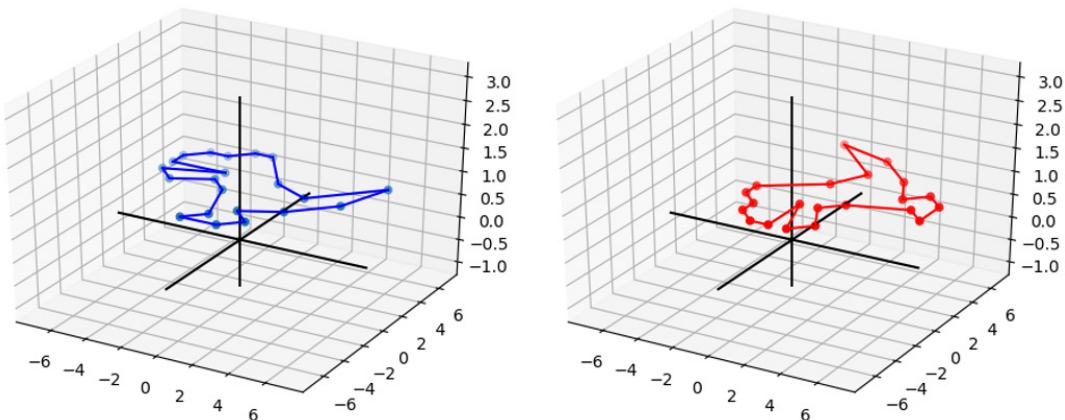


Figure 5.35 The dinosaur is rotated and translated by a single matrix.

Once you get the hang of doing 2D translations with a matrix, we can apply the same approach to do a 3D translation. To do that, we'll have to use a 4×4 matrix, and enter the mysterious fourth dimension.

5.3.4 Translating 3D objects in a 4D world

What is the fourth dimension? A 4D vector would be an arrow with some length, width, depth, and one other dimension. When we built 3D space from 2D space, we added a z-coordinate. That meant that objects could live in the x,y -plane, where $z = 0$, or they could live in any other parallel plane where z took a different value. Here are some of the parallel planes:

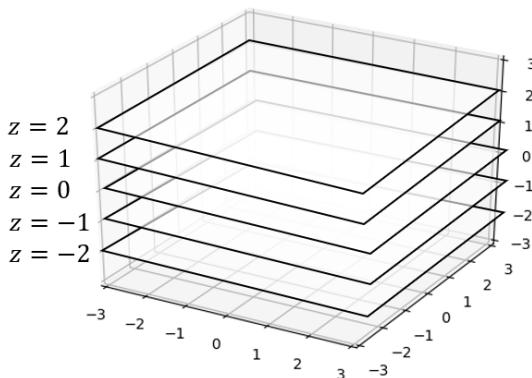


Figure 5.36 Building 3D space out of a stack of parallel planes, each looking like the x,y plane but at a different z -coordinates.

We can think of four dimensions in analogy to this model, as a collection of 3D spaces that are indexed by some fourth coordinate. One way to interpret the fourth coordinate is “time”. Each snapshot at a given time is a 3D space, but the collection of all of the snapshots is four-dimensional and called a *spacetime*. The origin of the spacetime is the origin of the space at the moment when the time, t , is equal to 0.

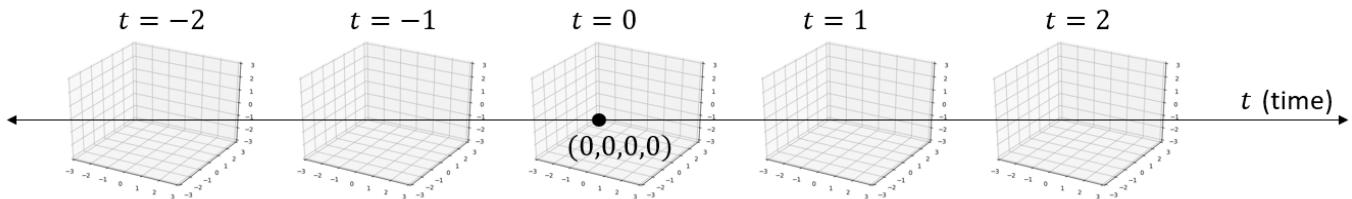


Figure 5.37 An illustration of 4D spacetime. While a slice of 3D space at a given z value is a 2D plane, a slice of 4D spacetime at a given t value is a 3D space.

This is the starting point for Einstein’s theory of special relativity. (In fact, you are now qualified to go read about this theory because it is all based on 4D spacetime and linear transformations given by 4×4 matrices.)

Vector math is indispensable in higher dimensions, because we run out of good analogies quickly. For five, six, seven, or more dimensions, I have a hard time picturing them but the coordinate math is no harder than in two or three dimensions. For our current purposes, it’s sufficient to think of a four-dimensional vector as a four-tuple of numbers.

Let’s replicate the trick that worked for translating 2D vectors in 3D. If we start with a 3D vector like (x,y,z) and we want to translate it by a vector (a,b,c) , we can attach a fourth coordinate of 1 to the target vector and use an analogous 4D matrix to do the translation. Doing the matrix multiplication confirms that we’ll get the result we want:

$$\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \\ z+c \\ 1 \end{pmatrix}$$

Figure 5.38 Giving the vector (x,y,z) a fourth coordinate of 1, we can translate it by (a,b,c) using this matrix.

This matrix increases the x coordinate by a , the y -coordinate by b , and the z -coordinate by c , so it does the transformation required to translate by the vector (a,b,c) . In Python, we can package in a function the work of adding a fourth coordinate, applying this 4×4 matrix, and then deleting the fourth coordinate.

```
def translate_3d(translation):
    def new_function(target):
        a,b,c = translation
        x,y,z = target
```

1

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/math-for-programmers>

```

matrix = ((1,0,0,a),(0,1,0,b),(0,0,1,c),(0,0,0,1))           2
vector = (x,y,z,1)
x_out, y_out, z_out, _ = multiply_matrix_vector(matrix, vector) 3
return (x_out,y_out,z_out)
return new_function

```

- ➊ Our translate_3d function will take a translation vector and return a new function that applies that translation to a 3D vector.
- ➋ Build the 4x4 matrix to do the translation, and on the next line turn (x,y,z) into a 4D vector with fourth coordinate 1.
- ➌ Do the 4D matrix transformation.

Finally, drawing the teapot as well as the teapot translated by (2,2,-3), we can see that the teapot is moved appropriately.

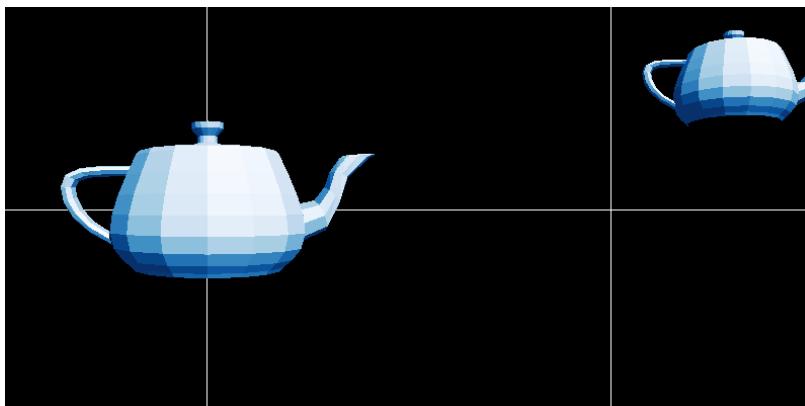


Figure 5.39 The un-translated teapot (left) and translated teapot (right). As expected, it moves up and to the right, and away from our viewpoint.

With translation packaged as a matrix operation, we could now combine it with other 3D linear transformations and do them in one step. It turns out you *can* interpret the artificial fourth-coordinate in this set-up as time. The two images above could be snapshots of a teapot at $t = 0$ and $t = 1$ which is moving in the direction $(2,2,-3)$ at constant speed. If you're looking for a fun challenge, you can replace the vector $(x,y,z,1)$ in this implementation with vectors of the form (x,y,z,t) , where the coordinate t changes over time. With $t = 0$ and $t = 1$, the teapot should match the frames above, and between the two it should move smoothly between the two positions. If you can figure out how this works, you'll be catching up with Einstein!

So far we've been focused exclusively on vectors as points in space that we can render to a computer screen. This is clearly an important use-case, but it only scratches the surface of what we can do with vectors and matrices. The study of how vectors and linear transformations work together in general is called *linear algebra*, and I'll give you a broader picture of this subject in the next chapter, along with some fresh examples which are relevant to programmers.

5.3.5 Exercises

EXERCISE

Show that the matrix doesn't work if you move a 2D figure to the plane $z = 2$. What happens instead?

SOLUTION

Using instead `[(x,y,2) for x,y in dino_vectors]` and applying the same 3×3 matrix, the dinosaur is translated twice as far, by the vector $(6,2)$ instead of $(3,1)$. This is because the vector $(0,0,1)$ is translated by $(3,1)$ and the transformation is linear.

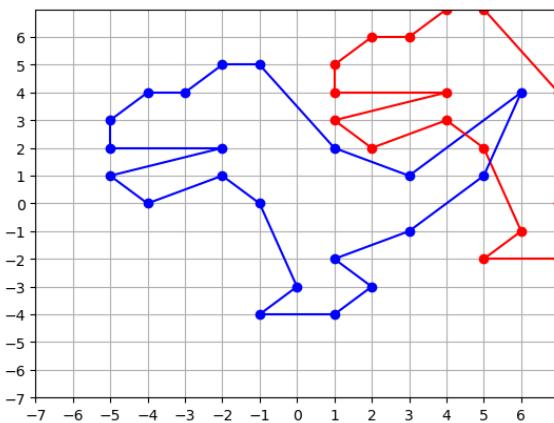


Figure 5.40 A dinosaur in the plane where $z = 2$ is translated twice as far by the same matrix.

EXERCISE

Come up with a matrix to translate the dinosaur by -2 units in the x-direction and -2 units in the y-direction. Execute the transformation and show the result.

SOLUTION

Replacing the values 3 and 1 in the original matrix with -2 and -2, we get:

$$\begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}$$

and the dinosaur indeed translates down and to the left by the vector $(-2,-2)$.

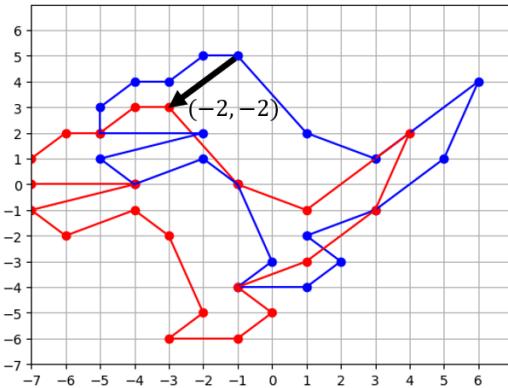


Figure 5.41. The dinosaur is translated by $(-2, -2)$.

EXERCISE

Show that any matrix of the form

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix}$$

doesn't affect the z-coordinate of a 3D column vector it is multiplied by.

SOLUTION

If the initial z-coordinate of a 3D vector is a number z , this matrix leaves that coordinate unchanged:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ 0x + 0y + z \end{pmatrix}$$

MINI PROJECT

Give a 3×3 matrix that rotates a 2D figure in the plane $z = 1$ by 45 degrees, decreases its size by a factor of 2, and translates it by the vector $(2, 2)$. Demonstrate that it works by applying it to the vertices of the dinosaur.

SOLUTION

First let's find a 2×2 matrix for rotating a 2D vector by 45 degrees:

```
>>> from vectors import rotate2d
>>> from transforms import *
>>> from math import pi
```

```
>>> rotate_45_degrees = curry2(rotate2d)(pi/4) ①
>>> rotation_matrix = infer_matrix(2,rotate_45_degrees)
>>> rotation_matrix
((0.7071067811865476, -0.7071067811865475), (0.7071067811865475,
0.7071067811865476))
```

- ① Build a function that executes `rotate2d` with an angle of 45 degrees (or $\pi/4$ radians) for an input 2D vector.

This matrix is approximately:

$$\begin{pmatrix} 0.707 & -0.707 \\ 0.707 & 0.707 \end{pmatrix}$$

Similarly, we can find a matrix to scale by a factor of $\frac{1}{2}$.

$$\begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

Multiplying these matrices together, we accomplish both at once:

```
>>> from matrices import *
>>> scale_matrix = ((0.5,0),(0,0.5))
>>> rotate_and_scale = matrix_multiply(scale_matrix,rotation_matrix)
>>> rotate_and_scale
((0.3535533905932738, -0.35355339059327373), (0.35355339059327373,
0.3535533905932738))
```

A 3x3 matrix that would translate the dinosaur by (2,2) in the plane where z=1 would be:

$$\begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

We can plug our 2x2 rotation and scaling matrix into the top left of this matrix, giving us the final matrix we want:

```
>>> ((a,b),(c,d)) = rotate_and_scale
>>> final_matrix = ((a,b,2),(c,d,2),(0,0,1))
>>> final_matrix
((0.3535533905932738, -0.35355339059327373, 2), (0.35355339059327373,
0.3535533905932738, 2), (0, 0, 1))
```

Moving the dinosaur to the plane z=1, applying this matrix in 3D, and then projecting back to 2D gives us the rotated, scaled, and translated dinosaur, using only one matrix multiplication:

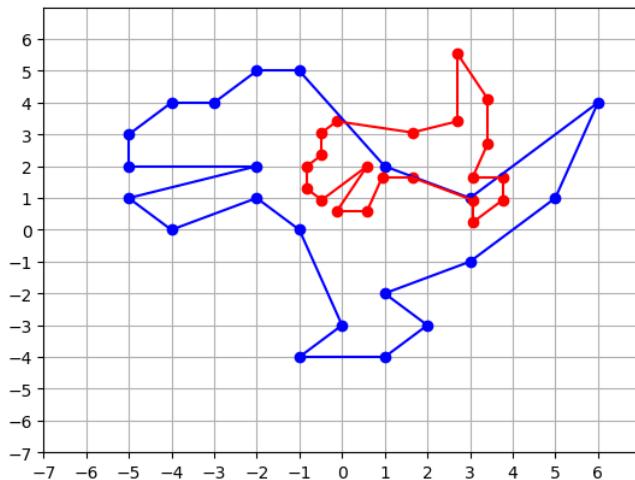


Figure 5.42 One matrix was needed to rotate, scale, and translate the dinosaur.

EXERCISE

The matrix in the preceding section rotates the dinosaur by 90 degrees and *then* translates it by (3,1). Using matrix multiplication, build a matrix that does this in the opposite order.

SOLUTION

If the dinosaur is in the plane where $z = 1$, then the following matrix does a rotation by 90 degrees with *no* translation:

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We want to translate first and *then* rotate, so we multiply this rotation matrix by the translation matrix:

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 0 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

This is different from the other matrix, which rotates before the translation. In this case, we see the translation vector (3,1) is affected by the 90 degree rotation; the new effective translation is (-1,3).

EXERCISE

Write a function analogous to `translate_3d` called `translate_4d`, which uses a 5x5 matrix to translate a 4D vector by another 4D vector. Run an example to show that the coordinates are translated.

SOLUTION

The set-up is the same, except that we lift the 4D vector to 5D by giving it a fifth coordinate of 1.

```
def translate_4d(translation):
    def new_function(target):
        a,b,c,d = translation
        x,y,z,w = target
        matrix = (
            (1,0,0,0,a),
            (0,1,0,0,b),
            (0,0,1,0,c),
            (0,0,0,1,d),
            (0,0,0,0,1))
        vector = (x,y,z,w,1)
        x_out,y_out,z_out,w_out,_ = multiply_matrix_vector(matrix, vector)
        return (x_out,y_out,z_out,w_out)
    return new_function
```

We can see that the translation works (the effect is the same as adding the two vectors).

```
>>> translate_4d((1,2,3,4))((10,20,30,40))
(11, 22, 33, 44)
```

5.4 Summary

In this chapter, you learned that

- A linear transformation is defined by what it does to standard basis vectors. When you apply a linear transformation to the standard basis, the resulting vectors contain all the data required to do the transformation.
- This means that only nine numbers are required to specify a 3D linear transformation of any kind (the three coordinates of each of these three resulting vectors). For a 2D linear transformation, four numbers are required.
- In *matrix* notation, we represent a linear transformation by putting these numbers in a rectangular grid. By convention, you build a matrix by applying the transformation to the standard basis and putting the resulting coordinate vectors side-by-side as columns.
- Using a matrix to evaluate the result of the linear transformation it represents on a given vector is called *multiplying* the matrix by the vector. When you do this multiplication, the vector is typically written as a column of its coordinates, from top to bottom, rather than a tuple.
- Two square matrices can also be “multiplied” together. The resulting matrix represents the composition of the linear transformations of the original two matrices.
- To calculate the product of two matrices, you take the dot products of rows of the first with columns of the second. For instance, the dot product of row *i* of the first matrix and column *j* of the second matrix gives you the value in row *i* and column *j* of the product.

- As square matrices represent linear transformations, non-square matrices represent *linear functions* from vectors of one dimension to vectors of another dimension. That is, these functions send vector sums to vector sums and scalar multiples to scalar multiples.
- The *dimension* of a matrix tells you what kind of vectors its corresponding linear function accepts and returns. A matrix with m rows and n columns is called an " m by n " (written $m \times n$) matrix. It defines a linear function from n -dimensional space to m -dimensional space.
- Translation is *not* a linear function, but it can be made linear if you perform it in a higher dimension. This observation allows us to do translations (simultaneously with other linear transformations) by matrix multiplication.

In the previous chapters, we used visual examples in 2D and 3D to motivate vector and matrix arithmetic. As we've gone along, we've put more emphasis on computation: at the end of the chapter we were calculating vector transformations in higher dimensions that we don't have any intuition for. This is one of the benefits of linear algebra: it gives you the tools to solve geometric problems that are too complicated to picture. We'll survey the broad range of applications in the next chapter.

6

Generalizing to Higher Dimensions

This chapter covers

- Implementing a Python abstract base class representing general vectors
- Defining *vector spaces* and listing their useful properties
- Interpreting functions, matrices, images, and sound waves as vectors
- Finding useful *subspaces* of vector spaces, containing data of interest

Even if you're not interested in animating teapots, the machinery of vectors, linear transformations, and matrices can be useful. In fact, these concepts are so useful there's an entire branch of math devoted to them: *linear algebra*. Linear algebra generalizes everything we know about 2D and 3D geometry to study different kinds of data having any number of dimensions.

As a programmer, you've already got experience with generalization. When writing a piece of software, it's common to find yourself writing similar code over and over. At some point you catch yourself doing this, and you consolidate the code into one class or function capable of handling all of the cases you've seen. This saves you typing and often improves code organization. Mathematicians follow the same process: after encountering similar patterns over and over, they can better state exactly what their seeing and refine their definitions.

In this chapter, we'll use this kind of thinking to define *vector spaces*. Vector spaces are collections of objects we can treat like vectors. These can be arrows in the plane, tuples of numbers, or objects completely different from the ones we've seen so far. For instance, you can treat images as vectors and take a linear combination of them:



Figure 6.1 A linear combination of two pictures produces a new picture.

The key operations in a vector space are vector addition and scalar multiplication. With these you can do linear combinations (including negation, subtraction, weighted averages, and so on) and you can reason about which transformations are linear. It turns out these operations help us make sense of the word “dimension.” For instance, we’ll see that the images I used above are 270,000-dimensional objects! We’ll cover high-dimensional and even infinite-dimensional spaces soon enough, but let’s start by reviewing the 2D and 3D spaces we already know.

6.1 Generalizing our definition of vectors

Python supports object-oriented programming, which is a great framework for generalization. Specifically, Python classes support inheritance: you can create new classes of objects that “inherit” properties and behaviors of an existing parent class. In our case, we want to realize the 2D vectors and 3D vectors we’ve already seen as instances of a more general class of objects simply called “vectors.” Any other objects that inherit behaviors from the parent class can rightly be called vectors as well.

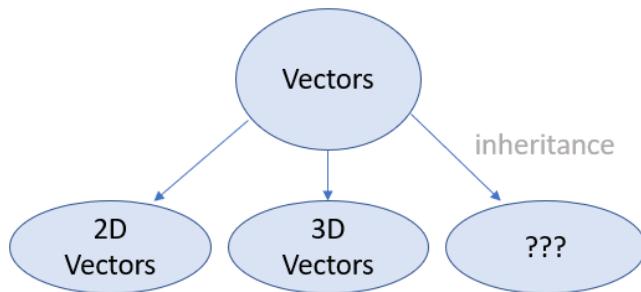


Figure 6.2 Treating 2D vectors, 3D vectors, and potentially other objects as special cases of vectors, using inheritance.

If you haven’t done object-oriented programming or you haven’t seen it done in Python, don’t worry. I stick to simple use-cases in this chapter, and will help you pick it up as we go. In case you want to learn more about classes and inheritance in Python before getting started, I’ve covered them in the appendix.

6.1.1 Creating a class for 2D coordinate vectors

In code, our 2D and 3D vectors have been *coordinate* vectors, meaning that they've been defined as tuples of numbers which are their coordinates. (We've also seen that vector arithmetic can be defined geometrically, but that's not the best way to program it.) For 2D coordinate vectors, the data is the ordered pair of the x-coordinate and y-coordinate. A tuple was a great way to store this data, but we can equivalently use a class. We'll call the class representing 2D coordinate vectors `Vec2`.

```
class Vec2():
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

We can initialize a vector like `v = Vec2(1.6, 3.8)` and retrieve its coordinates as `v.x` and `v.y`. Next, we can give this class the methods required to do 2D vector arithmetic, specifically addition and scalar multiplication. The addition function takes a second vector as an argument, and returns a new `Vec2` object whose coordinates are the sum of the x-coordinates and y-coordinates respectively.

```
class Vec2():
    ...
    def add(self, v2):
        return Vec2(self.x + v2.x, self.y + v2.y)

Doing vector addition with Vec2 could look like this:

v = Vec2(3,4)      # 1
w = v.add(Vec2(-2,6)) # 2
print(w.x)          # 3
```

- 1 Create a new `Vec2` called `v` with x-coordinate 3 and y-coordinate 4.
- 2 Add a second `Vec2` to `v` to produce a new `Vec2` instance called `w`, the result is $(3,4) + (-2,6) = (1,10)$.
- 3 Print the x-coordinate of `w`. The result is 1.

Like our original implementation of vector addition, the addition is not performed in-place. That is, the two input vectors are not modified; a new `Vec2` object is created to store the sum. We can implement scalar multiplication in a similar way, taking a scalar as an input and returning a new, scaled vector as an output.

```
class Vec2():
    ...
    def scale(self, scalar):           return Vec2(scalar * self.x, scalar * self.y)
```

So `Vec(1,1).scale(50)` returns a new vector with x-coordinate and y-coordinate both equal to 50. There's one more critical detail we need to take care of: currently the output of a comparison like `Vec2(3,4) == Vec2(3,4)` is `False`. This is problematic, since these instances represent the same vector. By default, Python compares instances by their references (asking whether they are located in the same place in memory) rather than by their values. We can

fix this by overriding the equality method, which causes Python to treat the “==” operator different for objects of the `Vec2` class. If you haven’t seen this before, I explain in some more depth in the appendix.

```
class Vec2():
    ...
    def __eq__(self,other):
        return self.x == other.x and self.y == other.y
```

We want two 2D coordinate vectors to be equal if their x-coordinates and y-coordinates agree, and this new definition of equality captures that. With this implemented, you’ll find that `Vec2(3, 4) == Vec2(3, 4)`.

This `Vec2` class now has the fundamental vector operations of addition and scalar multiplication, as well as an equality test that makes sense, so we can turn our attention to some cosmetic details.

6.1.2 Improving the `Vec2` class

As we changed the behavior of the “==” operator, we can customize the operators “+” and “*” to mean vector addition and scalar multiplication. This is called *operator overloading* and it is covered in the appendix.

```
class Vec2():
    ...
    def __add__(self, v2):
        return self.add(v2)
    def __mul__(self, scalar):
        return self.scale(scalar)
    def __rmul__(self,scalar): ①
        return self.scale(scalar)
```

- ① The `__mul__` and `__rmul__` methods define both orders of multiplication, so we can multiply vectors by scalars on the left or the right. Mathematically, we consider both orders to mean the same thing.

We can now write a linear combination concisely. For instance, `3.0 * Vec2(1, 0) + 4.0 * Vec2(0, 1)` gives us a new `Vec2` object with x-coordinate 3.0 and y-coordinate 4.0. It’s hard to read this in an interactive session though, because Python doesn’t print `Vec2` nicely:

```
>>> 3.0 * Vec2(1,0) + 4.0 * Vec2(0,1)
<__main__.Vec2 at 0x1cef56d6390>
```

Python gives us the memory address of the resulting `Vec2` instance, but we already observed that’s not what’s important to us. Fortunately we can change the string representation of `Vec2` objects by overriding the `__repr__` method.

```
class Vec2():
    ...
    def __repr__(self):
        return "Vec2({},{})".format(self.x,self.y)
```

This string representation shows the coordinates, which are the most important data for a `Vec2`. The results of `Vec2` arithmetic are much clearer now:

```
>>> 3.0 * Vec2(1,0) + 4.0 * Vec2(0,1)
Vec2(3.0,4.0)
```

We're doing the same math here as we did with our original tuple vectors, but in my opinion this is a lot nicer. Building a class required some boilerplate, like the custom equality we wanted, but it also enabled operator overloading for vector arithmetic. The custom string representation also makes it clear that we're not just working with *any* tuples, but rather 2D vectors that we intend to use in a certain way. Now, we have room to implement 3D vectors, represented by their own special class.

6.1.3 Repeating the process with 3D vectors

I'll call the 3D vector class `Vec3`, and it will look a lot like the 2D `Vec2` class except that its defining data will be three coordinates instead of two. In each method that explicitly references the coordinates, we need to make sure to properly use the `x`, `y`, and `z` values for `Vec3`.

Listing 6.1 A `Vec3` class which parallels the `Vec2` class.

```
class Vec3():
    def __init__(self,x,y,z): #1
        self.x = x
        self.y = y
        self.z = z
    def add(self, other):
        return Vec3(self.x + other.x, self.y + other.y, self.z + other.z)
    def scale(self, scalar):
        return Vec3(scalar * self.x, scalar * self.y, scalar * self.z)
    def __eq__(self,other):
        return self.x == other.x and self.y == other.y and self.z == other.z
    def __add__(self, other):
        return self.add(other)
    def __mul__(self, scalar):
        return self.scale(scalar)
    def __rmul__(self,scalar):
        return self.scale(scalar)
    def __repr__():
        return "Vec3({}, {}, {})".format(self.x, self.y, self.z)
```

We can now write 3D vector math in Python using the built-in arithmetic operators:

```
>>> 2.0 * (Vec3(1,0,0) + Vec3(0,1,0))
Vec3(2.0,2.0,0.0)
```

This `Vec3` class is much like the `Vec2` class, which puts us in a good place to think about generalization. There are a few different directions we could go, and like many software design choices the decision is subjective. We could, for example, focus on simplifying the arithmetic. Instead of implementing `add` differently for `Vec2` and `Vec3`, they could both use

the add function we built in chapter 3, which already handles coordinate vectors of any size. We could also generalize to store coordinates internally as a tuple or list, so the constructor could accept any number of coordinates and create a 2D, 3D or other coordinate vector. I'll leave these possibilities as exercises for you, and take us in a different direction.

The generalization I want to focus on is based on how we *use* the vectors, not how they work. This will get us to a mental model which both organizes the code well and aligns with the mathematical definition of a vector. We currently have the advantage that we can write generic functions that can be used on any kind of vector, like an average function:

```
def average(v1,v2):
    return 0.5 * v1 + 0.5 * v2
```

We can insert either 3D vectors or 2D vectors; for instance `average(Vec2(9.0, 1.0), Vec2(8.0,6.0))` and `average(Vec3(1,2,3), Vec3(4,5,6))` both give us correct and meaningful results. As a spoiler, we will be able to average pictures together as well. Once we've implemented a suitable class for images, we'll be able to write `average(img1, img2)` and get a new image back:

This is where we will see the beauty and the economy that come with generalization. We can write a single, generic function like `average`, and use it for a wide variety of types of inputs. The only constraint on the inputs is that they need to support multiplication by scalars and addition with one another. *How* to do arithmetic will vary between `Vec2` objects, `Vec3` objects, images, or other kinds of data, but there will always be an important overlap in *what* arithmetic we can do with them. When we separate the "what" from the "how," we open the door for code reuse and far-reaching mathematical statements.

How can we best describe *what* we can do with vectors, separately from the details of *how* we carry them out? We can capture this in Python using an abstract base class.

6.1.4 Building a Vector base class

The basic things we can do with a `Vec2` or `Vec3` are constructing a new instance, adding with other vectors, multiplying by a scalar, testing equality with another vector, and representing it as a string. Of these, only addition and scalar multiplication are distinctive vector operations. The rest are automatically included with any new Python class. This prompts a definition of a Vector base class:

```
from abc import ABCMeta, abstractmethod

class Vector(metaclass=ABCMeta):
    @abstractmethod
    def scale(self,scalar):
        pass
    @abstractmethod
    def add(self,other):
        pass
```

The `abc` module contains helper classes, functions, and method decorators that help define an *abstract base class*, a class which is not intended to be instantiated. Instead, it's designed to be used as a template for classes that inherit from it. The `@abstractmethod` decorator means that a method is not implemented and needs to be implemented by the child class. For instance, if you try to instantiate a vector with code like `v = Vector()`, you'll get the following `TypeError`:

```
TypeError: Can't instantiate abstract class Vector with abstract methods add, scale
```

This makes sense: there is no such thing as a vector that is "just a vector." It needs to have some concrete manifestation: for instance as a list of coordinates, an arrow in the plane, or something else. But, this is still a useful base class because it forces any child class to include requisite methods.

It is also useful to have this base class because we can equip it with all the methods that depend only on addition and scalar multiplication, like our operator overloads:

```
class Vector(metaclass=ABCMeta):
    ...
    def __mul__(self, scalar):
        return self.scale(scalar)
    def __rmul__(self, scalar):
        return self.scale(scalar)
    def __add__(self, other):
        return self.add(other)
```

In contrast to the abstract methods `scale` and `add`, these are implementations that will automatically be available to any child class. We can simplify `Vec2` and `Vec3` to inherit from `Vector`. Here's a new implementation for `Vec2`:

```
class Vec2(Vector):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def add(self, other):
        return Vec2(self.x + other.x, self.y + other.y)
    def scale(self, scalar):
        return Vec2(scalar * self.x, scalar * self.y)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
    def __repr__(self):
        return "Vec2({}, {})".format(self.x, self.y)
```

This has indeed saved us from repeating ourselves. The methods that were identical between `Vec2` and `Vec3` now live in the `Vector` class. All remaining methods on `Vec2` are specific to 2D vectors, they'd need to be modified to work for `Vec3` (as you will see in the exercises) or for vectors with any other number of coordinates.

The `Vector` base class is a good representation of what we can do with vectors. If we can add any useful methods to it, chances are they'll be useful for *any* kind of vector. For instance we can add two methods to `Vector`:

```
class Vector(metaclass=ABCMeta):
    ...
    def subtract(self, other):
        return self.add(-1 * other)
    def __sub__(self, other):
        return self.subtract(other)
```

and without any modification of Vec2 we are automatically able to subtract them:

```
>>> Vec2(1,3) - Vec2(5,1)
Vec2(-4,2)
```

This abstract class will make it easier to implement general vector operations, and it also agrees with the mathematical definition of a vector. Let's switch languages from Python to English, and see how the abstraction can carry over from code to become a real mathematical definition.

6.1.5 Defining vector spaces

In math, a vector is defined by what it does rather than what it is, much like how we defined the abstract `Vector` class. Here's a first (incomplete) definition of a vector.

DEFINITION

A vector is an object equipped with a suitable way to add it to other vectors and multiply it by scalars.

Our `Vec2` or `Vec3` objects, or any other objects inheriting from the `Vector` class can be added to each other and multiplied by scalars. This definition is incomplete, because I haven't said what "suitable" means -- that ends up being the most important part of the definition. There are a few important rules outlawing weird behaviors, many of which you might have already assumed. It's not necessary to memorize all these rules. If you ever find yourself testing whether a new kind of object can be thought of as a vector, you can refer back to them.

The first set of rules say that addition should be well-behaved. Specifically:

1. Adding vectors in any order shouldn't matter: $v + w = w + v$ for any vectors v and w .
2. Adding vectors in any grouping shouldn't matter: $u + (v + w)$ should be the same as $(u + v) + w$, meaning a statement like $u + v + w$ should be unambiguous.

A good counterexample would be "adding" strings by concatenation. In Python you can do the sum `"cat" + "dog"` but it doesn't support the case that strings could be vectors: the sums `"cat" + "dog"` and `"dog" + "cat"` are not equal, violating rule 1.

Scalar multiplication also needs to be well-behaved, and compatible with addition. By compatible with multiplication, I mean that for whole number scalar multiples, multiplication should behave like scalar multiplication.

1. Multiplying vectors by several scalars should be the same as multiplying all at once: if a and b are scalars and v is a vector, then $a \cdot (b \cdot v)$ should be the same as $(a \cdot b) \cdot v$.

2. Multiplying a vector by 1 should leave it unchanged $1 \cdot v = v$.
3. Addition of scalars should be compatible with scalar multiplication: $a \cdot v + b \cdot v$ should be the same as $(a+b) \cdot v$.
4. Addition of vectors should also be compatible with scalar multiplication: $a \cdot (v+w)$ should be the same as $a \cdot v + a \cdot w$.

None of these should be too surprising. For instance $3 \cdot v + 5 \cdot v$ could be translated to English as "3 of v added together plus 5 of v added together." Of course, this is the same as 8 of v added together, or $8 \cdot v$, agreeing with rule five above.

The takeaway from these rules is that not all addition and multiplication operations are created equal. We need to check these properties to verify that the operations really behave like the vector operations. If so, the objects can rightly be called vectors.

A *vector space* is a collection of compatible vectors:

DEFINITION

A vector space is a collection of objects (called vectors), equipped with suitable vector addition and scalar multiplication operations, such that every linear combination of vectors in the collection produces a result vector which is also in the collection.

A collection like `[Vec2(1,0), Vec2(5,-3), Vec2(1.1,0.8)]` is a collection of vectors that can be suitably added and multiplied, but it is not a vector space. For instance `1 * Vec2(1,0) + 1 * Vec2(5,-3)` is a linear combination whose result is `Vec2(6,-3)` which is not in the collection. What *is* a vector space is the infinite collection of all possible 2D vectors. In fact, most vector spaces you'll meet are infinite sets -- there are infinitely many linear combinations using infinitely many scalars after all!

There are two implications of the fact that vector spaces need to contain all their scalar multiples, and these implications are important enough to mention on their own. First of all, no matter what vector v you pick in a vector space, $0 \cdot v$ gives you the same result, called the zero vector and denoted 0 . Adding the zero vector to any vector leaves that vector unchanged: $0 + v = v + 0 = v$. The second implication is that every vector v has an opposite vector, $-1 \cdot v$ written $-v$. By property five above, $v + -v = (1 + -1) \cdot v = 0 \cdot v = 0$. This means every vector has another vector in the vector space which "cancels it out" by addition. As an exercise, you can improve the Vector class by adding a zero vector and a negation function as required members.

A class like `Vec2` or `Vec3` is not a collection per se, but it does describe a collection of values. In this way, we can think of the classes `Vec2` and `Vec3` as representing two different vector spaces, and their instances represent vectors. We'll see a lot more examples of vector spaces with classes that represent them in the next section, but first let's look at how to validate that they satisfy the specific rules we've covered.

6.1.6 Unit testing vector space classes

It was helpful to use an abstract `Vector` base class to think about what a vector should be able to do rather than how. But even giving the base class an abstract “`add`” method doesn’t guarantee every inheriting class will implement a suitable addition operation. In math, the usual way we guarantee suitability is writing a proof. In code, and especially in a dynamic language like Python, the best we can do is write unit tests.

For instance, we can check property 6 above by making up two vectors and a scalar and making sure the equality holds:

```
>>> s = -3
>>> u, v = Vec2(42, -10), Vec2(1.5, 8)
>>> s * (u + v) == s * v + s * u
True
```

This is often how unit tests are written, but it’s a pretty weak test because we’re only trying one example. We can make it stronger by plugging in random numbers and ensuring it works. I’ll use the `random.uniform` function to generate evenly distributed floating point numbers between -10 and 10.

```
from random import uniform

def random_scalar():
    return uniform(-10,10)

def random_vec2():
    return Vec2(random_scalar(),random_scalar())

a = random_scalar()
u, v = random_vec2(), random_vec2()
assert a * (u + v) == a * v + a * u
```

Unless you’re lucky, this test will fail with an `AssertionError`. Here are the offending values of `a`, `u`, and `v` which caused the test to fail for me:

```
>>> a, u, v
(0.17952747449930084,
Vec2(0.8353326458605844,0.2632539730989293),
Vec2(0.555146137477196,0.34288853317521084))
```

And the expressions from the left and right of the equals sign have these values:

```
>>> a * (u + v), a * u + a * v
(Vec2(0.24962914431749222,0.10881923333807299),
Vec2(0.24962914431749225,0.108819233338073))
```

These are two different vectors, but only because their components differ by a few *quadrillionths* -- very small numbers. This doesn’t mean the math is wrong, just that floating point arithmetic is approximate rather than exact. To ignore such small discrepancies we can use another notion of equality suitable for testing. Python’s `math.isclose` function checks that

two float values don't differ by by a significant amount (by default, by more than one billionth of the larger value). Using it instead, the test passes 100 times in a row.

```
from math import isclose

def approx_equal_vec2(v,w):
    return isclose(v.x,w.x) and isclose(v.y,w.y)      ①

for _ in range(0,100):
    a = random_scalar()                            ②
    u, v = random_vec2(), random_vec2()
    assert approx_equal_vec2(a * (u + v), a * v + a * u) ③
```

- ① Test whether the x and y components are close (even if not equal).
- ② Run the test for 100 different randomly generated scalars and pairs of vectors.
- ③ Replace a strict equality check with the new function.

With the floating point error removed from the equation, we can test all six of the vector space properties in this way:

```
def test(eq, a, b, u, v, w): #①
    assert eq(u + v, v + u)
    assert eq(u + (v + w), (u + v) + w)
    assert eq(a * (b * v), (a * b) * v)
    assert eq(1 * v, v)
    assert eq((a + b) * v, a * v + b * v)
    assert eq(a * v + a * w, a * (v + w))

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_vec2(), random_vec2(), random_vec2()
    test(approx_equal_vec2,a,b,u,v,w)
```

- ① We pass in the equality test function as “eq.” This keeps the `test` function agnostic as to the particular concrete vector implementation being passed in.

This test shows that all six properties hold for 100 different random selections of scalars and vectors. That 600 randomized unit tests pass is a good indication that our `Vec2` class satisfies the properties above. Once you've implemented the `zero()` property and the negation operator in the exercises, you can test a few more properties.

This setup isn't completely generic; we had to write special functions to generate random `Vec2` instances and to compare them. The important part is that the test function itself and the expressions within it are completely generic. As long as the class we're testing inherits from `Vector`, it will be able to run expressions like `a * v + a * w` and `a * (v + w)`, which we can then test for equality. Now, we can go wild exploring all the different objects that can be treated as vectors, and we know how to test them as we go.

6.1.7 Exercises

EXERCISE

Implement a “subtract” method and overload the “-” operator for the `Vector` class.

SOLUTION:

```
class Vector(metaclass=ABCMeta):
    ...
    def subtract(self,other):
        return self.add(-1 * other)
    def __sub__(self,other):
        return self.subtract(other)
```

EXERCISE

Implement a `Vec3` class inheriting from `Vector`.

SOLUTION

```
class Vec3(Vector):
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z
    def add(self,other):
        return Vec3(self.x + other.x, self.y + other.y, self.z + other.z)
    def scale(self,scalar):
        return Vec3(scalar * self.x, scalar * self.y, scalar * self.z)
    def __eq__(self,other):
        return self.x == other.x and self.y == other.y and self.z == other.z
    def __repr__(self):
        return "Vec3({}, {}, {})".format(self.x, self.y, self.z)
```

MINI-PROJECT

Implement a `CoordinateVector` class inheriting from `Vector`, with an abstract property representing the dimension. This should save repeated work implementing specific coordinate vector classes; all you should need to do is implement a `Vec6` class that inherits from `CoordinateVector` and setting the dimension to 6.

SOLUTION

We can use the dimension-independent operations `add` and `scale` from chapters 2 and 3. The only thing not implemented in the following class is the dimension, but it still prevents us from instantiating a `CoordinateVector` without knowing what dimension we’re working in.

```
from abc import abstractproperty
from vectors import add, scale

class CoordinateVector(Vector):
```

```

@abstractproperty
def dimension(self):
    pass
def __init__(self,*coordinates):
    self.coordinates = tuple(x for x in coordinates)
def add(self,other):
    return self.__class__(*add(self.coordinates, other.coordinates))
def scale(self,scalar):
    return self.__class__(*scale(scalar, self.coordinates))
def __repr__(self):
    return "{}{}".format(self.__class__.__qualname__, self.coordinates)

```

Once we pick a dimension (say six), we have a concrete class that we can instantiate.

```

class Vec6(CoordinateVector):
    def dimension(self):
        return 6

```

The definitions of addition, scalar multiplication, and so on are picked up from the `CoordinateVector` base class.

```

>>> Vec6(1,2,3,4,5,6) + Vec6(1, 2, 3, 4, 5, 6)
Vec6(2, 4, 6, 8, 10, 12)

```

EXERCISE

Add a “zero” abstract method to `Vector`, designed to return the zero vector in a given vector space, as well as an implementation for the negation operator. These are useful, because we’re required to have a zero vector and negations of any vector in a vector space.

SOLUTION:

```

from abc import ABCMeta, abstractmethod, abstractproperty

class Vector(metaclass=ABCMeta):
    ...
    @classmethod      #1
    @abstractproperty #2
    def zero():
        pass

    def __neg__(self): #3
        return self.scale(-1)

```

We don’t need to implement `__neg__` for any child class, since its definition is included in the parent class, based only on scalar multiplication. We do need to implement “zero.”

```

class Vec2(Vector):
    ...
    def zero():
        return Vec2(0,0)

```

EXERCISE

Write unit tests to show that the addition and scalar multiplication operations for `Vec3` satisfy the vector space properties.

SOLUTION

Since the test function is general, we only need to supply a new equality function for `Vec3` objects and 100 random sets of inputs.

```
def random_vec3():
    return Vec3(random_scalar(),random_scalar(),random_scalar())

def approx_equal_vec3(v,w):
    return isclose(v.x,w.x) and isclose(v.y,w.y) and isclose(v.z, w.z)

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_vec3(), random_vec3(), random_vec3()
    test(a,b,u,v,w)
```

EXERCISE

Add unit tests to check that $0 + v = v$, $0 \cdot v = 0$, and $-v + v = 0$ for any vector v .

SOLUTION

Since the zero vector is different depending on which class we're testing, we need to pass it in as an argument to the function:

```
def test(zero,eq,a,b,u,v,w):
    ...
    assert eq(zero + v, v)
    assert eq(0 * v, zero)
    assert eq(-v + v, zero)
```

We can update the `Vec3` tests from the previous exercise as follows.

```
for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_vec3(), random_vec3(), random_vec3()
    test(Vec3.zero(), approx_equal_vec2, a,b,u,v,w)
```

EXERCISE

As equality is implemented above for `Vec2` and `Vec3`, it turns out that `Vec2(1,2) == Vec3(1,2,3)` returns True. Python's duck typing is too forgiving for its own good! Fix this by adding a check that classes match before testing vector equality.

SOLUTION

It turns out we need to do the check for addition as well!

```
class Vec2(Vector):
    ...
    def add(self,other):
        assert self.__class__ == other.__class__
        return Vec2(self.x + other.x, self.y + other.y)
    ...
    def __eq__(self,other):
        return (self.__class__ == other.__class__
                and self.x == other.x and self.y == other.y)
```

To be safe, you can add checks like this to other child classes of `Vector` as well.

EXERCISE

Implement a `__truediv__` function on `Vector`, allowing us to divide vectors by scalars. We can divide vectors by a non-zero scalar by multiplying them by the reciprocal of the scalar (1.0/scalar).

SOLUTION:

```
class Vector(metaclass=ABCMeta):
    ...
    def __truediv__(self, scalar):
        return self.scale(1.0/scalar)
```

With this implemented, we can do division like `Vec2(1,2) / 2`, getting back `Vec2(0.5,1.0)`.

6.2 Exploring different vector spaces

Now that you know what a vector space is, let's take a look at a bunch of examples. In each case, we'll take a new kind of object and implement it as a class which inherits from `Vector`. At that point, no matter what kind of object it is, we'll be able to do addition, scalar multiplication, or any other vector operation with it.

6.2.1 Enumerating all coordinate vector spaces

We've spent a lot of time on the coordinate vectors `Vec2` and `Vec3` so far, so coordinate vectors don't need much more explanation. It is worth reviewing the fact that a vector space of coordinate vectors can have any number of coordinates. `Vec2` vectors have two coordinates, `Vec3` vectors have three, and we could just as well have a `Vec15` class with 15 coordinates. We can't picture it geometrically, but `Vec15` objects represent points in a 15-dimensional space.

One special case worth mentioning is the class we might call `Vec1`: vectors with a single coordinate. The implementation looks like this:

```
class Vec1(Vector):
    def __init__(self,x):
        self.x = x
    def add(self,other):
        return Vec1(self.x + other.x)
    def scale(self,scalar):
        return Vec1(scalar * self.x)
```

```

@classmethod
def zero(cls):
    return Vec1(0)
def __eq__(self,other):
    return self.x == other.x
def __repr__(self):
    return "Vec1({})".format(self.x)

```

This is a lot of boilerplate to wrap a single number, and it doesn't give us any arithmetic we didn't already have. Adding and scalar multiplying `Vec1` objects is just addition and multiplication of the underlying numbers.

```

>>> Vec1(2) + Vec1(2)
Vec1(4)
>>> 3 * Vec1(1)
Vec1(3)

```

For this reason, we probably never need a `Vec1` class. But it is important to know that numbers on their own are vectors. The set of all real numbers (including integers, fractions, and irrational numbers like π) is denoted \mathbb{R} and it is a vector space on its own. This is a special case where the scalars and the vectors are the same kind of object.

Coordinate vector spaces are denoted \mathbb{R}^n where n is the dimension, or number of coordinates. For instance, the 2D plane is denoted \mathbb{R}^2 and 3D space is denoted \mathbb{R}^3 . As long as you use real numbers as your scalars, any vector space you stumble across will be some \mathbb{R}^n in disguise¹. This is why we need to mention the vector space \mathbb{R} , even if it is boring. The other vector space we need to mention is the *zero-dimensional* one, \mathbb{R}^0 . This is the set of vectors with zero coordinates, which we could describe as empty tuples or as a `Vec0` class inheriting from `Vector`:

```

class Vec0(Vector):
    def __init__(self):
        pass
    def add(self,other):
        return Vec0()
    def scale(self,scalar):
        return Vec0()
    @classmethod
    def zero(cls):
        return Vec0()
    def __eq__(self,other):
        return self.__class__ == other.__class__ == Vec0
    def __repr__(self):
        return "Vec0()"

```

¹ That is, as long as you can guarantee your vector space has only finitely many dimensions! There is a vector space called \mathbb{R}^∞ , but it is not the only infinitely dimensional vector space.

No coordinates doesn't mean that there are no possible vectors; it means there is exactly one zero-dimensional vector. This makes zero-dimensional vector math stupidly easy: any result vector is always the same.

```
>>> - 3.14 * Vec0()
Vec0()
>>> Vec0() + Vec0() + Vec0() + Vec0()
Vec0()
```

This is something like a singleton class from an object-oriented programming perspective. From a mathematical perspective, we know that every vector space has to have a 0 vector, so we can think of `Vec0()` as being this zero vector.

That covers it for coordinate vectors of dimensions zero, one, two, three, or more. Now, when you see a vector in the wild, you'll be able to match it up with one of these vector spaces. Here's an example.

6.2.2 Identifying vector spaces in the wild

Let's return to an example from chapter 1 and look at a data set of used Toyota Priuses. In the source code, you'll see how to load a data set generously provided by my friend Dan Rathbone at CarGraph.com. To make the cars easy to work with, I've loaded them into a class:

```
class CarForSale():
    def __init__(self, model_year, mileage, price, posted_datetime,
                 model, source, location, description):
        self.model_year = model_year
        self.mileage = mileage
        self.price = price
        self.posted_datetime = posted_datetime
        self.model = model
        self.source = source
        self.location = location
        self.description = description
```

It would be useful to think of `CarForSale` objects as vectors. Then, for example, I could average them together as a linear combination to see what the typical Prius for sale looks like. To do that I need to retrofit this class to inherit from `Vector`.

How can we add two cars? The numeric fields `model_year`, `mileage`, and `price` can be added like components of a vector, but the string properties can't be added in a meaningful way (we saw we can't think of strings as vectors). When we do arithmetic on cars, the result is not a real car for sale but a "virtual" car defined by its properties. To represent this, I'll change all the string properties to the string "(virtual)" to remind us of this. Finally, datetimes can't be added but time spans can be. I'll use the day I retrieved the data as a reference point, and add the time spans since the cars were posted for sale.

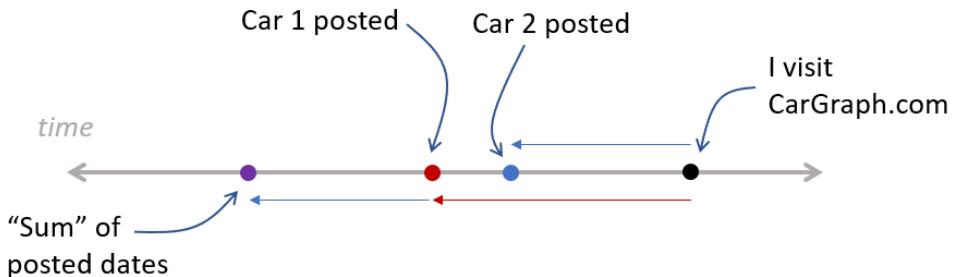


Figure 6.3 Timeline of cars posted for sale.

All this applies to scalar multiplication as well. We can multiply the numeric properties and the timespan since posting by a scalar. The string properties will no longer be meaningful.

Listing: Making the `CarForSale` class behave like a Vector by implementing the required methods.

```
class CarForSale(Vector):
    retrieved_date = datetime(2018,11,30,12)           ①
    def __init__(self, model_year, mileage, price, posted_datetime,
                 model="(virtual)", source="(virtual)", ②
                           location="(virtual)", description="(virtual)"):
        self.model_year = model_year
        self.mileage = mileage
        self.price = price
        self.posted_datetime = posted_datetime
        self.model = model
        self.source = source
        self.location = location
        self.description = description
    def add(self, other):
        def add_dates(d1, d2): ③
            age1 = CarForSale.retrieved_date - d1
            age2 = CarForSale.retrieved_date - d2
            sum_age = age1 + age2
            return CarForSale.retrieved_date - sum_age
        return CarForSale( ④
            self.model_year + other.model_year,
            self.mileage + other.mileage,
            self.price + other.price,
            add_dates(self.posted_datetime, other.posted_datetime)
        )
    def scale(self,scalar):
        def scale_date(d): ⑤
            age = CarForSale.retrieved_date - d
            return CarForSale.retrieved_date - (scalar * age)
        return CarForSale(
            scalar * self.model_year,
            scalar * self.mileage,
            scalar * self.price,
            scale_date(self.posted_datetime)
        )
```

- 1 I retrieved the data set from CarGraph.com on 11/30/2018 at noon.
- 2 To simplify construction of virtual cars, all of the string parameters are optional with default value “(virtual)”.
- 3 Helper function which adds dates by adding the time spans from the reference date.
- 4 As with our other vectors, we add CarForSale objects by adding underlying properties and constructing a new object.
- 5 Helper function to scale a datetime by scaling the timespan from the reference date.

With the list of cars loaded, we can try some vector arithmetic:

```
>>> (cars[0] + cars[1]).__dict__
{'model_year': 4012,
 'mileage': 306000.0,
 'price': 6100.0,
 'posted_datetime': datetime.datetime(2018, 11, 30, 3, 59),
 'model': '(virtual)',
 'source': '(virtual)',
 'location': '(virtual)',
 'description': '(virtual)'}
```

The sum of the first two cars is evidently a Prius from model year 4012 (maybe it can fly?) with 306,000 miles on it and going for a price of \$6,100. It was posted for sale at 3:59 AM on the same day I looked at CarGraph.com. This doesn’t look too helpful, but bear with me; averages will look a lot more meaningful.

```
>>> average_prius = sum(cars, CarForSale.zero()) * (1.0/len(cars))
>>> average_prius.__dict__
{'model_year': 2012.5365853658536,
 'mileage': 87731.63414634147,
 'price': 12574.731707317074,
 'posted_datetime': datetime.datetime(2018, 11, 30, 9, 0, 49, 756098),
 'model': '(virtual)',
 'source': '(virtual)',
 'location': '(virtual)',
 'description': '(virtual)'}
```

We can learn real things from this result. The average prius for sale is about 6 years old, has about 88,000 miles on it, is selling for about \$12,500, and was posted at 9:49 AM the morning I accessed the website. In Part 3, we’ll spend a lot of time learning from data sets by treating them as vectors.

Ignoring the text data, CarForSale behaves like a vector. In fact, it behaves like a 4D vector, having dimensions price, model year, mileage, and datetime of posting. It’s not quite a coordinate vector because the posting date is not a number. Even though the data is not numeric, the class satisfies the vector space properties (you’ll verify this with unit tests in the exercises) so its objects are vectors and can be manipulated as such. Specifically, they are 4D vectors so it is possible to write a 1-to-1 mapping between CarForSale objects and Vec4 objects (also an exercise for you). For our next example, we’ll look at objects that look even less like coordinate vectors, but still satisfy the defining properties.

6.2.3 Treating functions as vectors

It turns out that mathematical functions can be thought of as vectors. Specifically, I'm talking about functions that take in a single real number and return a single real number, though there are plenty of other types of mathematical functions. The mathematical shorthand to say that a function f takes any real number and returns a real number is $f : \mathbb{R} \rightarrow \mathbb{R}$. In Python, we'll be thinking of functions that take `float` values in and return `float` values.

As with 2D or 3D vectors, we can do addition and scalar multiplication of functions visually or algebraically. To start, we can write functions algebraically, for instance $f(x) = 0.5 \cdot x + 3$ or $g(x) = \sin(x)$. Alternatively, we can visualize them by graphing them.

In the source code I've written you a simple `plot` function that draws the graph of one or more functions on a specified range of inputs. For instance, the following code plots both of our functions $f(x)$ and $g(x)$ on x values between -10 and 10.

```
def f(x):
    return 0.5 * x + 3
def g(x):
    return sin(x)
plot([f,g], -10, 10)
```

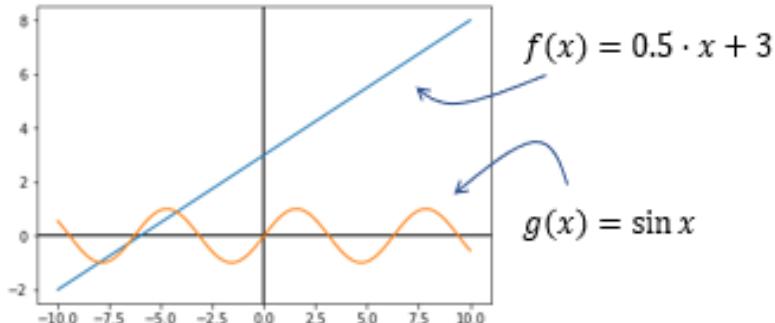


Figure 6.4 Graphs of the functions $f(x) = 0.5 \cdot x + 3$ and $g(x) = \sin(x)$

Algebraically, we add functions by adding their expressions. This means $f + g$ is a function defined by $(f + g)(x) = f(x) + g(x) = 0.5 \cdot x + 3 + \sin(x)$. Graphically, the y-values of each point are added, so it's something like stacking the two functions together.

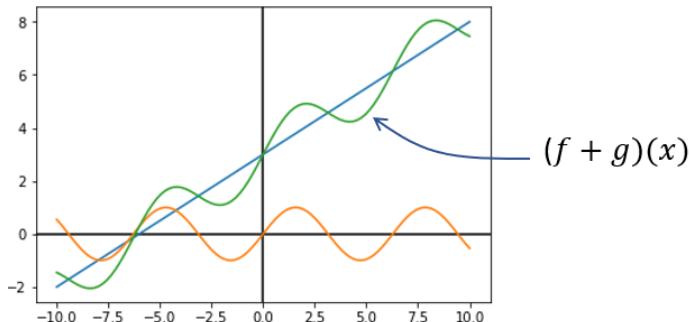


Figure 6.5 Visualizing the sum of two functions by graphing it.

To implement this sum, you can write some functional Python again. This code takes two functions as inputs and returns a new one which is their sum.

```
def add_functions(f,g):
    def new_function(x):
        return f(x) + g(x)
    return new_function
```

Likewise we can multiply a function by a scalar by multiplying its expression by the scalar. For instance $3g$ is defined by $(3g)(x) = 3 \cdot g(x) = 3 \cdot \sin(x)$. This has the effect of stretching the graph of the function g in the y -direction by a factor of 3.

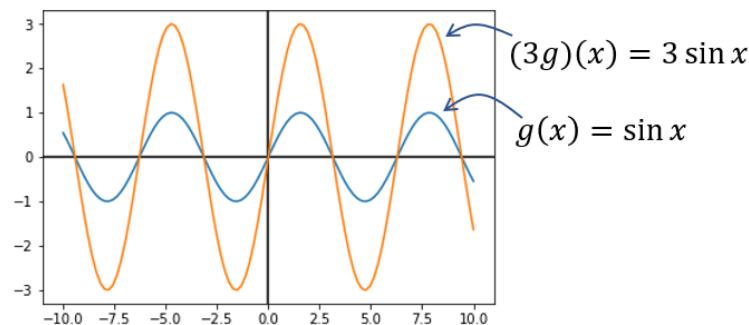


Figure 6.6 the function $(3g)$ looks like the function g stretched by a factor of 3 in the y -direction.

It's possible to nicely wrap Python functions in a class that inherits from `vector`, and I leave it as an exercise. After doing so, you can write satisfying function arithmetic expressions like $3 * f$ or $2 * f - 6 * g$. You can even make the class *callable*, or able to accept arguments as if it were a function, to allow expressions like $(f + g)(6)$. Unfortunately, unit testing to determine if functions satisfy the vector space properties is much harder, because it's hard to generate random functions or to test whether two functions are equal. To really know if two

functions are equal, you have to know that they return the same output for every single possible input. That would mean a test for every real number, or at least every `float` value!

This brings us to another question: what is the *dimension* of the vector space of functions? Or, to be concrete, how many real number coordinates would be needed to uniquely identify a function? Instead of naming the coordinates of a `Vec3` x , y , and z , you could index them from $i = 1$ to 3. Likewise you could index the coordinates of a `Vec15` from $i = 1$ to 15. A function, however, has infinitely many numbers that define it: the values $f(x)$ for any value of x . In other words, you can think of the coordinates of f as being its values at every point, indexed by all real numbers instead of the first few integers. This means that the vector space of functions is *infinite dimensional*. This has important implications, but it mostly makes the vector space of all functions hard to work with. We'll return to this space later, specifically looking at some subsets that are simpler. For now, let's return to the comfort of finitely-many dimensions and look at two more examples.

6.2.4 Treating matrices as vectors

Because an $n \times m$ matrix is a list of $n \cdot m$ numbers, albeit arranged in a rectangle, we can treat it as a $n \cdot m$ -dimensional vector. The only difference between the vector space of, say, 5×3 matrices from the vector space of 15-dimensional coordinate vectors is that the coordinates are presented in a matrix. We still add and scalar multiply coordinate-by-coordinate. Here's how addition looks.

$$\begin{pmatrix} 2 & 0 & 6 \\ 10 & 6 & -3 \\ 9 & -3 & -5 \\ -5 & 5 & 5 \\ -2 & 8 & -2 \end{pmatrix} + \begin{pmatrix} -3 & -4 & 3 \\ 1 & -1 & 8 \\ -1 & -3 & 8 \\ 4 & 7 & -4 \\ -3 & 10 & 6 \end{pmatrix} = \begin{pmatrix} -1 & -4 & 9 \\ 11 & 5 & 5 \\ 8 & -6 & 3 \\ -1 & 12 & 1 \\ -5 & 18 & 4 \end{pmatrix}$$

$$2 + (-3) = -1$$

Figure 6.7 Adding two 5×3 matrices by adding their corresponding entries.

Implementing a class for 5×3 matrices inheriting from `Vector` is more typing than implementing a `Vec15` class, since you need two loops to iterate over a matrix, but the arithmetic is no more complicated.

Listing: a class representing 5×3 matrices, thought of as vectors.

```
class Matrix5_by_3(Vector):
    rows = 5
    columns = 3
    def __init__(self, matrix):
        self.matrix = matrix
```

```

def add(self, other):
    return Matrix5_by_3(tuple(
        tuple(a + b for a,b in zip(row1, row2))
        for (row1, row2) in zip(self.matrix, other.matrix)
    ))
def scale(self, scalar):
    return Matrix5_by_3(tuple(
        tuple(scalar * x for x in row)
        for row in self.matrix
    ))
@classmethod
def zero(cls):
    return Matrix5_by_3(tuple(
        tuple(0 for j in range(0, cls.columns)) ②
        for i in range(0, cls.rows)
    ))

```

- ① You need to know the number of rows and columns to be able to construct the zero matrix.
 ② The “zero vector” for 5×3 matrices is a 5×3 matrix consisting of all zeroes. Adding this to any other 5×3 matrix M will return M .

You could just as well create a `Matrix2_by_2` class or a `Matrix99_by_17` class to represent different vector spaces. In these cases, much of the implementation would be the same, but the dimensions would no longer be 15, they would be $2 \cdot 2 = 4$ or $99 \cdot 17 = 1683$. As an exercise, you can create a `Matrix` class inheriting from `Vector` that includes all the data except for specified numbers of rows and columns. Then any `MatrixM_by_N` class could inherit from `Matrix`.

The interesting thing about matrices isn’t that they are numbers arranged in grids, but rather that we can think of them as representing linear functions. We’ve already seen that lists of numbers and functions are two cases of vector spaces, but it beautifully turns out that matrices are vectors in both senses. If a matrix A has n rows and m columns, it represents a linear function from m -dimensional space to n -dimensional space (you could write $A : \mathbb{R}^m \rightarrow \mathbb{R}^n$ to say this same sentence in mathematical shorthand). Just as we added and scalar-multiplied functions from $\mathbb{R} \rightarrow \mathbb{R}$, so can we add and scalar multiply functions $\mathbb{R}^m \rightarrow \mathbb{R}^n$. In a mini-project at the end of the section, you can try running the vector space unit tests on matrices to check they are vectors in both senses.

That doesn’t mean grids of numbers aren’t useful in their own right; sometimes we don’t care to interpret them as functions. For instance, arrays of numbers can be used to represent images.

6.2.5 Manipulating images with vector operations

On a computer, images are displayed as arrays of colored squares called *pixels*. A typical image may be a few hundred pixels tall by a few hundred pixels wide. In a color image, three numbers are needed to specify the red, green, and blue content of the color of any given pixel. In total, a 300 pixel by 300 pixel image is specified by $300 \cdot 300 \cdot 3 = 270,000$

numbers. Thinking of images of this size as vectors, they live in a 270,000-dimensional space!



Figure 6.8 Zooming in on a picture of my dog, Melba, until we can pick out one pixel, with red, green, and blue content (230, 105, 166) respectively.

Depending on what format you're reading this, you may or may not see the pink color of Melba's tongue. But because we'll represent color numerically rather than visually in this discussion, everything should still make sense. You can also see the pictures in full color in the source code.

Python has a de-facto standard image manipulation library PIL, which is distributed in pip under the package name "pillow." You won't need to learn much about the library because we'll immediately encapsulate our use of it inside of a new class. This class, `ImageVector`, will inherit from `Vector`, store the pixel data of a 300 by 300 image, and support addition and scalar multiplication.

Listing: A class representing an image as a Vector.

```
from PIL import Image

class ImageVector(Vector):
    size = (300,300)
    def __init__(self,input):
        try:
            img = Image.open(input).resize(ImageVector.size) ①
            self.pixels = img.getdata()
        except:
            self.pixels = input
    def image(self):
        img = Image.new( 'RGB', image_size) ②
        img.putdata([(int(r), int(g), int(b))
                    for (r,g,b) in self.pixels])
        return img
    def add(self,img2): ③
        return ImageVector([(r1+r2,g1+g2,b1+b2)
                           for ((r1,g1,b1),(r2,g2,b2))
                           in zip(self.pixels,img2.pixels)]) ④
    def scale(self,scalar): ⑤
        return ImageVector([(scalar*r,scalar*g,scalar*b)
                           for (r,g,b) in self.pixels])
    @classmethod ⑥
    def zero(cls): ⑦
```

```

    total_pixels = cls.size[0] * cls.size[1]
    return ImageVector([(0,0,0) for _ in range(0,total_pixels)])
def __repr_png__(self):
    return self.image().__repr_png__()

```

- ➊ This class is meant to handle images of fixed size: 300 pixels by 300 pixels.
- ➋ The constructor can accept the name of an image file. We create an `Image` object with PIL, resize it to 300 by 300, and then extract its list of pixels with the `getdata()` method. Each pixel is a triple consisting of a red, green, and blue value.
- ➌ We allow the constructor to also accept a list of pixels directly.
- ➍ This method returns the underlying PIL `Image`, reconstructed from the pixels stored as an attribute on the class. The values must be converted to integers to create a displayable image.
- ➎ Vector addition for images is done by adding the respective red, green, and blue values for each pixel.
- ➏ Scalar multiplication is done by multiplying every red, green, and blue value for every pixel by the given scalar.
- ➐ The zero image has zero red, green, or blue content at any pixel.
- ➑ Jupyter notebooks can display PIL Images inline, as long as we pass the implementation of the function `_repr_png_` along from the underlying image.

Equipped with this library, we can load images by file name and do vector arithmetic with them. For instance, the average of two pictures can be written as a linear combination:

```
0.5 * ImageVector("inside.JPG") + 0.5 * ImageVector("outside.JPG")
```



Figure 6.9 The average of two images of Melba as a linear combination.

While any `ImageVector` is valid, the minimum and maximum color values that render as visually different are 0 and 255 respectively. Because of this, the negative of any image you import will be black, having gone below the minimum brightness at every pixel. Likewise, positive scalar multiples quickly become washed out, with most pixels exceeding the maximum displayable brightness.



Figure 6.10 Negation and scalar multiplication of an image.

To make visually interesting changes, you need to do operations that will land you in the right brightness range for all colors. The zero vector (black) and the vector with all values equal to 255 (white) are good reference points. For instance, subtracting an image from an all white image has the effect of reversing the colors. If our white vector is

```
white = ImageVector([(255,255,255) for _ in range(0,300*300)])
```

then subtracting an image from it yields an eerily recolored picture (the difference should be striking even if you're looking at the picture in black and white).



Figure 6.11 Reversing the color of an image by subtracting it from a plain white image.

Vector arithmetic is clearly a very general concept: the defining concepts of addition and scalar multiplication apply to numbers, coordinate vectors, functions, matrices, images, and many other kinds of objects. It's striking to see such visual results when we apply the same math across unrelated domains. We'll keep all of these examples of vector spaces in mind, and continue to explore generalizations we can make across them.

6.2.6 Exercises

EXERCISE

Run the vector space unit tests with `u`, `v`, and `w` as floats rather than objects inheriting `Vector`. This demonstrates that real numbers are indeed vectors.

SOLUTION

With vectors as random scalars, the number zero as the zero vector, and `math.isclose` as the equality test, the 100 random tests pass.

```
for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_scalar(), random_scalar(), random_scalar()
    test(0, isclose, a,b,u,v,w)
```

EXERCISE

What is the zero vector for a `CarForSale`? Implement the `CarForSale.zero()` function to make it available.

SOLUTION:

```
class CarForSale(Vector):
    ...
    @classmethod
    def zero(cls):
        return CarForSale(0, 0, 0, CarForSale.retrieved_date)
```

MINI-PROJECT

Run the vector space unit tests for `CarForSale` to show its objects form a vector space (ignoring their textual attributes).

SOLUTION

Most of the work is generating random data and building an approximate equality test that handles datetimes.

```
from math import isclose
from random import uniform, random, randint
from datetime import datetime, timedelta

def random_time():
    return CarForSale.retrieved_date - timedelta(days=uniform(0,10))

def approx_equal_time(t1, t2):
    test = datetime.now()
    return isclose((test-t1).total_seconds(), (test-t2).total_seconds())

def random_car():
    return CarForSale(randint(1990,2019), randint(0,250000),
                    27000. * random(), random_time())
```

```

def approx_equal_car(c1,c2):
    return (isclose(c1.model_year,c2.model_year)
            and isclose(c1.mileage,c2.mileage)
            and isclose(c1.price, c2.price)
            and approx_equal_time(c1.posted_datetime, c2.posted_datetime))

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_car(), random_car(), random_car()
    test(CarForSale.zero(), approx_equal_car, a,b,u,v,w)

```

EXERCISE

Implement class `Function(Vector)` that takes a function of 1 variable as an argument to its constructor, with a `__call__` implemented so we can treat it as a function. You should be able to run `plot([f,g,f+g,3*g],-10,10)`.

SOLUTION:

```

class Function(Vector):
    def __init__(self, f):
        self.function = f
    def add(self, other):
        return Function(lambda x: self.function(x) + other.function(x))
    def scale(self, scalar):
        return Function(lambda x: scalar * self.function(x))
    @classmethod
    def zero(cls):
        return Function(lambda x: 0)
    def __call__(self, arg):
        return self.function(arg)

f = Function(lambda x: 0.5 * x + 3)
g = Function(sin)

plot([f, g, f+g, 3*g], -10, 10)

```

The result of the last line is the plot below:

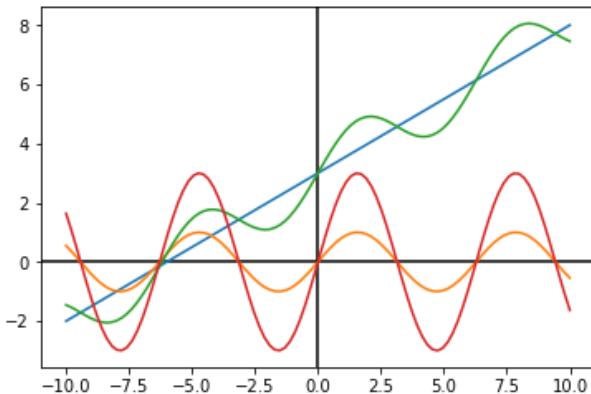


Figure 6.12 Our objects f and g behave like vectors, so we can add and scalar multiply them. Since they also behave like functions we can plot them.

MINI-PROJECT

Testing equality of functions is difficult. Do your best to write a function to test whether two functions are equal.

SOLUTION

Since we're usually interested in well-behaved, continuous functions it may be enough to check that their values are close for a few random input values.

```
def approx_equal_function(f,g):
    results = []
    for _ in range(0,10):
        x = uniform(-10,10)
        results.append(isclose(f(x),g(x)))
```

return all(results)Unfortunately it can give us misleading results. The following returns true, even though the functions cannot be equal at zero.

```
approx_equal_function(lambda x: (x*x)/x, lambda x: x)
```

It turns out that computing equality of functions is an *undecidable* problem. That is, it has been proved there is no algorithm that can guarantee whether *any* two functions are the same.

MINI-PROJECT

Unit test your Function class to demonstrate that functions satisfy the vector space properties.

SOLUTION

It's difficult to test function equality, and it's also difficult to generate random functions. I used the `Polynomial` class (that you'll meet in the next section) to generate some "random" functions. Using `approx_equal_function` from the previous mini-project, we can get the tests to pass.

```

def random_function():
    degree = randint(0,5)
    p = Polynomial(*[uniform(-10,10) for _ in range(0,degree)])
    return Function(lambda x: p(x))

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_function(), random_function(), random_function()
    test(Function.zero(), approx_equal_function, a,b,u,v,w)

```

MINI-PROJECT

Implement a class `Function2(Vector)` that stores a function of two variables, like $f(x,y) = x + y$.

SOLUTION

The definition is not much different than the `Function` class, but all functions are given two arguments.

```

class Function(Vector):
    def __init__(self, f):
        self.function = f
    def add(self, other):
        return Function(lambda x,y: self.function(x,y) + other.function(x,y))
    def scale(self, scalar):
        return Function(lambda x,y: scalar * self.function(x,y))
    @classmethod
    def zero(cls):
        return Function(lambda x,y: 0)
    def __call__(self, *args):
        return self.function(*args)

```

For instance, the sum of $f(x,y) = x+y$ and $g(x,y) = x - y + 1$ should be $2x + 1$. We can confirm this:

```

>>> f = Function(lambda x,y:x+y)
>>> g = Function(lambda x,y: x-y+1)
>>> (f+g)(3,10)
7

```

EXERCISE

What is the dimension of the vector space of 9 by 9 matrices?

- a.) 9
- b.) 18
- c.) 27
- d.) 81

MINI-PROJECT

A 9 by 9 matrix has 81 entries, so there are 81 independent numbers (or coordinates) that determine it. It therefore is an 81-dimensional vector space, and answer d is correct.

MINI-PROJECT

Implement a `Matrix` class inheriting from `Vector` with abstract properties representing number of rows and number of columns. You should not be able to instantiate a `Matrix` class, but you could make a `Matrix5_by_3` by inheriting from `Matrix` and specifying the number of rows and columns explicitly.

SOLUTION:

```
class Matrix(Vector):
    @abstractproperty
    def rows(self):
        pass
    @abstractproperty
    def columns(self):
        pass
    def __init__(self, entries):
        self.entries = entries
    def add(self, other):
        return self.__class__(
            tuple(
                tuple(self.entries[i][j] + other.entries[i][j]
                      for j in range(0, self.columns()))
                    for i in range(0, self.rows())))
    def scale(self, scalar):
        return self.__class__(
            tuple(
                tuple(scalar * e for e in row)
                    for row in self.entries))
    def __repr__(self):
        return "%s(%r)" % (self.__class__.__qualname__, self.entries)
    def zero(self):
        return self.__class__(
            tuple(
                tuple(0 for i in range(0, self.columns()))
                    for j in range(0, self.rows())))
```

We can now quickly implement any class representing a vector space of matrices of fixed size, for instance 2 by 2:

```
class Matrix2_by_2(Matrix):
    def rows(self):
        return 2
    def columns(self):
        return 2
```

Then we can compute with 2 by 2 matrices as vectors.

```
>>> 2 * Matrix2_by_2(((1,2),(3,4))) + Matrix2_by_2(((1,2),(3,4)))
Matrix2_by_2((3, 6), (9, 12))
```

EXERCISE

Unit test the `Matrix5_by_3` class to demonstrate that it obeys the defining properties of a vector space.

SOLUTION:

```

def random_matrix(rows, columns):
    return tuple(
        tuple(uniform(-10,10) for j in range(0,columns))
        for i in range(0,rows)
    )

def random_5_by_3():
    return Matrix5_by_3(random_matrix(5,3))

def approx_equal_matrix_5_by_3(m1,m2):
    return all([
        isclose(m1.matrix[i][j],m2.matrix[i][j])
        for j in range(0,3)
        for i in range(0,5)
    ])

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_5_by_3(), random_5_by_3(), random_5_by_3()
    test(Matrix5_by_3.zero(), approx_equal_matrix_5_by_3, a,b,u,v,w)

```

MINI-PROJECT:

Write a `LinearMap3d_to_5d` class, inheriting from `Vector`, that uses a 5 by 3 matrix as its data but implements `__call__` to act as a linear map from \mathbb{R}^3 to \mathbb{R}^5 . Show that it agrees with `Matrix5_by_3` in its underlying computations, and that it independently passes the defining properties of a vector space.

EXERCISE

Write a Python function enabling you to multiply `Matrix5_by_3` objects by `Vec3` objects, in the sense of matrix multiplication.

MINI-PROJECT

Update your overloading of the `**` operator for the vector and matrix classes so you can multiply vectors on their left by either scalars or matrices.

EXERCISE

Convince yourself that the `"zero"` vector for the `ImageVector` class doesn't visibly alter any image when it is added.

SOLUTION

Look at the result of `ImageVector("my_image.jpg") + ImageVector.zero()` for any image of your choice.

EXERCISE

Pick two images and display 10 different weighted averages of them. These will be “points on a line segment” connecting the images in 270,000-dimensional space!

SOLUTION

I ran the following code with $s = 0.1, 0.2, 0.3, \dots, 0.9, 1.0$.

```
s * ImageVector("inside.JPG") + (1-s) * ImageVector("outside.JPG")
```

When you put your images side by side, you'll get something like this.



Figure 6.13 Several different weighted averages of two images.

EXERCISE

Adapt the vector space unit tests to images, and run them. What do your randomized unit tests look like as images?

SOLUTION

One way to generate random images is to put random red, green, and blue values at every pixel.

```
def random_image():
    return ImageVector([(randint(0,255), randint(0,255), randint(0,255))
                        for i in range(0,300 * 300)])
```

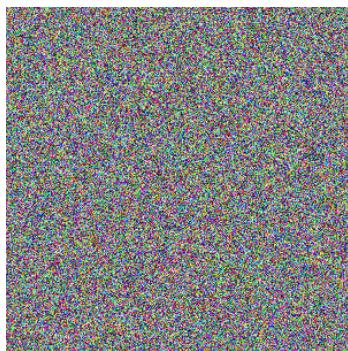


Figure 6.14 A “random” image.

The result is a fuzzy mess, but that doesn't matter to us. The unit tests will compare them numerically, pixel by pixel. With an approximate equality test, we're able to run the tests.

```
def approx_equal_image(i1,i2):
```

```

    return all([isclose(c1,c2)
               for p1,p2 in zip(i1.pixels,i2.pixels)
               for c1,c2 in zip(p1,p2)])

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_image(), random_image(), random_image()
    test(ImageVector.zero(), approx_equal_image, a,b,u,v,w)

```

6.3 Looking for smaller vector spaces

The vector space of 300-by-300 color images has a whopping 270,000 dimensions, meaning we need to list as many numbers to specify any image of that size. This isn't a problematic amount of data on its own, but when we have larger images, a large quantity of images, or thousands of images chained together to make a movie, the data can add up.

In this section, and as a theme that will recur through the rest of the book, we'll look at how to start with a vector space and find smaller ones (having fewer dimensions) that retain most of the interesting data from the original space. With images, we can reduce the number of distinct pixels used in an image or convert it to black and white. The result may not be beautiful, but it can still be recognizable. For instance, the image on the left takes 900 numbers to specify, compared to the 270,000 numbers to specify the image on the right.



Figure 6.15 Converting from an image specified by 270,000 numbers to another one specified by 900 numbers.

Pictures that look like the one on the right live in a 900-dimensional *subspace* of a 270,000-dimensional space. That means they are still 270,000-dimensional image vectors but that they can be represented or stored with only 900 coordinates. This is a starting point for a

study of *compression*. We won't go too deep into the best practices of compression, but we will take a close look at subspaces of vector spaces.

6.3.1 Identifying subspaces

A subspace is a vector space that exists inside another vector space. One example we've looked at a few times already is the 2D x,y -plane within 3D space, as the plane where $z = 0$. To be specific, the subspace consists of vectors of the form $(x,y,0)$. These vectors have three components, so they are veritable 3D vectors, but they form a subset that happens to be constrained to lie on a plane. For that reason, we say this is a two-dimensional subspace of \mathbb{R}^3 .

NOTE

At risk of being pedantic, the 2D vector space \mathbb{R}^2 , which consists of pairs (x,y) , is not technically a subspace of 3D space \mathbb{R}^3 . That's because vectors of the form (x,y) are not 3D vectors. However, it has a one-to-one correspondence the set of vectors $(x,y,0)$, and vector arithmetic looks the same whether or not the extra zero z -coordinate is present. For these reasons, we won't cause too many problems if we call \mathbb{R}^2 a subspace of \mathbb{R}^3 .

Not every subset of 3D vectors is a subspace. The plane where $z = 0$ is special because the vectors $(x,y,0)$ form a self-contained vector space. There's no way to build a linear combination of vectors in this plane that somehow "escape" it; the third coordinate will always remain zero. In math lingo, the precise way to say that a subspace is "self-contained" is to say it is *closed* under linear combinations.

To get the feel for what a vector subspace looks like in general, let's search for subsets of vector spaces which are also subspaces. What subsets of vectors in the plane can make a standalone vector space? Can we just draw any region in the plane and only take vectors that live within it?

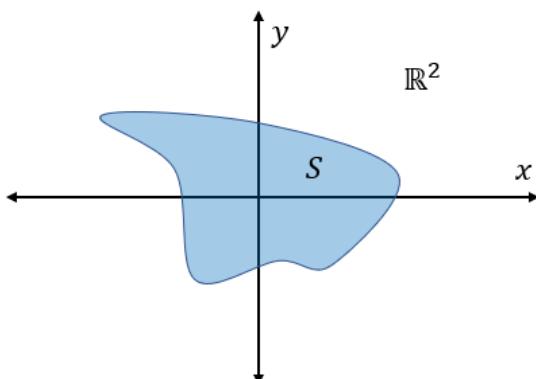


Figure 6.16 S is a subset of points (vectors) in the plane, \mathbb{R}^2 . Is S a subspace of \mathbb{R}^2 ?

The answer is no: for instance the subset drawn above contains some vectors that lie on the x -axis and some that live on the y -axis. These can respectively be scaled to give us the standard basis vectors $e_1 = (1,0)$ and $e_2 = (0,1)$. From these, we can make linear combinations to get to any point in the plane, not only the ones in S .

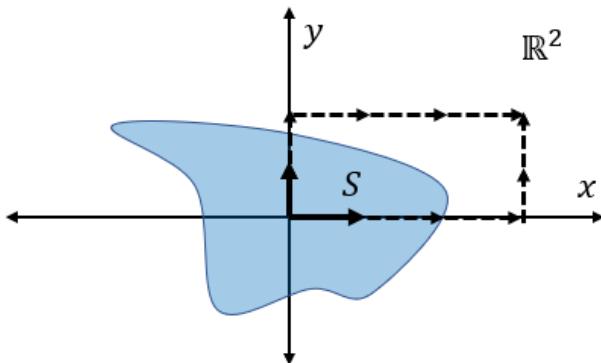


Figure 6.17 Linear combinations of two vectors in S give us an “escape route” from S . It cannot be a subspace of the plane.

Instead of drawing a random subspace, let’s mimic the example of the plane in 3D. There is no z -coordinate, so let’s instead choose the points where $y = 0$. This leaves us with the points on the x -axis, having the form $(x,0)$. No matter how hard we try, we can’t find a linear combination of vectors that look like this and cause them to have a non-zero y -coordinate.

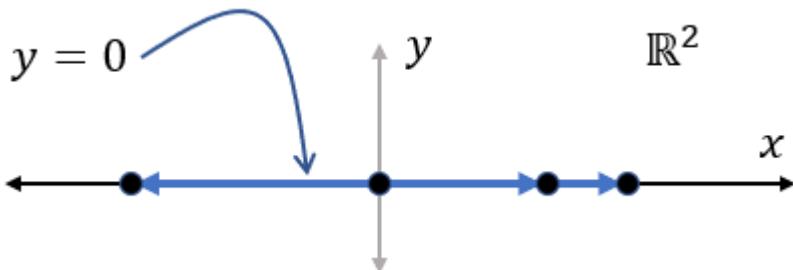


Figure 6.18 Focusing on the line where $y = 0$. This is a vector space, containing all linear combinations of its points.

This is a vector subspace of \mathbb{R}^2 . As we originally found a 2D subspace of 3D, we have found a 1D subspace of 2D. Instead of a 3D space or a 2D *plane*, a 1D vector space like this is called a *line*. In fact, we can identify this subspace as the real number line \mathbb{R} .

The next step could be to set $x = 0$ as well. Once we’ve set both $x = 0$ and $y = 0$ to zero, there’s only one point remaining: the zero vector. This is a vector subspace as well! No

matter how you take linear combinations of the zero vector, the result is the zero vector. This is a *zero-dimensional subspace* of the 1D line, the 2D plane, and 3D space. Geometrically, a zero-dimensional subspace is a point, and that point has to be zero. If it were some other point, v , it would also contain $0 \cdot v = 0$, and an infinity of other different scalar multiples like $3 \cdot v$ and $-42 \cdot v$. Let's run with this idea.

6.3.2 Starting with a single vector

A vector subspace containing a non-zero vector v contains (at least) all of the scalar multiples of v . Geometrically, the set of all scalar multiples of a non-zero vector v lie on a line through the origin.

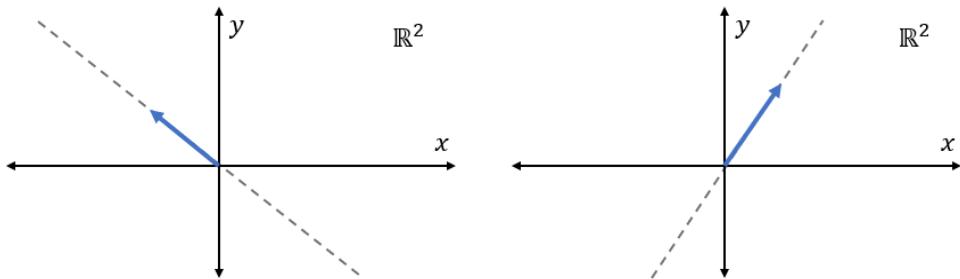


Figure 6.19 Two different vectors with dotted lines showing where all of their scalar multiples will lie.

Each of these lines through the origin is a vector space. There's no way to escape any line like this by adding or scaling vectors that lie in it. This is true of lines through the origin in 3D as well: they are all of the linear combinations of a single 3D vector, and they form a vector space. This is the first example of a general way of building subspaces: picking a vector and seeing all of the linear combinations that must come with it.

6.3.3 Spanning a bigger space

Given a set of one or more vectors, their *span* is defined as the set of all linear combinations. The important part of the span is that it's automatically a vector subspace. To rephrase what we just discovered, the span of a single vector v is a line through the origin. We denote a set of objects by including them in curly braces, so the set containing only v is $\{v\}$ and the span of this set could be written $\text{span}(\{v\})$.

As soon as we include another vector w which is not parallel to v , the space gets bigger because we are no longer constrained to a single, linear direction. The span of the set of two vectors $\{v, w\}$ includes two lines, $\text{span}(\{v\})$ and $\text{span}(\{w\})$, as well as linear combinations including both v and w , which lie on neither line.

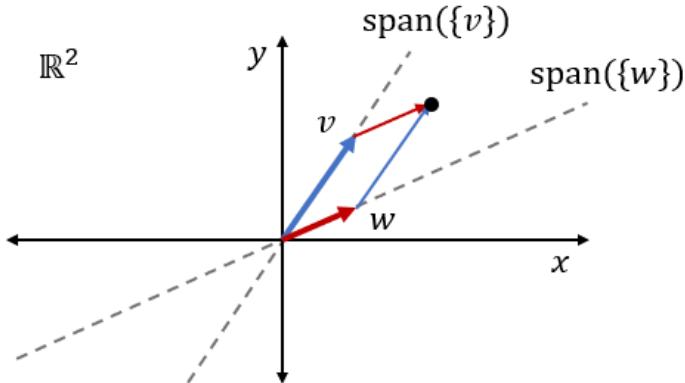


Figure 6.20 The span of two non-parallel vectors. Each individual vector spans a line, but together they span more points. For instance, $v + w$ lies on neither line.

It may not be obvious, but the span of these two vectors is the entire plane. This is true of any pair of non-parallel vectors in the plane, but most strikingly for the standard basis vectors. Any point (x,y) can be reached as the linear combination $x \cdot (1,0) + y \cdot (0,1)$. The same is true for other pairs of non-parallel vectors, like $v = (1,0)$ and $w = (1,1)$, but there's a bit more arithmetic to see it. You can get any point, like $(4,3)$ by taking the right linear combination of $(1,0)$ and $(1,1)$. The only way to get the y -coordinate of three is to have three of the vector $(1,1)$. That's $(3,3)$ instead of $(4,3)$, so you can correct the x -coordinate by adding one unit of $(1,0)$. That gets us a linear combination $3 \cdot (1,1) + 1 \cdot (1,0)$ which takes us to the point $(4,3)$.

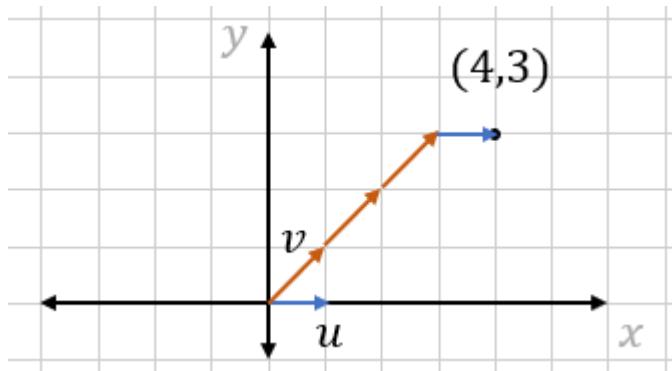


Figure 6.21 Getting to an arbitrary point $(3,4)$ by a linear combination of $(1,0)$ and $(1,1)$.

A single non-zero vector spans a line in 2D or 3D, and it turns out two non-parallel vectors can span either the whole 2D plane, or a plane subspace within 3D space. A plane spanned by two 3D vectors could look like this:

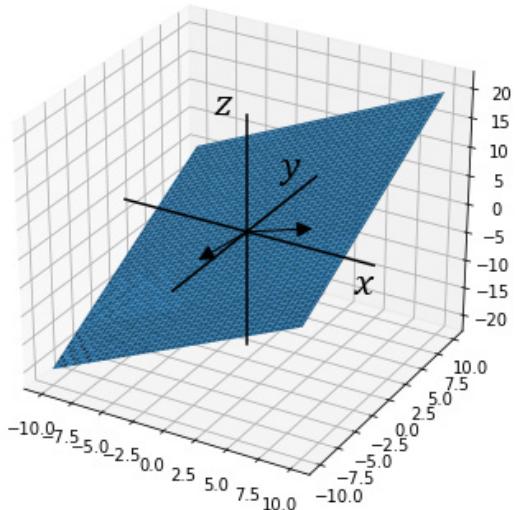


Figure 6.22 A plane spanned by two 3D vectors.

It's slanted, so it doesn't look like the plane where $z = 0$, and it doesn't contain any of the three standard basis vectors. But it's still a plane, and a vector subspace of 3D space.

One vector spans a 1D space, and two non-parallel vectors span a 2D space; if we add a third non-parallel vector to the mix, do the three vectors span a 3D space? Clearly the answer is no:

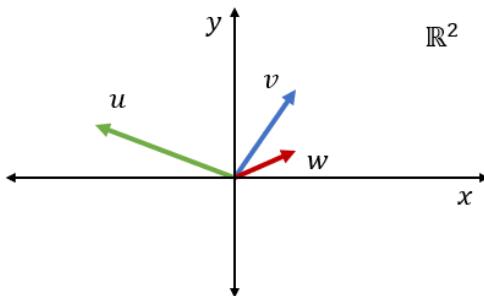


Figure 6.23 Three non-parallel vectors which only span a 2D space.

No pair of the vectors u , v , and w is parallel, but these vectors don't span a 3D space. They all live in the 2D plane, so no linear combination of them can magically obtain a z -coordinate. We need a better generalization of the concept of "non-parallel" vectors. If we want to add a vector to a set and span a higher dimensional space, the new vector needs to point in a "new"

direction which isn't included in the span of the existing ones. In the plane, three vectors will always have some redundancy. For instance, a linear combination of u and w gives us v .

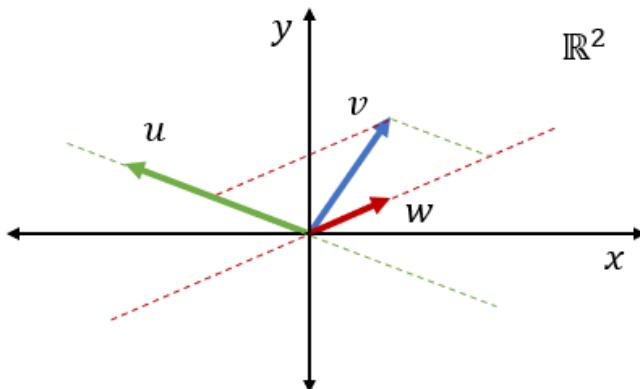


Figure 6.24 A linear combination of u and w gets us v , so the span of u , v , and w should be no bigger than the span of u and w .

The right generalization of “non-parallel” is *linear independence*. A collection of vectors is *linearly dependent* if any of its members can be obtained as a linear combination of the others. Two parallel vectors are linearly dependent, because they are scalar multiples of each other. Likewise the set of three vectors $\{u, v, w\}$ is linearly dependent because we can make v out of a linear combination of u and w (or w out of a linear combination of u and v , and so on). You should make sure to get a feel for this concept yourself: as one of the exercises at the end of the section, you can check that any of the three vectors $(1,0)$, $(1,1)$ and $(-1,1)$ can be written as a linear combination of the other two.

By contrast, the set $\{u, v\}$ is *linearly independent*: because they are non-parallel they can't be scalar multiples of one another. This means u and v span a bigger space than either on its own. Similarly, the standard basis $\{e_1, e_2, e_3\}$ for \mathbb{R}^3 is a linearly independent set. None of these vectors can be built as a linear combination of the other two, and all three are required to span 3D space. We're starting to get at the properties of a vector space or subspace that indicate its dimension.

6.3.4 Defining the word “dimension”

Here's a motivational question: is the following set of 3D vectors linearly independent?

$$\{(1,1,1), (2,0,-3), (0,0,1), (-1, -2, 0)\}$$

To answer this, you could draw these vectors in 3D, or attempt to find a linear combination of three of them to get the fourth. But there's an easier answer -- only three vectors are needed to span all of 3D space, so any list of four 3D vectors has to have some redundancy. We know

that a set with one or two 3D vectors will span a line or plane, respectively, rather than all of \mathbb{R}^3 . Three is the magic number of vectors that can both span a three-dimensional space and still be linearly independent. That's really *why* we call it three-dimensional: there are three independent directions after all.

A linearly independent set of vectors which spans a whole vector space, like $\{e_1, e_2, e_3\}$ for \mathbb{R}^3 , is called a *basis*. Any basis for a space will have the same number of vectors, and that number is its *dimension*. For instance, we saw $(1,0)$ and $(1,1)$ are linearly independent and span the whole plane, so they are a basis for the vector space \mathbb{R}^2 . Likewise $(1,0,0)$ and $(0,1,0)$ are linearly independent and span the plane where $z = 0$ in \mathbb{R}^3 . That makes them a basis for this 2D subspace, albeit not a basis for all of \mathbb{R}^3 .

I have already been using the word “basis” in the context of the “standard basis” for \mathbb{R}^2 and for \mathbb{R}^3 . These are called “standard” because they are such natural choices. It takes no computation to decompose a coordinate vector in the standard basis; the coordinates are the scalars in this decomposition. For instance $(3,2)$ means the linear combination $3 \cdot (1,0) + 2 \cdot (0,1)$ or $3e_1 + 2e_2$.

In general, deciding whether vectors are linearly independent requires some work. Even if you know that a vector is a linear combination of some other vectors, finding that linear combination requires doing some algebra. In the next chapter we'll cover how to do that -- it ends up being a ubiquitous computational problem in linear algebra. Before that, let's get some more practice identifying subspaces and measuring their dimensions.

6.3.5 Finding subspaces of the vector space of functions

Mathematical functions from \mathbb{R} to \mathbb{R} contain an infinite amount of data, namely the output value when they are given any of infinitely many real numbers as inputs. That doesn't mean that it takes infinite data to describe a function though. For instance, a *linear function* requires only two real numbers to specify. They are the values of a and b in this general formula you've probably seen:

$$f(x) = ax + b$$

where a and b can be any real number. This is much more tractable than the infinite-dimensional space of all functions. Any linear function can be specified by two real numbers, so it looks like the subspace of linear functions will be two-dimensional.

CAUTION

I've used the word “linear” in a lot of new contexts in the last few chapters. Here, I'm returning to a meaning you used in high school algebra: a linear function is a function whose graph is a straight line. Unfortunately functions of this form are not linear in the sense we spent all of chapter 4 discussing, and you can prove it yourself in an exercise. I'll try to be clear as to which sense of the word “linear” I'm using at any point.

We can quickly implement a `LinearFunction` class inheriting from `Vector`. Instead of holding a function as its underlying data, it can hold two numbers for the coefficients a and b . We can add these functions by adding coefficients, because

$$(ax + b) + (cx + d) = (ax + cx) + (b + d) = (a+c)x + (b+d)$$

And we can scale the function by multiplying both coefficients by the scalar: $r(ax + b) = rax + rb$. Finally, it turns out the zero function $f(x) = 0$ is linear -- it's the case where $a = b = 0$. Here's the implementation.

```
class LinearFunction(Vector):
    def __init__(self,a,b):
        self.a = a
        self.b = b
    def add(self,v):
        return LinearFunction(self.a + v.a, self.b + v.b)
    def scale(self,scalar):
        return LinearFunction(scalar * self.a, scalar * self.b)
    def __call__(self,x):
        return self.a * x + self.b
    @classmethod
    def zero(cls):
        return LinearFunction(0,0,0)
```

The result is a linear function alright: `plot([LinearFunction(-2,2)], -5, 5)` shows us the straight line graph of $f(x) = -2x + 2$.

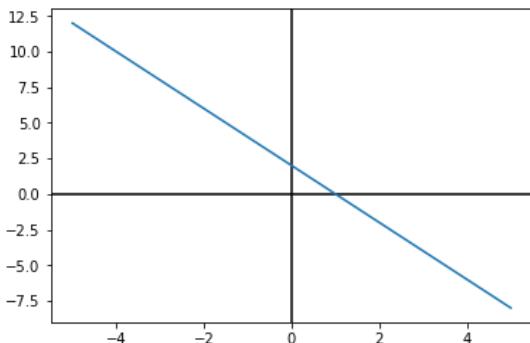


Figure 6.25 The graph of `LinearFunction(-2,2)` representing $f(x) = -2x + 2$.

We can prove to ourselves that linear functions form a vector subspace of dimension 2 by writing down a basis. The basis vectors should both be functions, they should span the whole space of linear functions, and they should be linearly independent (not multiples of one another). Such a set is $\{x, 1\}$, or more specifically $\{f(x) = x, g(x) = 1\}$. Named this way, functions of the form $ax + b$ can be written as a linear combination $a \cdot f + b \cdot g$.

This is as close as we can get to a “standard basis” for linear functions: $f(x) = x$ and $f(x) = 1$ are clearly different functions, not scalar multiples of one another. By contrast $f(x) = x$ and

$h(x) = 4x$ are scalar multiples of one another, and would not be a linearly independent pair. But $\{x, 1\}$ is not the only basis we could have chosen: $\{4x+1, x-3\}$ is also a basis.

The same concept applies to *quadratic functions*, having the form $f(x) = ax^2 + bx + c$. These form a three-dimensional subspace of the vector space of functions, with one choice of basis being $\{x^2, x, 1\}$. Linear functions form a vector subspace of the space of quadratic functions, where the “ x^2 component” is zero. Linear functions and quadratic functions are examples of *polynomial functions*, which are linear combinations of powers of x :

$$f(x) = a_1 + a_1x + a_2x^2 + \dots + a_nx^n$$

Linear and quadratic functions have *degree* 1 and 2 respectively, because those are the highest powers of x that appear in each. The polynomial written above has degree n , and has $n+1$ coefficients in total. In the exercises, you’ll see that the space of polynomials of *any* degree forms another vector subspace of the space of functions.

6.3.6 Subspaces of images

Since our `ImageVector` objects are represented by 270,000 numbers, we could follow the “standard basis” recipe and construct a basis of 270,000 images, each with one of the 270,000 numbers equal to 1 and all others equal to zero. Here’s what the first basis vector would look like:

Listing: Pseudocode showing how we would build a first standard basis vector.

```
ImageVector([
    (1,0,0), (0,0,0), (0,0,0), ..., (0,0,0), #①
    (0,0,0), (0,0,0), (0,0,0), ..., (0,0,0), #②
    ... #③
])
```

- ① Only the first pixel in the first row is non-zero: it has a red value of 1. All of the other pixels have value (0,0,0)
- ② The second row consists of 300 black pixels, each with value (0,0,0).
- ③ I skipped the next 298 rows, but they are all identical to row 2; no pixels have any color values.

This single vector spans a one-dimensional subspace, consisting of the images which are black except for a single, red pixel in the top left corner. Scalar multiples of this image could have brighter or dimmer red pixels at this location, but no other pixels could be illuminated. In order to show more pixels, we need more basis vectors.

There’s not too much to be learned from writing out these 270,000 basis vectors. Let’s instead look for a small set of vectors that span an interesting subspace. Here’s a single `ImageVector` consisting of dark gray pixels at every position.

```
gray = ImageVector([
    (1,1,1), (1,1,1), (1,1,1), ..., (1,1,1),
    (1,1,1), (1,1,1), (1,1,1), ..., (1,1,1),
    ...
])
```

More concisely, we could write

```
gray = ImageVector([(1,1,1) for _ in range(0,300*300)])
```

One way to picture the subspace spanned by the single vector gray is to look at some vectors that belong to it. These are all scalar multiples of gray:

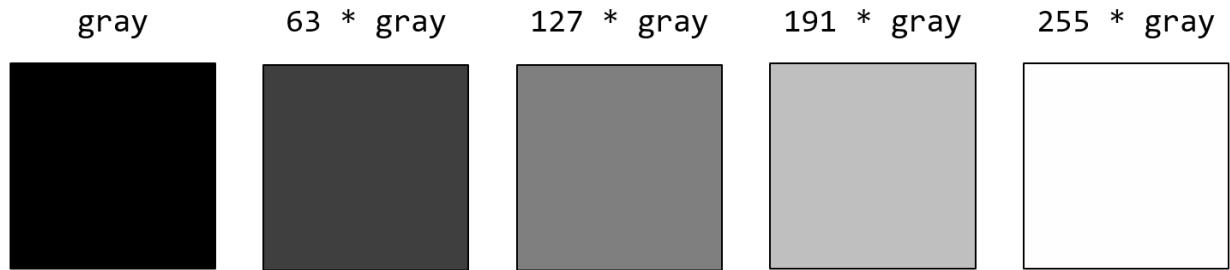


Figure 6.26 Some of the vectors in the one-dimensional subspace of images spanned by the `gray` `ImageVector`.

This collection of images is “one-dimensional” in the colloquial sense. There’s only one thing changing about them, which is their brightness. Another way we can look at this subspace is by thinking about the pixel values. In this subspace, any image has the same value at each pixel. For any given pixel, there is a 3D space of color possibilities, measured by red, green, and blue coordinates. Gray pixels form a 1D subspace of this, containing points with all coordinates $s * (1,1,1)$ for some scalar s .

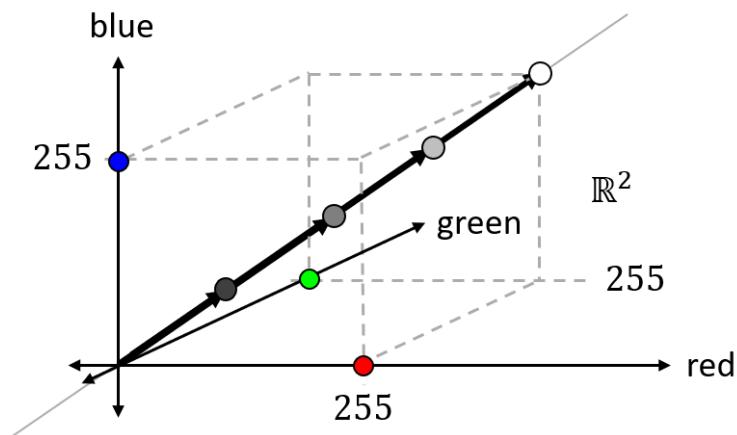


Figure 6.27 Gray pixels of varying brightness on a line. The gray pixels form a 1D subspace of the 3D vector space of pixel values.

Each of the images in the basis would be black, except for one pixel which would be a very dim red, green, or blue. Changing one pixel at a time doesn't yield very striking results, so let's look for smaller and more interesting subspaces.

There are many subspaces of images you can explore. You could look at solid color images of any color. These would be images of the form:

```
ImageVector([
    (r,g,b), (r,g,b), (r,g,b), ... , (r,g,b),
    (r,g,b), (r,g,b), (r,g,b), ... , (r,g,b),
    ...
])
```

There are no constraints on the pixels themselves; the only constraint on "solid color" images is that every pixel is the same.

As a final example, you could consider a subspace consisting of low-resolution grayscale images like the following:

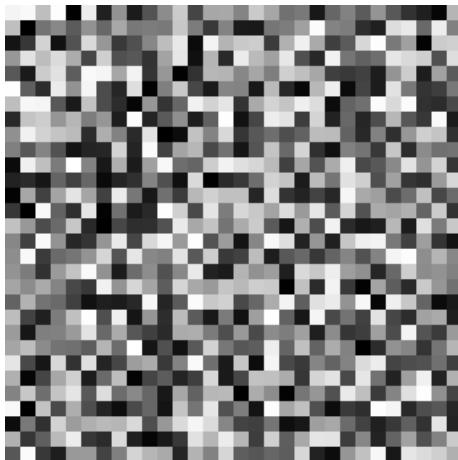


Figure 6.28 A low-resolution grayscale image. Each 10x10 block of pixels has the same value.

Each 10 pixel by 10 pixel block has a constant gray value across its pixels, making it look like a 30x30 grid. There are only $30 \cdot 30 = 900$ numbers defining this image, so images like this one define a 900-dimensional subspace of the 270,000 dimensional space of images. It's a lot less data, but it's still possible to create recognizable images.

One way to make an image in this subspace is to start with any image and average all red, green, and blue values in each 10 pixel by 10 pixel block. This average gives you the brightness b , and you can set all pixels in the block to (b,b,b) to build your new image. This turns out to be a linear map, and you can implement it as a mini-project.

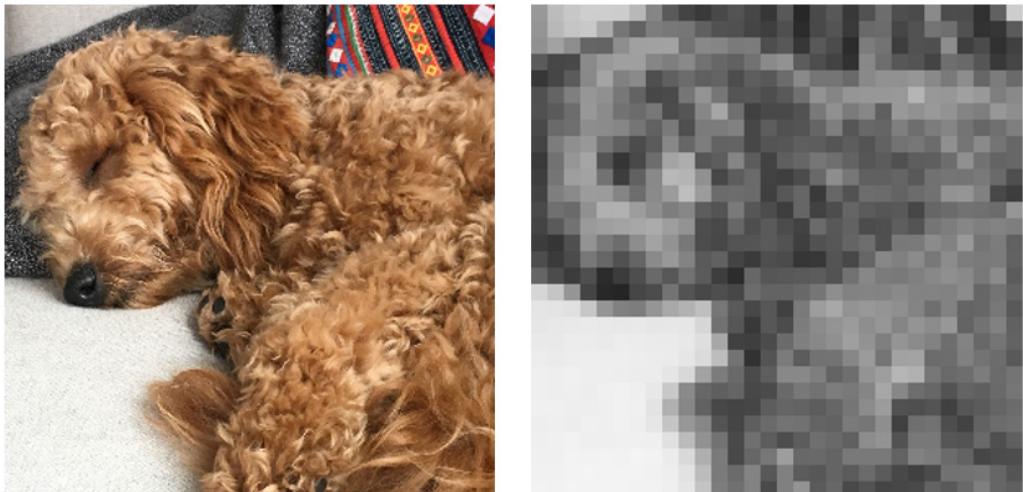


Figure 6.29 A linear map takes any image (left) and returns a new one (right) lying in a 900-dimensional subspace.

My dog, Melba, isn't as photogenic in the second picture, but the picture is still recognizable. This is the example I mentioned at the beginning of the section, and the remarkable thing is that you can tell it's the same picture with only 0.3% of the data. There's clearly room for improvement, but the approach of mapping to a subspace is a starting point for more fruitful exploration. In Chapter 10, we'll see how to compress other kinds of data in this way, and with even better fidelity.

6.3.7 Exercises

EXERCISE

Give a geometric argument for why the following region S of the plane can't be a vector subspace of the plane.

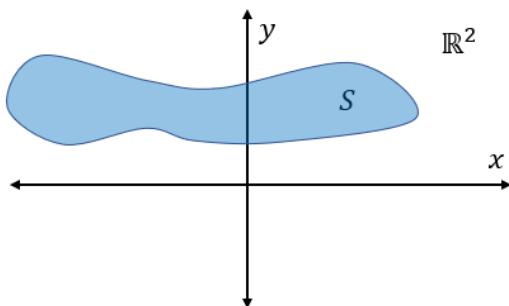


Figure 6.30 Why isn't this subset of the plane a vector subspace of the plane?

SOLUTION

There are many linear combinations of points in this region which don't end up in the region. More obviously, this region cannot be a vector space because it doesn't include the zero vector. The zero vector is a scalar multiple of any vector (by the scalar zero), so it must be included in any vector space or subspace.

EXERCISE

Show that the region of the plane where $x = 0$ forms a 1D vector space.

SOLUTION

These are the vectors that lie on the y-axis, and have the form $(0,y)$ for a real number y . Addition and scalar multiplication of vectors of the form $(0,y)$ is the same as for real numbers; there just happens to be an extra "0" along for the ride. We can conclude that this is \mathbb{R} in disguise, and therefore a 1D vector space. If you want to be more rigorous, you can check all of the vector space properties explicitly.

EXERCISE

Show that any of the three vectors $(1,0)$, $(1,1)$, and $(-1,1)$ are linearly dependent by writing each one as a linear combination of the other two.

SOLUTION:

$$\begin{aligned}(1,0) &= \frac{1}{2} \cdot (1,1) - \frac{1}{2} \cdot (-1,1) \\ (1,1) &= 2 \cdot (1,0) + (-1,1) \\ (-1,1) &= (1,1) - 2 \cdot (1,0)\end{aligned}$$

EXERCISE

Show that you can get any vector (x,y) as a linear combination of $(1,0)$ and $(1,1)$.

SOLUTION

We know that $(1,0)$ can't contribute to the y-coordinate, so we need y times $(1,1)$ as part of the linear combination. To make the algebra work, we need $(x-y)$ units of $(1,0)$:

$$(x,y) = (x-y) \cdot (1,0) + y(1,1).$$

EXERCISE

Given a single vector v , explain why "all linear combinations of v " is the same thing as "all scalar multiples of v "

SOLUTION

Linear combinations of a vector and itself reduce to scalar multiples, according to one of the vector space laws. For instance, the linear combination $a \cdot v + b \cdot v$ is equal to $(a+b) \cdot v$.

EXERCISE

From a geometric perspective, explain why a line that doesn't pass through the origin will not be a vector subspace (of the plane or of 3D space).

SOLUTION

One simple reason this cannot be a subspace is that it doesn't contain the origin (the zero vector). Another reason is that such a line will have two non-parallel vectors. Their span would be the whole plane, which is much bigger than the line.

EXERCISE

Any two of $\{e_1, e_2, e_3\}$ will fail to span all of \mathbb{R}^3 , and will instead span 2D subspaces of 3D space. What are these subspaces?

SOLUTION

The span of the set $\{e_1, e_2\}$ consists of all linear combinations $a e_1 + b e_2$, or $a(1,0,0) + b(0,1,0) = (a,b,0)$. Depending on the choice of a and b , this can be any point in the plane where $z = 0$, often called the x,y -plane. By the same argument the vectors $\{e_2, e_3\}$ span the plane where $x = 0$, called the y,z -plane, and the vectors $\{e_1, e_3\}$ span the plane where $y = 0$, called the x,z -plane.

EXERCISE

Write the vector $(-5, 4)$ as a linear combination of $(0,3)$ and $(-2,1)$.

SOLUTION

Only $(-2,1)$ can contribute to the x -coordinate, so we need to have $2.5 \cdot (-2,1)$ in the sum. That gets us to $(-5, 2.5)$, so we need an additional 1.5 units on the x -coordinate, or $0.5 \cdot (0,3)$. The linear combination is:

$$(-5, 4) = 0.5 \cdot (0,3) + 2.5 \cdot (-2,1).$$

MINI-PROJECT

Are $(1,2,0)$, $(5,0,5)$ and $(2,-6,5)$ linearly independent or linearly dependent vectors?

SOLUTION

It's not easy to find, but there is a linear combination of the first two vectors that yields the third:

$$-3 \cdot (1,2,0) + (5,0,5) = (2,-6,5).$$

This means that the third vector is redundant, and the vectors are linearly dependent. They only span a 2D subspace of 3D rather than all of 3D space.

EXERCISE

Explain why the “linear function” $f(x) = ax + b$ is not a linear map from the vector space \mathbb{R} to itself unless $b = 0$.

SOLUTION

We can turn directly to the definition: a linear map must preserve linear combinations. We see f doesn't preserve linear combinations of real numbers. For instance $f(1+1) = 2a + b$ while $f(1) + f(1) = (a + b) + (a + b) = 2a + 2b$. This won't hold unless $b = 0$.

As an alternative explanation, we know that linear functions $\mathbb{R} \rightarrow \mathbb{R}$ should be representable as 1-by-1 matrices. Matrix multiplication of a 1D column vector $[x]$ by a 1-by-1 matrix $[a]$ gives you $[ax]$. This is an unusual case of matrix multiplication, but your implementation from chapter 5 will confirm this result. If a function $\mathbb{R} \rightarrow \mathbb{R}$ is going to be linear, it must agree with 1-by-1 matrix multiplication, and therefore be multiplication by a scalar.

EXERCISE

Rebuild the `LinearFunction` class by inheriting from `Vec2` and simply implementing the `__call__` method.

SOLUTION

The data of a `Vec2` are called "x" and "y" instead of "a" and "b", but otherwise the functionality is the same. All we need to do is implement `__call__`:

```
class LinearFunction(Vec2):
    def __call__(self, input):
        return self.x * input + self.y
```

EXERCISE

Prove (algebraically!) that linear functions $f(x) = ax + b$ form a vector subspace of the vector space of functions.

SOLUTION

To prove this, we need to be sure a linear combination of two linear functions is another linear function. If $f(x) = ax + b$ and $g(x) = cx + d$, then $rf + sg$ gives us:

$$r \cdot f + s \cdot g = r \cdot (ax+b) + s \cdot (cx+d) = rax + b + scx + d = (ra + sc)x + (b+d)$$

Since $(ra + sc)$ and $(b+d)$ are scalars, this has the form we want. We can conclude that linear functions are closed under linear combinations, and therefore that they form a subspace.

EXERCISE:

Find a basis for the set of 3-by-3 matrices. What is the dimension of this vector space?

SOLUTION

Here's a basis consisting of 9 3-by-3 matrices.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 6.31 A basis for the vector space of 3-by-3 matrices.

They are linearly independent: each contributes a unique entry to any linear combination. They also span the space, since any matrix can be constructed as a linear combination of these; the coefficient on any particular matrix decides one entry of the result. Because these nine vectors provide a basis for the space of 3-by-3 matrices, the space has dimension nine.

MINI-PROJECT

Implement a class `QuadraticFunction(Vector)` representing the vector subspace of functions of the form $ax^2 + bx + c$. What is a basis for this subspace?

SOLUTION

The implementation looks a lot like `LinearFunction`, except there are three coefficients instead of two, and the `__call__` function has a square term.

```
class QuadraticFunction(Vector):
    def __init__(self,a,b,c):
        self.a = a
        self.b = b
        self.c = c
    def add(self,v):
        return QuadraticFunction(self.a + v.a, self.b + v.b, self.c + v.c)
    def scale(self,scalar):
        return QuadraticFunction(scalar * self.a, scalar * self.b, scalar *
self.c)
    def __call__(self,x):
        return self.a * x * x + self.b * x + self.c
    @classmethod
    def zero(cls):
        return QuadraticFunction(0,0,0)
```

We can take note that $ax^2 + bx + c$ looks like a linear combination of the set $\{x^2, x, 1\}$. Indeed, these three functions span the space and none of these three can be written as a linear combination of the others. There's no way to get a x^2 term by adding together linear functions, for example. Therefore, this is a basis. Because there are three vectors, we can conclude that this is a 3D subspace of the space of functions.

MINI-PROJECT

I claimed that $\{4x+1, x-2\}$ are a basis for the set of linear functions. Show that you can write $-2x + 5$ as a linear combination of these two functions.

SOLUTION

$(1/9) \cdot (4x + 1) - (22/9) \cdot (x - 2) = -2x + 5$. If your algebra skills aren't too rusty, you can figure this out by hand. Otherwise, don't worry – we'll cover how to solve tricky problems like this in the next chapter.

MINI-PROJECT

Vector space of all polynomials is an infinite-dimensional subspace – implement it and describe a basis (which will need to be an infinite set!).

SOLUTION:

```
class Polynomial(Vector):
    def __init__(self, *coefficients):
        self.coefficients = coefficients
    def __call__(self,x):
        return sum(coefficient * x ** power for (power,coefficient) in
enumerate(self.coefficients))
    def add(self,p):
        return Polynomial([a + b for a,b in zip(self.coefficients,
p.coefficients)])
    def scale(self,scalar):
        return Polynomial([scalar * a for a in self.coefficients])
        return "$ %s $" % (" + ".join(monomials))
    @classmethod
    def zero(cls):
        return Polynomial(0)
```

A basis for the set of all polynomials is the infinite set $\{1, x, x^2, x^3, x^4, \dots\}$. Given all of the possible powers of x at your disposal, you can build any polynomial as a linear combination.

EXERCISE

I showed you pseudocode for a basis vector for the 270,000 dimensional space of images. What would the second basis vector look like?

SOLUTION

The second vector could be given by putting a one in the next possible place. It would yield a dim green pixel in the very top left of the image.

```
ImageVector([
    (0,1,0), (0,0,0), (0,0,0), ..., (0,0,0), #①
    (0,0,0), (0,0,0), (0,0,0), ..., (0,0,0), #②
    ...
])
```

① For the second basis vector, the 1 has moved to the second possible slot.

- ② All other rows remain empty.

EXERCISE

Write a function `solid_color(r,g,b)` that returns an solid color `ImageVector` with the given red, green, and blue content at every pixel.

SOLUTION:

```
def solid_color(r,g,b):
    return ImageVector([(r,g,b) for _ in range(0,300*300)])
```

MINI-PROJECT

Write a linear map that generates an `ImageVector` from a 30 by 30 grayscale image, implemented as a 30 by 30 matrix of brightness values. Then, implement the linear map that takes a 300 by 300 image to a 30 by 30 grayscale image by averaging the brightness (average of red, green and blue) at each pixel.

SOLUTION:

```
image_size = (300,300)
total_pixels = image_size[0] * image_size[1]
square_count = 30 #①
square_width = 10

def ij(n):
    return (n // image_size[0], n % image_size[1])

def to_lowres_grayscale(img): #②

    matrix = [
        [0 for i in range(0,square_count)]
        for j in range(0,square_count)
    ]
    for (n,p) in enumerate(img.pixels):
        i,j = ij(n)
        weight = 1.0 / (3 * square_width * square_width)
        matrix[i // square_width][j // square_width] += (sum(p) * weight)
    return matrix

def from_lowres_grayscale(matrix): #③
    def lowres(pixels, ij):
        i,j = ij
        return pixels[i // square_width][j // square_width]
    def make_highres(limg):
        pixels = list(matrix)
        triple = lambda x: (x,x,x)
        return ImageVector([triple(lowres(matrix, ij(n))) for n in
range(0,total_pixels)])
    return make_highres(matrix)
```

① This constant indicates that we're breaking the picture into a 30 by 30 grid.

② The function takes an `ImageVector`, and returns an array of 30 arrays of 30 values each, giving grayscale values square by square.

- ③ The second function takes a 30 by 30 matrix and returns an image built out of 10 pixel by 10 pixel blocks, having brightnesses given by the matrix values.

Calling `from_lowres_grayscale(to_lowres_grayscale(img))` transforms the image `img` in the way I showed in the chapter.

6.4 Summary

In this chapter you learned:

- A *vector space* is a generalization of the 2D plane and 3D space: a collection of objects that can be added and multiplied by scalars in suitable ways.
- We can generalize in Python by pulling common features of different data types into an abstract base class and inheriting from it.
- We can overload arithmetic operators in Python so that vector math looks the same in code, regardless of what kind of vectors we're using.
- Addition and scalar multiplication need to behave in certain ways to match our intuition, and we can verify these behaviors by writing unit tests involving random vectors.
- Real-world objects like used cars can be described by several numbers (coordinates), and therefore treated as vectors. This lets us think about abstract concepts like a "weighted average of two cars."
- Functions can be thought of as vectors; we add or multiply them by adding or multiplying the expressions that define them.
- Matrices can be thought of as vectors: the entries of an $m \times n$ matrix can be thought of as coordinates of an $(m \cdot n)$ -dimensional vector. Adding or scalar-multiplying matrices has the same effect as adding or scalar-multiplying the linear functions they define.
- Images of a fixed height and width make up a vector space. They are defined by a red, green, and blue value at each pixel, so the number of coordinates and therefore the dimension of the space is defined by three times the number of pixels.
- A subspace of a vector space is a subset of the vectors in a vector space, which is a vector space on its own. That is, linear combinations of vectors in the subspace stay in the subspace.
- For any line through the origin in 2D or 3D, the set vectors that lie on it form a 1D subspace. For any plane through the origin in 3D, the vectors that lie on it form a 2D subspace.
- The *span* of a set of vectors is the collection of all linear combinations of the vectors. It is guaranteed to be a subspace of whatever space the vectors live in.
- A set of vectors is *linearly independent* if you can't make any one of them as a linear combination of the others. Otherwise the set is *linearly dependent*. A set of linearly independent vectors that span a vector space (or subspace) is called a *basis* for that space. For a given space, any basis will have the same number of vectors. That number defines the *dimension* of the space.

- When you can think of your data as living in a vector space, subspaces often consist of data with similar properties. For instance, the subset of image vectors which are solid colors forms a subspace.

7

Solving Systems of Linear Equations

This chapter covers

- Detecting collisions of objects in a 2D video game
- Writing equations to represent lines and finding where lines intersect in the plane
- Picturing and solving systems of linear equations in 3D or beyond
- Re-writing vectors as linear combinations of other vectors

When you think of algebra, you probably think of questions that require “solving for x .” For instance, you probably spent quite a bit of time in algebra class learning to solve equations like “ $3x^2 + 2x + 4 = 0$,” that is, figuring out what value or values of x make the equation true.

Linear algebra, being a branch of algebra, has the same kinds of computational questions come up. The difference is that what you’re solving for can be a vector or matrix rather than a number. If you were taking a traditional linear algebra course, you’d have to memorize a lot of algorithms to solve these kinds of problems. But because you have Python at your disposal, you only need to know how to recognize the problem you’re facing and choose the right library to find the answer for you.

I’m going to cover the most important class of linear algebra problems you’ll see in the wild: *systems of linear equations*. These problems boil down to finding points where lines, planes, or their higher dimensional analogies intersect. One example is the infamous high school math problem involving two trains leaving Boston and New York at different times and speeds. Since I don’t assume railroad operation is interesting to you, I’ll use a more familiar example.

In this chapter we'll build a clone of the classic "Asteroids" arcade game. In this game, the player controls a triangle representing a spaceship and can fire a laser at polygons floating around it, which represent asteroids. The player must destroy asteroids to prevent them from hitting and destroying the spaceship.

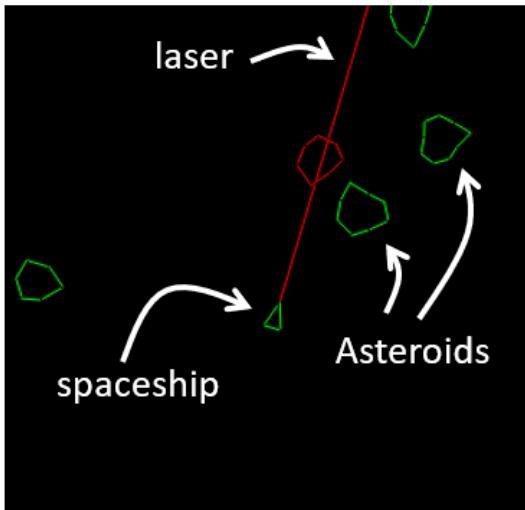


Figure 7.1 Setup of the classic "Asteroids" arcade game.

One of the key mechanics in this game is deciding whether the laser hits an asteroid. This requires us to figure out whether the line defining the laser beam intersects with the line segments outlining the asteroids. If these lines intersect, the asteroid is destroyed. Let's set up the game first and then we'll see how to solve the underlying linear algebra problems.

After that, I'll show you how this 2D example generalizes to 3D or any number of dimensions. The latter half of this chapter covers a bit more theory, but it will round out your linear algebra education. We'll have covered all of the major concepts you'd find in a college-level linear algebra class, albeit in less depth. After completing this chapter, you should be well prepared to crack open a denser textbook on linear algebra and fill in the details. But for now, let's focus on building our game.

7.1 Designing an arcade game

Let's not try to build a version of this game that's ready to ship to arcades everywhere. Rather, we can spend our time talking about the game mechanics and the math behind them. That said, I'll focus on a simplified version of the game where the ship and asteroids are static. In the source code, you'll see that I already made the asteroids move, and we'll cover how to make them move according to the laws of physics in Part 2 of the book. To get started, we'll

model the entities of the game -- the spaceship, laser, and asteroids -- and show how to render them to the screen.

7.1.1 Modeling the game

The spaceship and the asteroids will be displayed as polygons in the game. As we've done before, we'll model these as collections of vectors. For instance, we can represent an eight-sided asteroid by eight vectors (indicated by arrows) and we connect them to draw its outline.

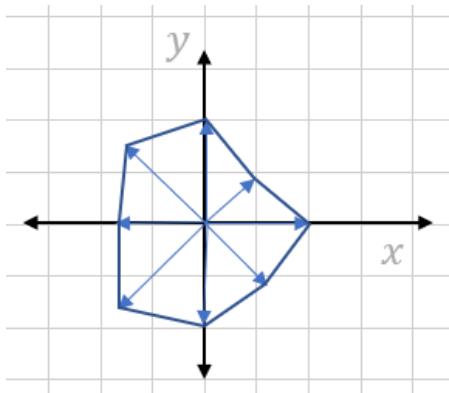


Figure 7.2 An eight-sided polygon representing an asteroid.

The asteroid or spaceship may translate or rotate as it travels through space, but its shape will remain the same. Therefore, we'll store the vectors representing this shape separately from the x,y-coordinates of its center, which could change over time. We'll also store an angle, indicating the rotation of the object at the current moment. The `PolygonModel` class will represent a game entity (the ship or an asteroid) that keeps its shape but may translate or rotate. It's initialized with a set of vector points that define the outline of the asteroid, and by default its center x and y-coordinates and its angle of rotation are set to zero.

```
class PolygonModel():
    def __init__(self,points):
        self.points = points
        self.rotation_angle = 0
        self.x = 0
        self.y = 0
```

When the spaceship or asteroid moves, we'll have to apply the translation by `(self.x, self.y)` and the rotation by `self.rotation_angle` to find out its actual location. As an exercise, you can give `PolygonModel` a method to compute the actual, transformed vectors outlining it.

The spaceship and asteroids are specific cases of `PolygonModel` that initialize automatically with respective shapes. For instance, the ship has a fixed triangular shape, given by three points:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/math-for-programmers>

```
class Ship(PolygonModel):
    def __init__(self):
        super().__init__([(0.5,0), (-0.25,0.25), (-0.25,-0.25)])
```

For the asteroid, we initialize it with somewhere between five and nine vectors at equally spaced angles and random lengths between 0.5 and 1.0. This randomness gives the asteroids some character.

```
class Asteroid(PolygonModel):
    def __init__(self):
        sides = randint(5,9) #1
        vs = [vectors.to_cartesian((uniform(0.5,1.0), 2*pi*i/sides)) #2
              for i in range(0,sides)]
        super().__init__(vs)
```

- ① An asteroid has a random number of sides, between 5 and 9.
- ② Given the definition in terms of angles and lengths of vectors, it's easier to write them down in polar coordinates. Their lengths are randomly selected between 0.5 and 1.0, and the angles are multiples of $2\pi/n$, where n is the number of sides.

With these objects defined, we can turn our attention to instantiating them and rendering them to the screen.

7.1.2 Rendering the game

For the initial state of the game, we need a ship and several asteroids. The ship can begin at the center of the screen, but the asteroids should be randomly spread over the screen. We'll show an area of the plane ranging from -10 to 10 in the x and y directions.

```
ship = Ship()

asteroid_count = 10
asteroids = [Asteroid() for _ in range(0,asteroid_count)] #1

for ast in asteroids: #2
    ast.x = randint(-9,9)
    ast.y = randint(-9,9)
```

- ① Create a list of a specified number of Asteroid objects, in this case ten.
- ② For each asteroid, set its position to a random point, with coordinates between -10 and 10 so it shows up on the screen.

I'll use a 400 pixel by 400 pixel screen, which will require transforming the x and y coordinates before rendering them. Using PyGame's built-in 2D graphics (instead of OpenGL), the top left pixel on the screen has coordinate (0,0) and the bottom right has coordinate (400,400). So these coordinates are not only bigger, they're also translated and upside-down. We'll need to write a `to_pixels` function that does the transformation from our coordinate system to PyGame's pixels.

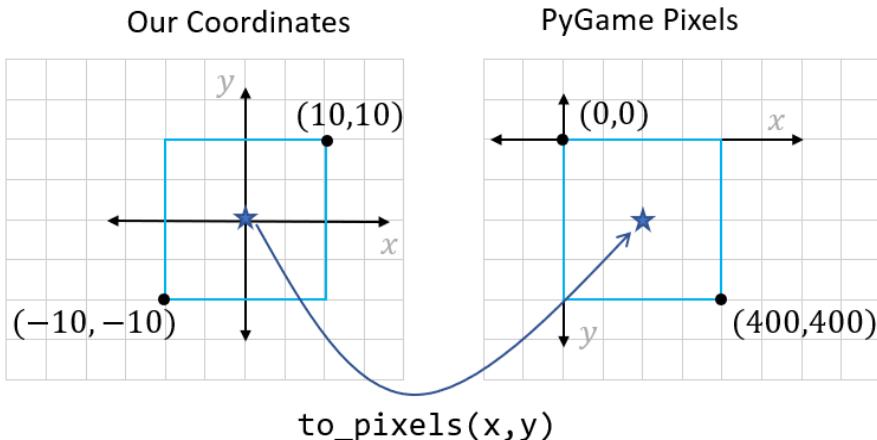


Figure 7.3 A `to_pixels` function should map an object from the center of our coordinate system to the center of the PyGame screen.

With the `to_pixels` function implemented, we can write a function to draw a polygon defined by points to the PyGame screen. First we take the transformed points (translated and rotated) which define the polygon, and convert them to pixels. Then we draw them with a PyGame function.

```
GREEN = (0, 255, 0)
def draw_poly(screen, polygon_model, color=GREEN):
    pixel_points = [to_pixels(x,y) for x,y in polygon_model.transformed()]
    pygame.draw.aalines(screen, color, True, pixel_points, 10) #❶
```

- ❶ The `pygame.draw.aalines` function draws lines connecting given points to a specified `pygame "screen"` object. The “`True`” parameter indicates that the first and last point should be connected to create a closed polygon.

You can see the whole game loop in source code, but it basically calls `draw_poly` for the ship and each asteroid every time a frame is rendered. The result is our simple triangular spaceship surrounded by an asteroid field in a PyGame window.



Figure 7.4 The game rendered in a PyGame window.

7.1.3 Shooting the laser

Now it's time for the most important part: giving our ship a way to defend itself! The player should be able to aim the ship using the left and right arrow keys and then shoot a laser by pressing the spacebar. The laser beam should come out of the tip of the spaceship and extend to the edge of the screen.

In the 2D world we've invented, the laser beam should be a line segment starting at the *transformed* tip of the spaceship, and extending in whatever direction the ship is pointed. We can make sure it reaches the end of the screen by making it sufficiently long. Since the laser's line segment is associated with the state of the `Ship` object, we can make a method on the `Ship` class to compute it.

```
class Ship(PolygonModel):
    ...
    def laser_segment(self):
        dist = 20. * sqrt(2) # ①
        x,y = self.transformed()[0] #②
        end = (x + dist * cos(self.rotation_angle), y + dist *
               sin(self.rotation_angle)) ③
        return (x,y), end
```

- ① Since our coordinate system stretches 20 units in each direction, the pythagorean theorem tells us that this is the longest segment that could fit on the screen (check it yourself!).
- ② The tip of the ship happens to be the first of the points defining the ship, we get its value after doing the transformations.

- ③ Using some trigonometry to find an endpoint for the laser, if it extends `dist` units from the tip `(x, y)` at an angle `self.rotation_angle`. This is shown in the diagram below.

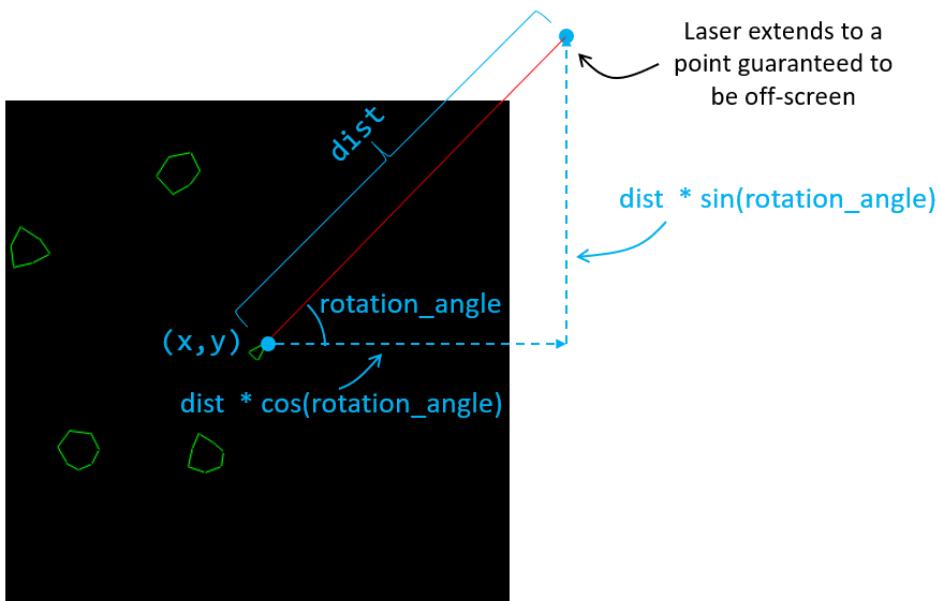


Figure 7.5 Using trigonometry to find the off-screen point where the laser beam ends.

In the source code, you can see how to make PyGame respond to key presses, and draw the laser as a line segment only if the spacebar is pressed.

Finally, if the player fires the laser and hits an asteroid, we want to know something happened. In every iteration of the game loop, we want to check each asteroid to see if it is currently hit by the laser. We'll do this with a `does_intersect(segment)` method on the `PolygonModel` class, which computes whether the input segment intersects any segment of the given `PolygonModel`. The final code will include some lines like the following:

```

laser = ship.laser_segment()           #1
keys = pygame.key.get_pressed()
if keys[pygame.K_SPACE]:              #2
    draw_segment(*laser)

    for asteroid in asteroids:
        if asteroid.does_intersect(laser): #3
            asteroids.remove(asteroid)

```

- ① Calculate the line segment representing the laser beam, based on the ship's current position and orientation.
- ② Detect which keys are pressed, and if the spacebar is pressed, render the laser beam to the screen with a helper function `draw_segment` (similar to `draw_poly`).

- ③ For every asteroid, check whether the laser line segment intersects it. If so destroy the given asteroid by removing it from the list of asteroids.

The work that remains is implementing the `does_intersect(segment)` method. Next, we'll cover the math to do so.

7.1.4 Exercises

EXERCISE

Implement a `transformed()` method on the `PolygonModel` that returns the points of the model translated by the object's `x` and `y` attributes and rotated by its `rotation_angle` attribute.

SOLUTION

Make sure to apply the rotation first, otherwise the translation vector will be rotated by the angle as well.

```
class PolygonModel():
    ...
    def transformed(self):
        rotated = [vectors.rotate2d(self.rotation_angle, v) for v in
self.points]
        return [vectors.add((self.x, self.y), v) for v in rotated]
```

EXERCISE

Write a function `to_pixels(x, y)` that takes a pair of `x` and `y` coordinates in the square where $-10 < x < 10$ and $-10 < y < 10$ and maps them to the corresponding PyGame `x` and `y` pixel coordinates which each range from 0 to 400.

SOLUTION:

```
width, height = 400, 400
def to_pixels(x,y):
    return (width/2 + width * x / 20, height/2 - height * y / 20)
```

7.2 Finding intersection points of lines

The problem at hand is to decide whether the laser beam hits the asteroid. To do this, we'll look at each line segment defining the asteroid and decide whether it intersects with the segment defining the laser beam. There are a few algorithms we could use, but we'll solve this as a *system of linear equations in two variables*. Geometrically, this means looking at the lines defined by an edge of the asteroid and the laser beam and seeing where they intersect.

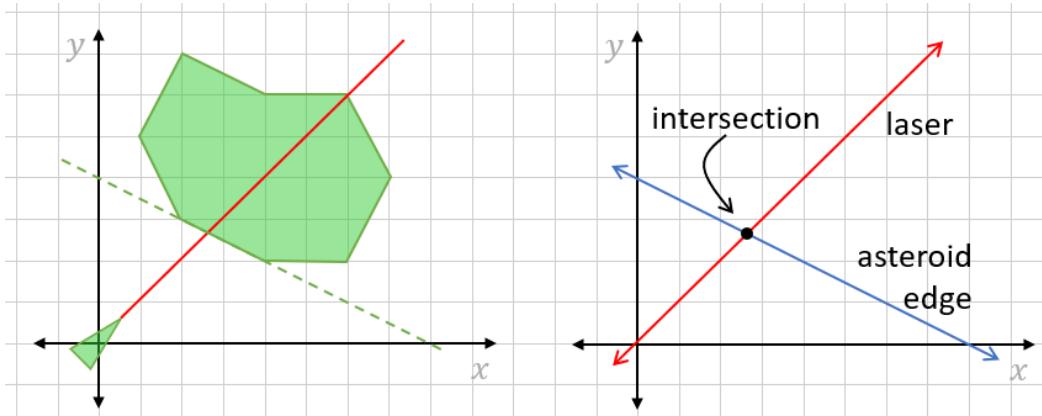


Figure 7.6 The laser hitting an edge of an asteroid (left) and the corresponding system of linear equations (right).

Once we know the location of intersection, we can see whether it lies within the bounds of both segments. If so, the segments collide and the asteroid is hit. We'll first review equations for lines in the plane, then cover how to find where pairs of lines intersect, and finally we'll code up the `does_intersect` method for our game.

7.2.1 Choosing the right formula for a line

In the previous chapter, we saw that one-dimensional subspaces of the 2D plane are lines. These subspaces consist of all of the scalar multiples $t \cdot v$ for a single chosen vector v . Because one such scalar multiple is $0 \cdot v$, these lines always pass through the origin. So $t \cdot v$ is not quite a general formula for any line we encounter.

If we start with a line through the origin and are allowed to *translate* it by another vector u , we can get any possible line. The points on this line will have the form $u + t \cdot v$ for some scalar t . For instance, take $v = (2, -1)$. Points of the form $t \cdot (2, -1)$ lie on a line through the origin. But if we translate by a second vector $u = (2, 3)$ the points are now $(2, 3) + t \cdot (2, -1)$, which constitute a line that *doesn't* pass through the origin.

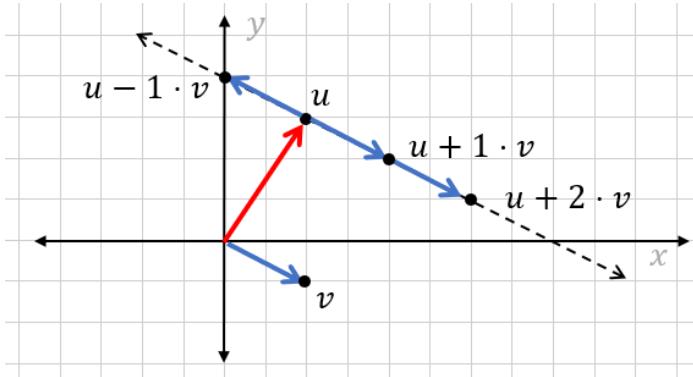


Figure 7.7 Vectors $u = (2,3)$ and $v = (2,-1)$. Points of the form $u + t \cdot v$ lie on a straight line.

Any line can be described as the points $u + t \cdot v$ for some selection of vectors u and v and *all* possible scalar multiples t .

This is probably not the general formula for a line you're used to. Instead of writing y as a function of x , we've given both the x and y coordinates of points on the line as functions of another parameter t . Sometimes you'll see the line written $r(t) = u + t \cdot v$ to indicate that this line is a vector-valued function of the scalar parameter t . The input t decides how many units of v you go from the starting point u to get the output $r(t)$.

The advantage of this kind of formula for a line is that it's dead simple to find if you have two points on the line. If your points are u and w , then you can use u as the translation vector and $w - u$ as the vector that is scaled. The formula is $r(t) = u + t \cdot (w - u)$.

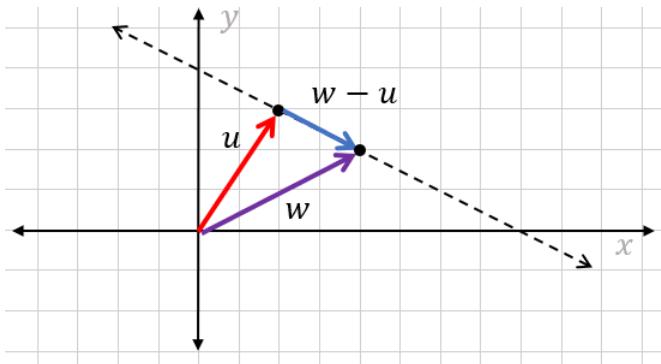


Figure 7.8 Given u and w , the line that connects them is $r(t) = u + t(w-u)$.

The formula $r(t) = u + t \cdot v$ also has its downsides. As you'll see in the exercises, there are multiple ways to write the same line in this form. The extra parameter t also makes it harder to solve equations because there is one extra unknown variable. Let's take a look at some alternative formulas with other advantages.

If you recall any formula for a line from high school, it is probably $y = m \cdot x + b$. This formula is useful because it gives you a y -coordinate explicitly as a function of the x -coordinate. In this form, it's easy to graph a line: you go through a bunch of x -values, compute the corresponding y -values, and put dots at the resulting (x,y) points. But this formula also has some limitations. Most importantly you can't represent a vertical line, like $r(t) = (3,0) + t \cdot (0,1)$. This is the line consisting of vectors where $x = 3$.

We'll continue to use the *parametric* formula $r(t) = u + t \cdot v$ because it avoids this problem, but it would be great to have a formula with no extra parameter t that can represent any line. The one we'll use is: $ax + by = c$. As an example, the line we've been looking at in the last few images can be written as $x + 2y = 8$. That is, it is the set of (x,y) points in the plane satisfying that equation.

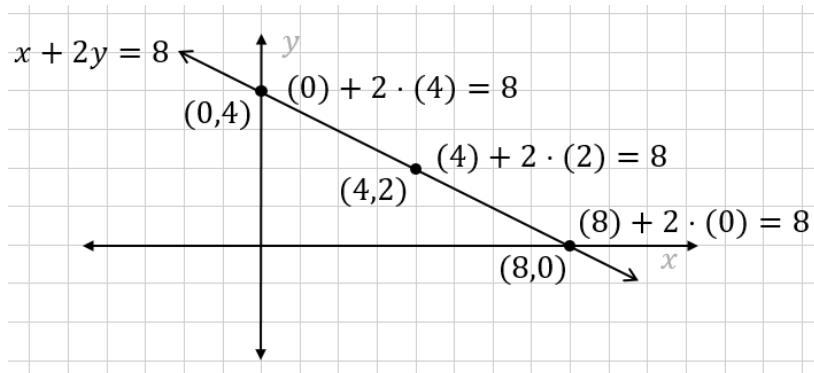


Figure 7.9 All (x,y) points on the line we've been looking at satisfy $x + 2y = 8$.

The form $ax + by = c$ has no extra parameters and can represent any line. Even a vertical line can be written in this form, for instance $x = 3$ is the same as $1 \cdot x + 0 \cdot y = 3$. Any equation representing a line is called a *linear equation*, and this in particular is called the *standard form* for a linear equation. We'll prefer it in this chapter because it will make it easy to organize our computations.

7.2.2 Finding the standard form equation for a line

The formula $x + 2y = 8$ was the line containing one of the segments on the example asteroid. Next, we'll look at another one and then try to systematize finding the standard form for linear equations. Brace yourself for a bit of algebra: I'll explain each of the steps carefully, but it may be a bit dry to read. You'll have a better time if you follow along on your own with a pencil and paper.

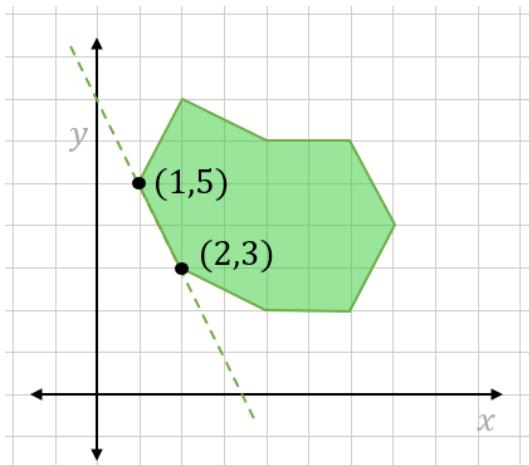


Figure 7.10 The points $(1,5)$ and $(3,2)$ define a second segment of the asteroid.

The vector $(1,5) - (2,3)$ is $(-1,2)$, which is parallel to the line. Since $(2,3)$ lies on the line, a parametric equation for the line is $r(t) = (2,3) + t \cdot (-1,2)$. Knowing that all points on the line have the form $(2,3) + t \cdot (-1,2)$ for some t , how can we rewrite this condition to be a standard form equation? We'll need to do some algebra, and particularly get rid of t .

Since $(x,y) = (2,3) + t \cdot (-1,2)$, we really have two equations to start with:

$$x = 2 - t$$

$$y = 3 + 2t$$

We can manipulate both of them to get two new equations that have the same value ($2t$).

$$4 - 2x = 2t$$

$$y - 3 = 2t$$

Since both of the expressions on the left-hand sides equal $2t$, they equal each other:

$$4 - 2x = y - 3$$

We've now gotten rid of t ! Finally, pulling the x and y terms to one side, we get the standard form equation.

$$2x + y = 7.$$

This process isn't too hard, but we need to be more precise about how to do it if we want to convert it to code. Let's try to solve the general problem: given two points (x_1, y_1) and (x_2, y_2) , what is the equation of the line that passes through them?

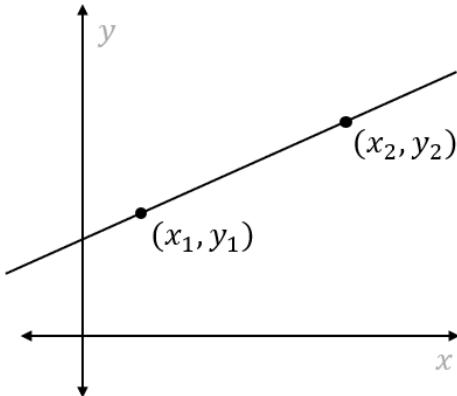


Figure 7.11 The general problem of finding the equation of the line that passes through two known points.

Using the parametric formula, the points on the line have the following form:

$$(x, y) = (x_1, y_1) + t \cdot (x_2 - x_1, y_2 - y_1)$$

There are a lot of x and y variables here, but remember that x_1 , x_2 , y_1 , and y_2 are all constants for the purpose of this discussion. We assume we have two points with known coordinates, and we could have called them (a, b) and (c, d) just as easily. The variables are x and y (with no subscripts), which stand for coordinates of *any* point on the line. As before, we can break this equation into two pieces:

$$x = x_1 + t \cdot (x_2 - x_1)$$

$$y = y_1 + t \cdot (y_2 - y_1)$$

We can move x_1 and y_1 to the left-hand side of their respective equations.

$$x - x_1 = t \cdot (x_2 - x_1)$$

$$y - y_1 = t \cdot (y_2 - y_1)$$

Our next goal is to make the right-hand side of both equations look the same, so we can set the left-hand sides equal to each other. Multiplying both sides of the first equation by $(y_2 - y_1)$ and both sides of the second equation by $(x_2 - x_1)$ gives us:

$$(y_2 - y_1) \cdot (x - x_1) = t \cdot (x_2 - x_1) \cdot (y_2 - y_1)$$

$$(x_2 - x_1) \cdot (y - y_1) = t \cdot (x_2 - x_1) \cdot (y_2 - y_1)$$

Since the right-hand sides are identical, we know that the first and second equations' left-hand sides equal each other. That lets us create a new equation with no t in it.

$$(y_2 - y_1) \cdot (x - x_1) = (x_2 - x_1) \cdot (y - y_1)$$

Remember, we want an equation of the form $ax + by = c$, so we need to get x and y on the same side, and constants on the other side. The first thing we can do is expand both sides:

$$(y_2 - y_1) \cdot x - (y_2 - y_1) \cdot x_1 = (x_2 - x_1) \cdot y - (x_2 - x_1) \cdot y_1.$$

Then we can move the constants to the left and the variables to the right:

$$(y_2 - y_1) \cdot x - (x_2 - x_1) \cdot y = (y_2 - y_1) \cdot x_1 - (x_2 - x_1) \cdot y_1.$$

Expanding the right side, we see some of the terms cancel out:

$$(y_2 - y_1) \cdot x - (x_2 - x_1) \cdot y = y_2 x_1 - y_1 x_1 - x_2 y_1 + x_1 y_1 = x_1 y_2 - x_2 y_1.$$

We've done it! This is the linear equation in standard form $ax + by = c$, where $a = (y_2 - y_1)$, $b = -(x_2 - x_1)$ or in other words $(x_1 - x_2)$, and $c = (x_1 y_2 - x_2 y_1)$.

Let's check this with the previous example we did, using two points $(x_1, y_1) = (2, 3)$ and $(x_2, y_2) = (1, 5)$. In this case,

$$a = y_2 - y_1 = 5 - 3 = 2$$

$$b = -(x_2 - x_1) = -(1 - 2) = 1$$

and

$$c = x_1 y_2 - x_2 y_1 = 2 \cdot 5 - 3 \cdot 1 = 7.$$

As expected, this means the standard form equation is $2x + y = 7$. This formula seems trustworthy! As one final application, let's find the standard form equation for the line defined by the laser. It looks like it passes through $(2, 2)$ and $(4, 4)$ as I drew it before:

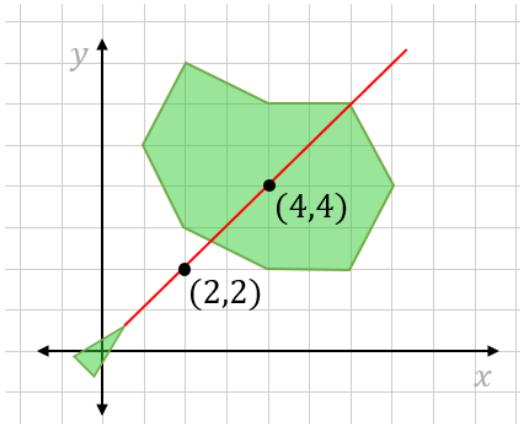


Figure 7.12 The laser passes through the points $(2,2)$ and $(4,4)$.

In our asteroid game, we have exact start and end points for the laser line segment, but these numbers are nice for an example. Plugging in to the formula, we find

$$a = y_2 - y_1 = 4 - 2 = 2$$

$$b = -(x_2 - x_1) = -(4 - 2) = -2$$

and

$$c = x_1y_2 - x_2y_1 = 2 \cdot 4 - 2 \cdot 4 = 0.$$

This means the line is $2y - 2x = 0$, which is equivalent to saying $x - y = 0$ (or simply $x = y$). To decide whether the laser hits the asteroid, we'll have to find where the line $x - y = 0$ intersects the line $x + 2y = 8$, the line $2x + y = 7$, or any of the other lines bounding the asteroid.

7.2.3 Linear equations in matrix notation

Let's focus on an intersection we can see: the laser clearly hits the closest edge of the asteroid, whose line has equation $x + 2y = 8$.

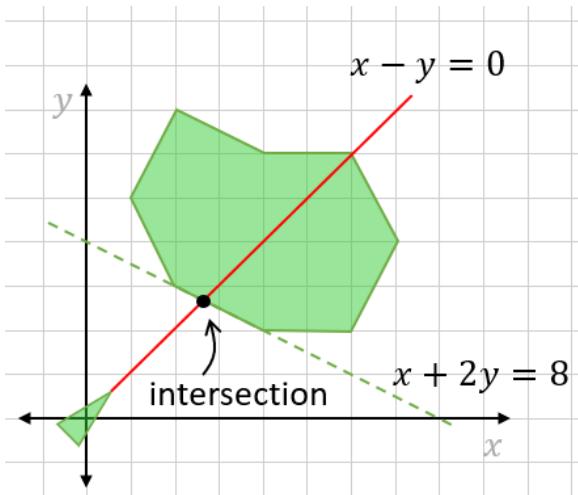


Figure 7.13 The laser hits the asteroid where the lines $x - y = 0$ and $x + 2y = 8$ intersect.

After quite a bit of build-up, we've met our first real system of linear equations. It's customary to write systems of linear equations in a grid like this, so that the variables x and y line up.

$$x - y = 0$$

$$x + 2y = 8$$

Thinking back to chapter 5, we can organize these two equations into a single matrix equation. One way to do this is to write a linear combination of column vectors, where x and y are coefficients.

$$x \begin{pmatrix} 1 \\ 1 \end{pmatrix} + y \begin{pmatrix} -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 8 \end{pmatrix}$$

Another way is to consolidate this even further, and write it as a matrix multiplication. The linear combination of $(1,1)$ and $(-1,2)$ with coefficients x and y is the same as a matrix product:

$$\begin{pmatrix} 1 & -1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 8 \end{pmatrix}$$

When we write it this way, the task of solving the system of linear equations looks like solving for a vector in a matrix multiplication problem. If we call the 2-by-2 matrix A , the problem becomes "what vector (x,y) is multiplied by the matrix A to yield $(0,8)$?" In other words, we

know that an output of the linear transformation A is $(0,8)$ and we want to know what input yielded it.

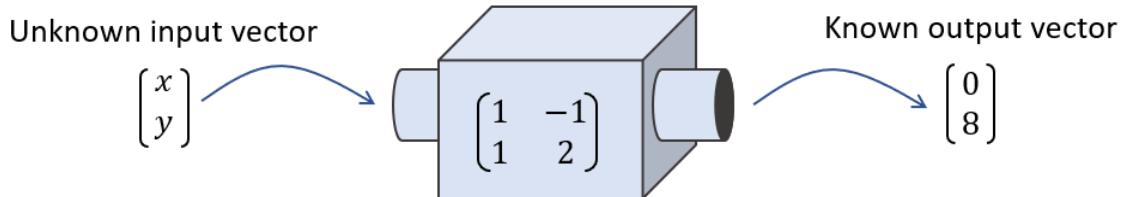


Figure 7.14 Framing the problem as finding an input vector that yields the desired output vector.

We're talking about a bunch of different concepts at once here: linear equations, linear combinations, matrices, and linear transformations. The point is that this is a fundamental kind of problem in linear algebra; you can look at it from several different perspectives. In the next section we'll see how they all fit together, but for now let's focus on solving the problem at hand.

7.2.4 Solving linear equations with numpy

Finding the intersection of $x - y = 0$ and $x + 2y = 8$ is the same as finding the vector (x,y) that satisfies the matrix multiplication equation:

$$\begin{pmatrix} 1 & -1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 8 \end{pmatrix}$$

This is only a notational difference, but framing the problem in this form allows us to use pre-built tools to solve it. Specifically, Python's NumPy library has a linear algebra module and a function that solves this kind of equation.

```
>>> import numpy as np
>>> matrix = np.array(((1,-1),(1,2))) #❶
>>> output = np.array((0,8))           #❷
>>> np.linalg.solve(matrix,output)    #❸
array([2.66666667, 2.66666667])    #❹
```

- ❶ The matrix needs to be packaged as a NumPy array object.
- ❷ The output vector also needs to be packaged as a NumPy array (though it needn't be reshaped to be a column vector).
- ❸ The `numpy.linalg.solve` library function takes a matrix and an output vector and finds the input vector that produces it.
- ❹ The result is $(x,y) = (2.66\dots, 2.66\dots)$.

Numpy has told us that the x and y coordinates of the intersection are approximately $2\frac{2}{3}$ or $\frac{8}{3}$ each, which looks about right geometrically. Eyeballing the diagram, it looks like both

coordinates of the intersection point should be between 2 and 3. We can check to see that this point lies on both lines by plugging it in to both equations.

$$1x - 1y = 1 \cdot (2.66666667) - 1 \cdot (2.66666667) = 0$$

$$1x + 2y = 1 \cdot (2.66666667) + 2 \cdot (2.66666667) = 8.00000001$$

These results are close enough to (0,8), and indeed would make an exact solution. This solution vector, roughly (8/3, 8/3) is also the vector that satisfies the matrix equation above.

$$\begin{pmatrix} 1 & -1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 8/3 \\ 8/3 \end{pmatrix} = \begin{pmatrix} 0 \\ 8 \end{pmatrix}$$

We can picture (8/3, 8/3) as the vector we pass into the linear transformation machine defined by the matrix that gives us the desired output vector.

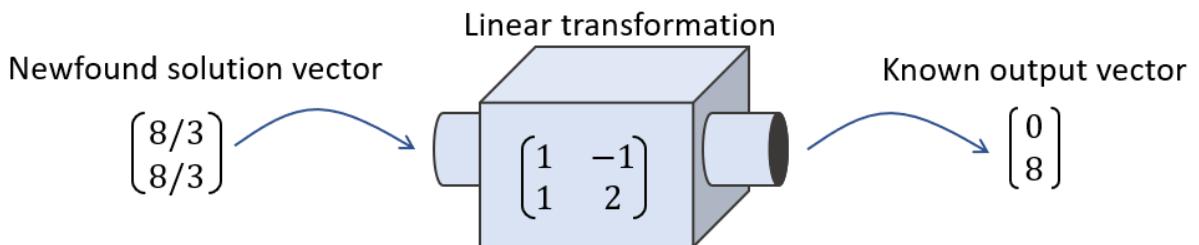


Figure 7.15 $(8/3, 8/3)$ is the vector that, when passed to the linear transformation, produces the desired output $(0,8)$.

We can think of the Python function `numpy.linalg.solve` as a differently shaped machine, which takes in matrices and output vectors and returns the “solution” vectors for the linear equation they represent together.

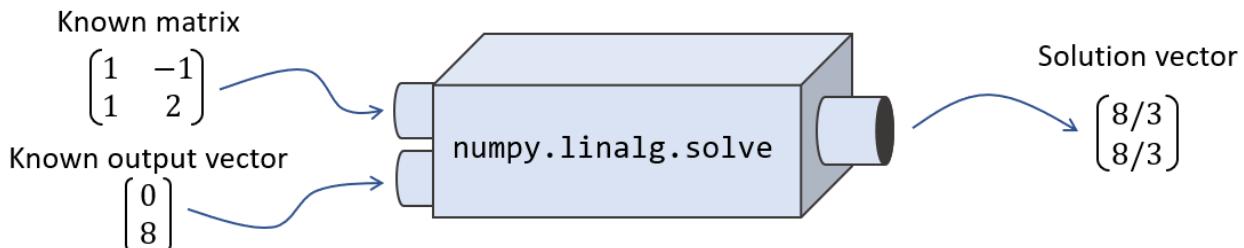


Figure 7.16 The `numpy.linalg.solve` function takes a matrix and a vector and outputs the solution vector to the linear system they represent.

This is perhaps the most important computational task in linear algebra: starting with a matrix A , and a vector w , and finding the vector v such that $Av = w$. Such a vector gives the solution to a system of linear equations represented by A and w . We're lucky to have a Python function that can do this for us, so we don't have to worry about the tedious algebra required to do it by hand. We can now use this function to find when our laser hits asteroids.

7.2.5 Deciding whether the laser hits an asteroid

The missing piece of our game was an implementation for the `does_intersect` method on the `PolygonModel` class. For any instance of this class, which represents a polygon-shaped object living in our 2D game world, this method should return true if an input line segment intersects any line segment of the polygon.

We'll need a few helper functions. First, we'll need to convert given line segments from pairs of endpoint vectors to linear equations in standard form. At the end of the section, I've given you an exercise to implement the function `standard_form` which takes two input vectors and returns a tuple (a,b,c) where $ax+by=c$ is the line on which the segment lies.

Next, given two segments, each represented by its pair of endpoint vectors, we'll want to find out where their lines intersect. If u_1 and u_2 are endpoints of the first segment and v_1 and v_2 are endpoints of the second, we need to first find the standard form equations and then pass them to NumPy to solve.

```
def intersection(u1,u2,v1,v2):
    a1, b1, c1 = standard_form(u1,u2)
    a2, b2, c2 = standard_form(v1,v2)
    m = np.array(((a1,b1),(a2,b2)))
    c = np.array((c1,c2))
    return np.linalg.solve(m,c)
```

The output is the point where the two lines on which the segments lie intersect. But this point may not lie on either of the segments, as shown below.

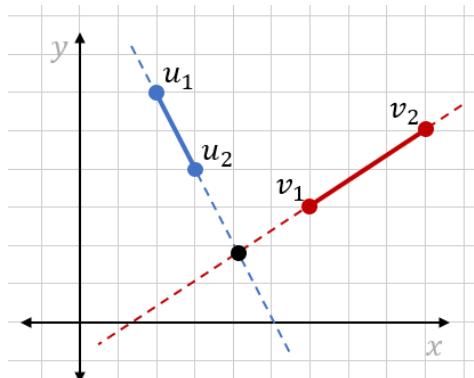


Figure 7.17 One segment connects u_1 and u_2 and the other connects points v_1 and v_2 . The lines extending the segments intersect, but the segments themselves don't.

To detect whether the two *segments* intersect, we need to check that the intersection point of their lines lies between the two pairs of endpoints. We can check that using distances. In the diagram above, the intersection point is further from the point v_2 than the point v_1 is. Likewise it's further from u_1 than u_2 is. These indicate that the point is on neither segment. With four total distance checks, we can confirm whether the intersection point of the lines, (x,y) , is an intersection point of the segments as well.

```
def do_segments_intersect(s1,s2):
    u1,u2 = s1
    v1,v2 = s2
    d1, d2 = distance(*s1), distance(*s2) #①
    x,y = intersection(u1,u2,v1,v2) #②
    return (distance(u1, (x,y)) <= d1 and #③
            distance(u2, (x,y)) <= d1 and
            distance(v1, (x,y)) <= d2 and
            distance(v2, (x,y)) <= d2)
```

- ① Store the lengths of the first and second segments as d_1 and d_2 respectively.
- ② Find the intersection point (x,y) of the lines on which the segments lie.
- ③ Do four checks to make sure the intersection point lies between the four endpoints of the line segments, confirming the segments intersect.

Finally, we can write the `does_intersect` method by checking whether `do_segments_intersect` returns `True` for the input segment and any of the edges of the (transformed) polygon.

```
class PolygonModel():
    ...
    def does_intersect(self, other_segment):
        point_count = len(self.points)
        points = self.transformed()
        segments = [(points[i], points[(i+1)%point_count]) #①
                    for i in range(0,point_count)] #②
        for segment in segments:
            if do_segments_intersect(other_segment,segment): #③
                return True
        return False
```

- ① Transform (rotate and translate) the polygon first, so we use the actual positions of its segments.
- ② Obtain the edge segments of the polygon, by taking all pairs of adjacent vertex points, including the last and first points.
- ③ If any of the segments of the polygon intersect `other_segment`, the method should return `True`.

In the exercises, you can confirm that this actually works by building an asteroid with known coordinate points and a laser beam with a known start and end point. With `does_intersect` implemented as in the source code, you should be able to rotate the spaceship to aim at asteroids and destroy them.

7.2.6 Identifying unsolvable systems

Let me leave you with one final admonition: not every system of linear equations in two dimensions can be solved. It's rare in an application like the asteroid game, but some pairs of linear equations in 2D don't have unique solutions, or even solutions at all. If we pass NumPy a system of linear equations with no solution, we'll get an exception, so we need to handle this case.

When a pair of lines in 2D are *not* parallel, they intersect somewhere. Even these two lines which are nearly parallel -- but not quite -- will intersect somewhere off in the distance.

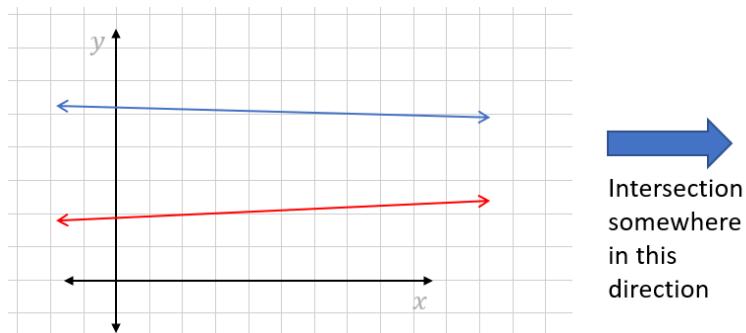


Figure 7.18 Two lines which are not quite parallel, and intersect somewhere in the distance.

Where we run into trouble is when the lines are parallel, meaning they never intersect (or they're the same line!).

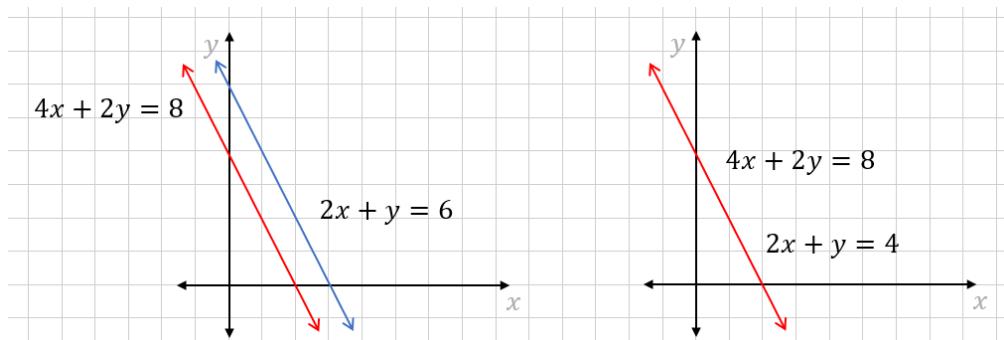


Figure 7.19 A pair of parallel lines that never intersect, and a pair of parallel lines which are in fact the same line (despite having different equations).

In the first case there are zero intersection points, while in the second there are *infinitely many* intersection points -- every point on the line is an intersection point. Both of these cases are problematic computationally, since our code demands a single, unique result. If we

try to solve either of these systems with NumPy, for instance the system consisting of $2x + y = 6$ and $4x + 2y = 8$, we get an exception.

```
>>> import numpy as np
>>> m = np.array(((2,1),(4,2)))
>>> v = np.array((6,4))
>>> np.linalg.solve(m,v)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
numpy.linalg.LinAlgError: Singular matrix
```

NumPy points to the matrix as the source of the error. The matrix

$$\begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix}$$

is called a *singular* matrix, meaning there is no unique solution to the linear system. A system of linear equations is defined by a matrix and a vector, but the matrix on its own is enough to tell us whether the lines are parallel and whether the system has a unique solution. For any non-zero w we pick, there won't be a unique v that solves the system.

$$\begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix} v = w$$

Figure: Regardless of what w we pick, there won't be a unique v which solves the system. The matrix itself is problematic.

We'll philosophize more about singular matrices later, but for now you can see that the rows $(2,1)$ and $(4,2)$ and the columns $(2,4)$ and $(1,2)$ are both parallel and therefore linearly dependent. This is the key clue that tells us the lines are parallel and that the system does not have a unique solution. Solvability of linear systems is one of the central concepts in linear algebra -- it closely relates to the notions of linear independence and dimension. We'll discuss it in the last two sections of the chapter.

For the purpose of our asteroid game, we can make the simplifying assumption that any parallel line segments don't intersect. Given that we're building the game with random floats, it's highly unlikely that any two segments are exactly parallel. Even if the laser lined up exactly with the edge of an asteroid, this would be a glancing hit, and the player doesn't deserve to have the asteroid destroyed. We can modify `do_segments_intersect` to catch the exception and return the default result of `False`.

```
def do_segments_intersect(s1,s2):
    u1,u2 = s1
    v1,v2 = s2
    l1, l2 = distance(*s1), distance(*s2)
    try:
        x,y = intersection(u1,u2,v1,v2)
        return (distance(u1, (x,y)) <= l1 and
```

```

        distance(u2, (x,y)) <= 11 and
        distance(v1, (x,y)) <= 12 and
        distance(v2, (x,y)) <= 12)
    except np.linalg.LinAlgError:
        return False

```

7.2.7 Exercises

EXERCISE

It's possible that $u + tv$ can be a line through the origin. In this case, what can you say about the vectors u and v ?

SOLUTION

One possibility is that $u = (0,0)$, in which case the line automatically passes through the origin; the point $u + 0v$ would be the origin in this case, regardless of what v is. Otherwise, if u and v are scalar multiples, say $u = sv$, then the line will pass through the origin as well because $u - sv = 0$ is on the line.

EXERCISE

If $v = (0,0)$, do points of the form $u + tv$ represent a line?

SOLUTION

No, regardless of the value of t , $u + tv = u + t(0,0) = u$. Every point of this form is equal to u .

EXERCISE

It turns out that the formula $u + tv$ is not unique; that is, you can pick different values of u and v and represent the same line. What is another line representing $(2,2) + t(-1,3)$?

SOLUTION

One possibility is to replace $v = (-1,3)$ with a scalar multiple of itself, like $(2, -6)$. The points of the form $(2,2)+t(-1,3)$ agree with the points $(2,2)+s(2,-6)$ when $t = -2s$. You can also replace u with any point on the line. Since $(2,2) + 1(-1,3) = (1,5)$ is on the line, $(1,5) + t(2,-6)$ is a valid equation for the same line as well.

EXERCISE

Does $ax + by = c$ represent a line for *any* values of a , b , and c ?

SOLUTION

No, if both a and b are zero, the equation does not describe a line. In that case, the formula would be $0 \cdot x + 0 \cdot y = c$. If $c = 0$, this would always be true, and if $c \neq 0$, it would never be true. Either way, it would establish no relationship between x and y , and therefore it would not describe a line.

EXERCISE

Find another equation for the line $2x + y = 3$, showing that the choices of a, b, and c are not unique.

SOLUTION

One example of another equation is $6x + 3y = 9$. In fact, multiplying both sides of the equation by the same non-zero number gives you a different equation for the same line.

SOLUTION

The equation $ax + by = c$ is equivalent to an equation involving a dot product of two 2D vectors: $(a,b) \cdot (x,y) = c$. You could therefore say that a line is a set of vectors whose dot product with a given vector is constant. What is the geometric interpretation of this statement?

SOLUTION

See the discussion in the next section.

EXERCISE

Confirm $(0,7)$ and $(3.5,0)$ satisfy the equation $2x + y = 7$.

SOLUTION

$2 \cdot 0 + 7 = 7$ and $2(3.5) + 0 = 7$.

EXERCISE

Graph $(3,0) + t(0,1)$ and convert it to standard form using the formula.

SOLUTION

This is a vertical line where $x = 3$.

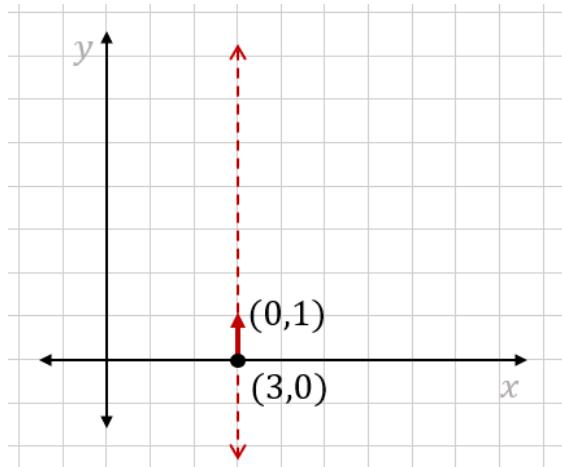


Figure 7.20 $(3,0) + t(0,1)$ yields a vertical line.

The formula “ $x = 3$ ” is already the equation of our line in standard form, but we can confirm this with the formulas. The first point on our line is already given: $(x_1,y_1) = (3,0)$. A second point on the line is $(3,0) + (0,1) = (3,1) = (x_2,y_2)$. We have $a = y_2 - y_1 = 1$, $b = x_1 - x_2 = 0$, and $c = x_1y_2 - x_2y_1 = 3 \cdot 1 - 1 \cdot 0 = 3$. This gives us $1x + 0y = 3$ or simply $x = 3$.

EXERCISE

Write the Python function `standard_form` that takes two vectors `v1` and `v2` and finds the equation line $ax + by = c$ passing through both of them. Specifically, it should output the tuple of constants (a,b,c) .

SOLUTION

All we need to do is translate the formulas we wrote down into Python.

```
def standard_form(v1, v2):
    x1, y1 = v1
    x2, y2 = v2
    a = y2 - y1
    b = x1 - x2
    c = x1 * y2 - y1 * x2
    return a,b,c
```

MINI-PROJECT

For each of the four distance checks in `do_segments_intersect`, find a pair of line segments which fail it but pass the other three checks.

SOLUTION

To make it easier to run experiments, we can create a modified version of `do_segments_intersect` that returns a list of the truth values returned by each of the four checks.

```
def segment_checks(s1,s2):
    u1,u2 = s1
    v1,v2 = s2
    l1, l2 = distance(*s1), distance(*s2)
    x,y = intersection(u1,u2,v1,v2)
    return [
        distance(u1, (x,y)) <= d1,
        distance(u2, (x,y)) <= d1,
        distance(v1, (x,y)) <= d2,
        distance(v2, (x,y)) <= d2
    ]
```

In general, these checks will fail when one endpoint of a segment is closer to the other endpoint than to the intersection point. Here are some solutions I found, using segments on the lines $y = 0$ and $x = 0$, which intersect at the origin. Each of these fails exactly one of the four checks. If in doubt, draw them yourself and see what is going on!

```
>>> segment_checks((-3,0),(-1,0)),((0,-1),(0,1)))
[False, True, True, True]
>>> segment_checks((1,0),(3,0)),((0,-1),(0,1))
[True, False, True, True]
>>> segment_checks((-1,0),(1,0)),((0,-3),(0,-1))
[True, True, False, True]
>>> segment_checks((-1,0),(1,0)),((0,1),(0,3))
[True, True, True, False]
```

EXERCISE

For the example laser line and asteroid, confirm the `does_intersect` function returns true. Hint: use grid lines to find the vertices of the asteroid and build a `PolygonModel` object representing it.

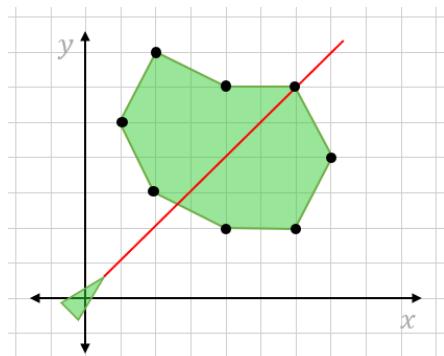


Figure 7.21 The laser hits the asteroid.

SOLUTION

In counterclockwise order starting with the topmost point, the vertices are $(2,7)$, $(1,5)$, $(2,3)$, $(4,2)$, $(6,2)$, $(7,4)$, $(6,6)$, and $(4,6)$. We can assume the endpoints of the laser beam are $(1,1)$ and $(7,7)$.

```
>>> from asteroids import PolygonModel
>>> asteroid = PolygonModel([(2,7), (1,5), (2,3), (4,2), (6,2), (7,4), (6,6),
```

```
(4,6])
>>> asteroid.does_intersect([(0,0),(7,7)])
True
```

This confirms the laser hits the asteroid! By contrast, a shot directly up the y-axis from (0,0) to (0,7) does not hit:

```
>>> asteroid.does_intersect([(0,0),(0,7)])
False
```

EXERCISE

Write a `does_collide(other_polygon)` method to decide whether the current `PolygonModel` object has collided with another one, `other_polygon`, by checking whether any of the segments that define the two are intersecting. This could help us decide whether an asteroid has hit the ship or another asteroid.

SOLUTION

First, it's convenient to add a `segments()` method to `PolygonModel` to avoid duplication of the work of returning the (transformed) line segments that constitute the polygon. Then, we can check every segment of the other polygon to see if it returns true for `does_intersect` with the current one.

```
class PolygonModel():
    ...
    def segments(self):
        point_count = len(self.points)
        points = self.transformed()
        return [(points[i], points[(i+1)%point_count])
                for i in range(0,point_count)]

    def does_collide(self, other_poly):
        for other_segment in other_poly.segments():
            if self.does_intersect(other_segment):
                return True
        return False
```

We can test this by building some squares which should and shouldn't overlap, and seeing whether the `does_collide` method correctly detects which is which. Indeed, it does:

```
>>> square1 = PolygonModel([(0,0), (3,0), (3,3), (0,3)])
>>> square2 = PolygonModel([(1,1), (4,1), (4,4), (1,4)])
>>> square1.does_collide(square2)
True
>>> square3 = PolygonModel([(-3,-3), (-2,-3), (-2,-2), (-3,-2)])
>>> square1.does_collide(square3)
False
```

MINI-PROJECT

We can't pick a vector w so that the following system has a unique solution v .

$$\begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix} v = w$$

Figure 7.22 A linear system which can't have a unique solution, regardless of the vector w .

Find a vector w such that there are *infinitely* many solutions to the system, that is infinitely many values of v that satisfy the equation.

SOLUTION

If $w = (0,0)$, for example, the two lines represented by the system are identical. (Graph them if you are skeptical!) The solutions have the form $v = (a, -2a)$ for any real number a . Here are some of the infinitely many possibilities for v .

$$\begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ -2 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix} \begin{pmatrix} -4 \\ 8 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix} \begin{pmatrix} 10 \\ -20 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Figure 7.23 Some of the infinitely many possibilities that solve the system when $w = (0,0)$.

7.3 Generalizing linear equations to higher dimensions

Now that we've built a functional (albeit minimal) game, let's broaden our perspective. We can represent a wide variety of problems as systems of linear equations, not just arcade games. Linear equations in the wild often have more than two "unknown" variables, x and y . Such equations describe collections of points in more than two dimensions. In more than three dimensions, it's hard to picture much of anything, but the 3D case can be a useful mental model. Planes in 3D end up being the analogy of lines in 2D, and they are also represented by linear equations.

7.3.1 Representing planes in 3D

To see why lines and planes are analogous, it's useful to think of lines in terms of dot products. As you saw in a previous exercise, or may have noticed yourself, the equation $ax + by = c$ is the set of points (x,y) in the 2D plane where the dot product with a fixed vector (a,b) is equal to a fixed number, c . That is, the equation $ax + by = c$ is equivalent to the equation $(a,b) \cdot (x,y) = c$. In case you didn't figure out how to interpret this geometrically in the exercise, let's go through it here.

If we have a point and a (non-zero) vector in 2D, there's a unique line which is perpendicular to the vector and also passes through the point.

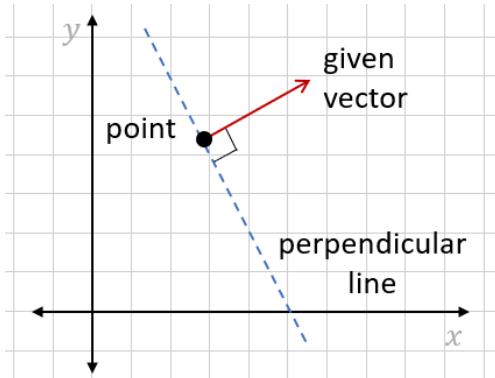


Figure 7.24 A unique line passing through a given point and perpendicular to a given vector.

If we call the given point (x_0, y_0) and the given vector (a, b) , we can write down a criterion for a point (x, y) to lie on the line. Specifically if (x, y) lies on the line, then $(x - x_0, y - y_0)$ is parallel to the line, and perpendicular to (a, b) .

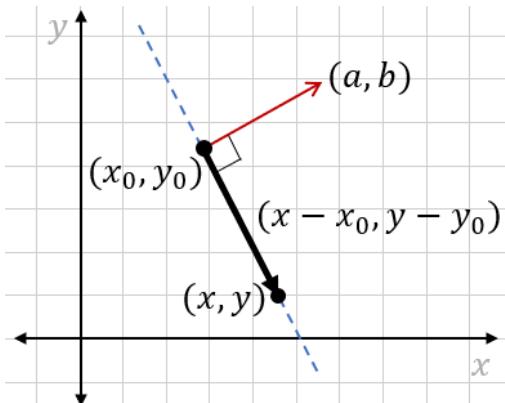


Figure 7.25 The vector $(x - x_0, y - y_0)$ is parallel to the line, and therefore perpendicular to (a, b) .

Since two perpendicular vectors have zero dot product, that's equivalent to the algebraic statement:

$$(a, b) \cdot (x - x_0, y - y_0) = 0.$$

That dot product expands to

$$a(x - x_0) + b(y - y_0) = 0$$

or

$$ax + by = ax_0 + by_0.$$

The quantity on the right-hand side of this equation is a constant, so we can rename it c , giving us the general form equation for a line: $ax + by = c$.

This is a handy geometric interpretation of the formula $ax + by = c$, and one that we can generalize to 3D. Given a point and a vector in 3D, there is a unique *plane* perpendicular to the vector and passing through the point. If the vector is (a,b,c) and the point is (x_0, y_0, z_0) we can conclude that if a vector (x,y,z) lies in the plane, then $(x-x_0, y-y_0, z-z_0)$ is perpendicular to (a,b,c) .

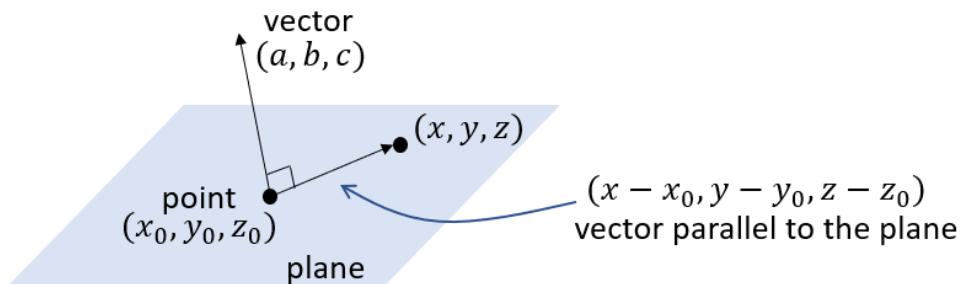


Figure 7.26 A plane parallel to the vector (a,b,c) which passes through the point (x_0,y_0,z_0) .

Every point on the plane gives us such a perpendicular vector to (a,b,c) , and every vector perpendicular to (a,b,c) leads us to a point in the plane. We can express this perpendicularity as a dot product of the two vectors, so the equation satisfied by every point (x,y,z) in the plane is:

$$(a,b,c) \cdot (x - x_0, y - y_0, z - z_0) = 0$$

This expands to

$$ax + by + cz = ax_0 + by_0 + cz_0$$

and since the right-hand side of the equation is a constant, we can conclude that every plane in 3D has an equation of the form $ax + by + cz = d$. In 3D, the computational problem will be to decide where planes intersect, or which values of (x,y,z) simultaneously satisfy multiple linear equations like this.

7.3.2 Solving linear equations in 3D

A pair of non-parallel lines in the plane intersects at exactly one point. Is that true for planes? If we draw a pair of intersecting planes, we can see that it's possible for non-parallel planes to intersect at many points. In fact, there is a whole *line*, consisting of infinitely many points, where two non-parallel planes intersect.

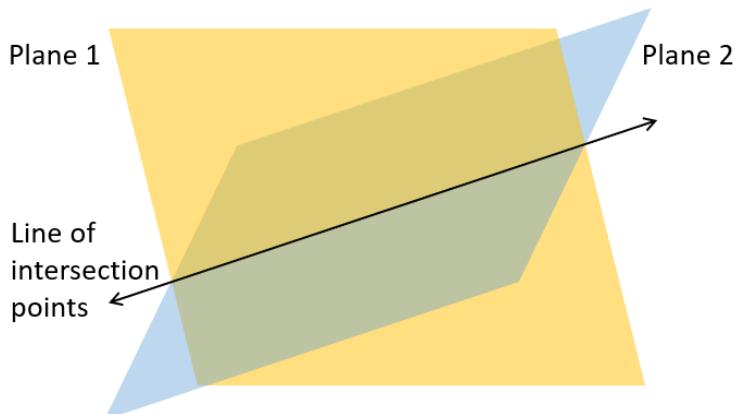


Figure 7.27 Two non-parallel planes intersect along a line.

Only if you add a third plane, which is not parallel to this intersection line, can you find a unique intersection point; each pair among the three planes intersects along a line, and the lines share a single point.

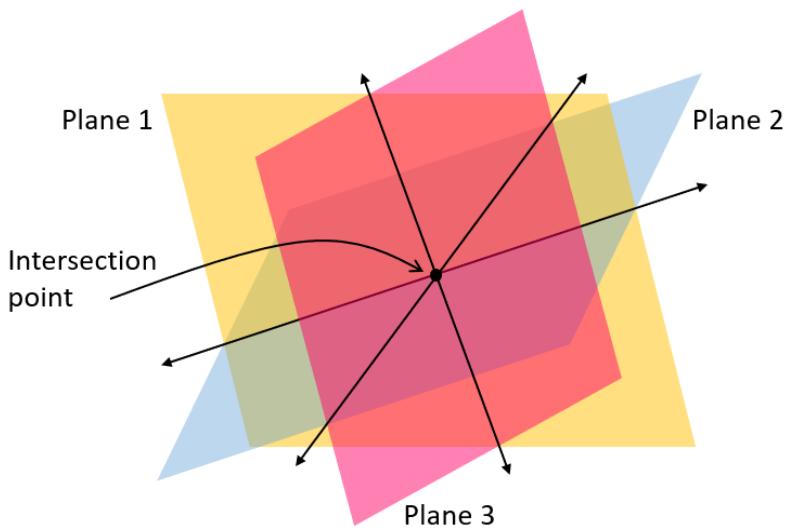


Figure 7.28 Two non-parallel planes intersect along a line.

Finding this point algebraically requires finding a common solution to three linear equations in three variables, each representing one of the planes, and having form $ax + by + cz = d$. Such a system of three linear equations would have the form:

$$a_1x + b_1y + c_1z = d_1$$

$$a_2x + b_2y + c_2z = d_2$$

$$a_3x + b_3y + c_3z = d_3$$

Each plane is determined by four numbers: a_i , b_i , c_i , and d_i , where $i = 1, 2$, or 3 is the index of the plane we're looking at. Subscripts like this are useful for systems of linear equations, where there can be a lot of variables that need to be named. These twelve numbers in total are enough to find the point (x,y,z) where the planes intersect, if there is one. To solve the system, we can convert the system into a matrix equation:

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

Let's try an example. Say our three planes are given by the following equations

$$x + y - z = -1$$

$$2y - z = 3$$

$$x + z = 2$$

You can see how to plot these planes in Matplotlib in the source code. The result is:

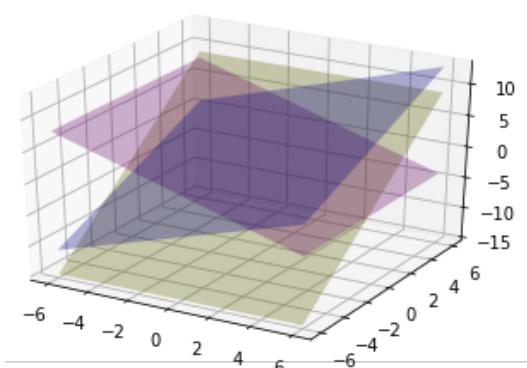


Figure 7.29 Three planes plotted in Matplotlib.

It's not easy to see, but somewhere in there the three planes intersect. To find that intersection point, we need the values of x , y , and z that satisfy all three linear equations simultaneously. Once again, we can convert the system to matrix form and use NumPy to solve it. The matrix equation which is equivalent to this linear system is

$$\begin{pmatrix} 1 & 1 & -1 \\ 0 & 2 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -1 \\ 3 \\ 2 \end{pmatrix}$$

Converting the matrix and vector to NumPy arrays in Python, we can quickly find the solution vector:

```
>>> matrix = np.array(((1,1,-1),(0,2,-1),(1,0,1)))
>>> vector = np.array((-1,3,2))
>>> np.linalg.solve(matrix,vector)
array([-1.,  3.,  3.])
```

This tells us that $(-1,3,3)$ is the (x,y,z) point where all three planes intersect, and the point which satisfies all three linear equations simultaneously.

While this result was easy to compute with NumPy, you can see it's already a bit harder to visualize systems of linear equations in 3D. Beyond 3D, it's difficult if not impossible to visualize linear systems, but solving them is mechanically the same. The analogy to a line or a plane in any number of dimensions is called a *hyperplane*, and the problem boils down to finding the points where multiple hyperplanes intersect.

7.3.3 Studying hyperplanes algebraically

To be precise, a hyperplane in n dimensions is a solution to a linear equation in n unknown variables. A line is a 1D hyperplane living in 2D, and a plane is a 2D hyperplane living in 3D. As you might guess, a linear equation in standard form in 4D has the form:

$$aw + bx + cy + dz = e.$$

The set of solutions (w,x,y,z) form a region which is a "3D" hyperplane living in 4D space. I need to be careful when I use the adjective "3D," because it isn't necessarily a 3D vector subspace of \mathbb{R}^4 . This is analogous to the 2D case -- the lines passing through the origin in 2D are vector subspaces of \mathbb{R}^2 but other lines are not. Vector space or not, the "3D hyperplane" is 3D in the sense that there are three linearly independent directions you could travel in the solution set, like there are two linearly independent directions you can travel on any plane. I've included a mini-project at the end of the section to help you check your understanding of this.

When we write linear equations in even higher numbers of dimensions, we're in danger of running out of letters to represent coordinates and coefficients. To solve this, we'll use letters with subscript indices. For instance, in 4D we could write a linear equation in standard form as:

$$a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 = b$$

Here, the coefficients are a_1 , a_2 , a_3 , and a_4 , and the 4D vector has coordinates (x_1, x_2, x_3, x_4) . We could just as easily write a linear equation in 10-dimensions:

$$a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5x_5 + a_6x_6 + a_7x_7 + a_8x_8 + a_9x_9 + a_{10}x_{10} = b$$

When the pattern of terms we're summing is clear, we sometimes use "..." to save ink. You may see equations like the above written $a_1x_1 + a_2x_2 + \dots + a_{10}x_{10} = b$. Another compact notation you'll see involves the summation symbol, Σ , which is the Greek letter "sigma." If I want to write the sum of terms of the form $a_i x_i$ with the index i ranging from $i = 1$ to $i = 10$, and I want to state that the sum is equal to some other number b , I can use the mathematical shorthand:

$$\sum_{i=1}^{10} a_i x_i = b$$

This equation means the exact same thing as the one above -- it is merely a more concise way of writing it.

Whatever number of dimensions n we're working in, the standard form of a linear equation has the same shape:

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = b.$$

To represent a system of m linear equations in n dimensions, we'll need even more indices. Our array of constants on the left-hand side of the equals sign can be denoted a_{ij} , where the subscript i indicates which equation we're talking about and the subscript j indicates which coordinate (x_j) the constant is multiplied by.

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$

...

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m$$

You can see that I also used "..." to skip equations three through $m-1$ in the middle. There are m equations and n constants in each equation, so there are mn constants of the form a_{ij} in total. On the right-hand side, there are m constants in total, one per equation: b_1, b_2, \dots, b_m .

Regardless of the number of dimensions (the same as the number of unknown variables) and the number of equations, we can represent such a system as a linear equation. The system above with n unknowns and m equations can be re-written as follows:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Figure 7.30 A system of m linear equations n unknowns written in matrix form.

7.3.4 Counting dimensions, equations, and solutions

We saw in 2D and in 3D that it's possible to write down linear equations that don't have a solution, or at least not a unique one. How will we know if a system of m equations in n unknowns is solvable? In other words, how will we know if m hyperplanes in n -dimensions have a unique intersection point? We'll discuss this in detail in the last section of the chapter, but there's one important conclusion we can draw now.

In 2D, a pair of lines *can* intersect at a single point. They won't always, for instance if the lines are parallel, but they can. The algebraic equivalent to this statement is that a system of two linear equations in two variables can have a unique solution. In 3D, three planes *can* intersect at a single point. Likewise, this is not always the case, but three is the minimum number of planes (or linear equations) required to specify a point in 3D. With only two planes, you'll have at least a 1D space of possible solutions which is the line of intersection. Algebraically, this means you need two linear equations to get a unique solution in 2D, and three linear equations to get a unique solution in 3D. In general, you need n linear equations to be able to get a unique solution in n -dimensions.

Here's a example, working in 4D with coordinates (x_1, x_2, x_3, x_4) , which may seem overly simple but is useful because of how concrete it is. Let's take our first linear equation to be $x_4 = 0$. The solutions to this linear equation form a 3D hyperplane, consisting of vectors of the form $(x_1, x_2, x_3, 0)$. This is clearly a 3D space of solutions, and it turns out to be a vector subspace of \mathbb{R}^4 , with basis $(1,0,0,0), (0,1,0,0), (0,0,1,0)$.

A second linear equation could be $x_2 = 0$. The solutions of this equation on its own are also a 3D hyperplane. The intersection of these two 3D hyperplanes is a 2D space, consisting of vectors of the form $(x_1, 0, x_3, 0)$ which satisfy both equations. If we could picture such a thing, we would see this as a 2D plane living in 4D. Specifically it is the plane spanned by $(1,0,0,0)$ and $(0,0,1,0)$.

Adding one more linear equation, $x_1 = 0$, which defines its own hyperplane, the solutions to all three equations are now a 1D space. The vectors in this 1D space lie on a line in 4D, and have the form $(0, 0, x_3, 0)$. This line is exactly the x_3 axis, which is a 1D subspace of \mathbb{R}^4 .

Finally, if we impose a fourth linear equation, $x_3 = 0$, the only possible solution is $(0,0,0,0)$, a zero-dimensional vector space. The statements $x_4 = 0, x_2 = 0, x_1 = 0$, and $x_3 = 0$ are in fact linear equations, but they are so simple they describe the solution exactly: $(x_1, x_2, x_3, x_4) = (0,0,0,0)$. Each time we added an equation, we reduced the dimension of the

solution space by one, until we got a zero-dimensional space, consisting of the single point $(0,0,0,0)$.

Had we chosen different equations, each step would not have been as clear -- we would have needed to test whether each successive hyperplane truly reduces the dimension of the solution space by one. For instance, if we started with

$$x_1 = 0$$

and

$$x_2 = 0$$

we would have reduced the solution set to a 2D space, but then adding another equation to the mix:

$$x_1 + x_2 = 0$$

There is no effect on the solution space. Since x_1 and x_2 are already constrained to be zero, the equation $x_1 + x_2 = 0$ is automatically satisfied. This third equation therefore adds no more specificity to the solution set. In the first case, four dimensions with three linear equations to satisfy left us with a $4 - 3 = 1$ dimensional solution space. But in this second case, three equations described a less specific 2D solution space. If you have n dimensions (n unknown variables), and n linear equations, it's possible there's a unique solution -- a zero-dimensional solution space -- but this is not always the case. More generally, if you're working in n dimensions, the lowest-dimension solution space you can get with m linear equations is $n - m$. In that case we call the system of linear equations *independent*.

Every basis vector in a space gives us a new independent direction we can move in the space. Independent directions in a space are sometimes called *degrees of freedom* -- the z direction, for instance, "freed" us from the plane into larger 3D space. By contrast, every independent linear equation we introduce is a constraint -- removing a degree of freedom, and restricting the space of solutions to have a smaller number of dimensions. When the number of independent degrees of freedom (dimensions) equals the number of independent constraints (linear equations) there are no longer any degrees of freedom, and we are left with a unique point. This is a major philosophical point in linear algebra, and one you can explore more in some mini-projects that follow. In the final section of the chapter, we'll connect the concepts of independent equations and (linearly) independent vectors.

7.3.5 Exercises

EXERCISE

What's the equation for a line which passes through $(5,4)$, and which is perpendicular to $(-3,3)$?

SOLUTION:

Here's the set up:

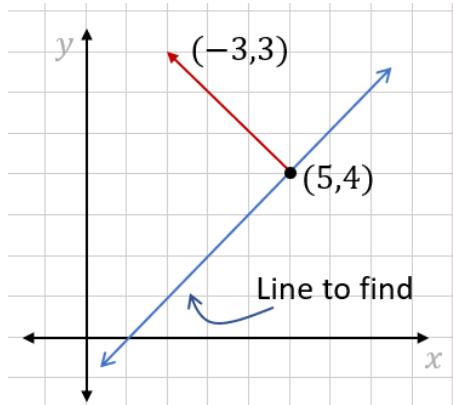


Figure 7.31 Finding a line which passes through $(5,4)$ and which is perpendicular to $(-3,3)$.

For every point (x,y) on the line, the vector $(x-5, y-4)$ is parallel to the line, and therefore perpendicular to $(-3,3)$. That means that the dot product $(x-5, y-4) \cdot (-3, 3)$ is zero for any (x,y) on the line. This equation expands to $-3x + 15 + 3y - 12 = 0$, which rearranges to give $-3x + 3y = -3$. We can divide both sides by -3 to get a simpler, equivalent equation $x - y = 1$.

MINI-PROJECT

Consider a system of two linear equations in 4D.

$$x_1 + 2x_2 + 2x_3 + x_4 = 0$$

$$x_1 - x_4 = 0.$$

Explain algebraically (rather than geometrically) why the solutions form a vector subspace of 4D.

SOLUTION

We can show that if (a_1, a_2, a_3, a_4) and (b_1, b_2, b_3, b_4) are two solutions, then a linear combination of them is a solution as well. That would imply that the solution set contains all linear combinations of its vectors, making it a vector subspace.

Let's start with the assumption that (a_1, a_2, a_3, a_4) and (b_1, b_2, b_3, b_4) are solutions to both linear equations, which explicitly means:

$$a_1 + 2a_2 + 2a_3 + a_4 = 0,$$

$$b_1 + 2b_2 + 2b_3 + b_4 = 0,$$

$$a_1 - a_4 = 0,$$

$$b_1 - b_4 = 0.$$

Picking scalars c and d , the linear combination $c(a_1, a_2, a_3, a_4) + d(b_1, b_2, b_3, b_4)$ is equal to $(ca_1 + db_1, ca_2 + db_2, ca_3 + db_3, ca_4 + db_4)$. Is this a solution to the two equations? We can find out by plugging the four coordinates in for x_1, x_2, x_3 , and x_4 . In the first equation,

$$x_1 + 2x_2 + 2x_3 + x_4$$

becomes

$$(ca_1 + db_1) + 2(ca_2 + db_2) + 2(ca_3 + db_3) + (ca_4 + db_4).$$

That expands to give us

$$ca_1 + db_1 + 2ca_2 + 2db_2 + 2ca_3 + 2db_3 + ca_4 + db_4$$

Which rearranges to

$$c(a_1 + 2a_2 + 2a_3 + a_4) + d(b_1 + 2b_2 + 2b_3 + b_4)$$

Since $a_1 + 2a_2 + 2a_3 + a_4$ and $b_1 + 2b_2 + 2b_3 + b_4$ are both zero, this expression is zero.

$$c(a_1 + 2a_2 + 2a_3 + a_4) + d(b_1 + 2b_2 + 2b_3 + b_4) = c \cdot 0 + d \cdot 0 = 0.$$

That means the linear combination is a solution to the first equation. Similarly, plugging the linear combination into the second equation, we see it's a solution to that equation as well.

$$(ca_1 + db_1) - (ca_4 + db_4) = c(a_1 - a_4) + d(b_1 - b_4) = c \cdot 0 + d \cdot 0 = 0.$$

Any linear combination of any two solutions is also a solution, so the solution set contains all of its linear combinations. That means the solution set is a vector subspace of 4D.

EXERCISE

What is the standard form equation for a plane which passes through the point (1,1,1) and is perpendicular to the vector (1,1,1)?

SOLUTION

For any point (x,y,z) in the plane, the vector (x-1,y-1,z-1) is perpendicular to (1,1,1). That means that the dot product (x-1, y-1, z-1)(1,1,1) is zero for any x, y, and z values giving a point in the plane. This expands to give us $x - 1 + y - 1 + z - 1 = 0$, or $x + y + z = 3$, the standard form equation for the plane.

MINI-PROJECT

Write a python function that takes three 3D points as inputs and returns the standard form equation of the plane that they lie in. For instance, if the standard form equation is $ax + by + cz = d$, the function could return the tuple (a,b,c,d). Hint: differences of any pairs of the three vectors are parallel to the plane, so cross products of the differences will be perpendicular.

SOLUTION

If the points given are p_1 , p_2 , and p_3 then the vector differences like $p_3 - p_1$ and $p_2 - p_1$ are parallel to the plane. The cross product $(p_2 - p_1) \times (p_3 - p_1)$ will then be perpendicular to the plane (all is well as long as the points p_1 , p_2 , and p_3 form a triangle, so the differences are not parallel). With a point in the plane (for instance, p_1), and a perpendicular vector, we can repeat the process of the previous two exercises.

```
from vectors import *
def plane_equation(p1,p2,p3):
    parallel1 = subtract(p2,p1)
```

```

parallel2 = subtract(p3,p1)
a,b,c = cross(parallel1, parallel2)
d = dot((a,b,c), p1)
return a,b,c,d

```

For example, these are three points from the plane $x + y + z = 3$ from the preceding exercise.

```
>>> plane_equation((1,1,1), (3,0,0), (0,3,0))
(3, 3, 3, 9)
```

The result is (3,3,3,9), meaning $3x + 3y + 3z = 9$, which is equivalent to $x + y + z = 3$. That means we got it right!

EXERCISE

How many total constants a_{ij} are in the following matrix equation? How many equations are there? How many unknowns? Write out the full matrix equation (no dots) and the full system of linear equations (no dots).

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{17} \\ a_{21} & a_{22} & \cdots & a_{27} \\ \vdots & \vdots & \ddots & \vdots \\ a_{51} & a_{52} & \cdots & a_{57} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_7 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_5 \end{pmatrix}$$

Figure 7.32 An abbreviated system of linear equations in matrix form.

SOLUTION

To be clear, we can write out the full matrix equation first.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}$$

Figure 7.33 The un-abbreviated version of the matrix equation above.

In total there are $5 \cdot 7 = 35$ entries in this matrix and 35 “ a_{ij} ” constants on the left-hand side of the equations in the linear system. There are seven unknown variables: x_1, x_2, \dots, x_7 and five equations (one per row of the matrix). You can get the full linear system by carrying out the matrix multiplication.

$$\begin{aligned}
 a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5 + a_{16}x_6 + a_{17}x_7 &= b_1 \\
 a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5 + a_{26}x_6 + a_{27}x_7 &= b_2 \\
 a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 + a_{36}x_6 + a_{37}x_7 &= b_3 \\
 a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5 + a_{46}x_6 + a_{47}x_7 &= b_4 \\
 a_{51}x_1 + a_{52}x_2 + a_{53}x_3 + a_{54}x_4 + a_{55}x_5 + a_{56}x_6 + a_{57}x_7 &= b_5
 \end{aligned}$$

Figure 7.34 The full system of linear equations represented by this matrix equation.

You can see why we avoid this tedious writing with abbreviations!

EXERCISE

Write out the following linear equation without summation shorthand. Geometrically, what does the set of solutions look like?

$$\sum_{i=1}^3 x_i = 1$$

SOLUTION

The left-hand side of this equation is a sum of terms of the form x_i for i ranging from 1 to 3. That gives us $x_1 + x_2 + x_3 = 1$. This is the standard form of a linear equation in three variables, so its solutions form a plane in 3D space.

EXERCISE

Sketch three planes, none of which are parallel, but which do not have a single point of intersection. (Better yet, find their equations and graph them!)

SOLUTION

Here are three planes $z + y = 0$, $z - y = 0$, and $z = 3$:

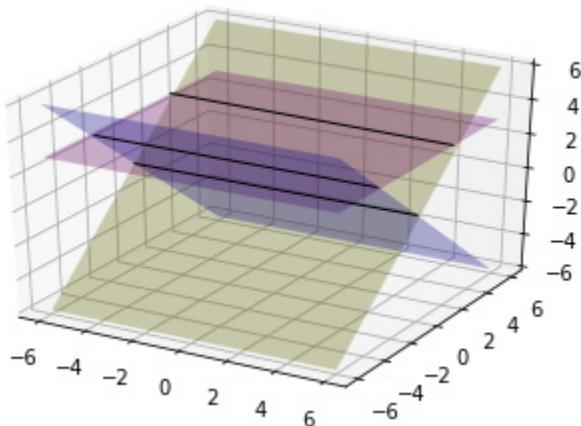


Figure 7.35 Three non-parallel planes which don't share an intersection point.

I've drawn the intersections of the three pairs of planes, which are parallel lines. Since these lines never meet, there is no single point of intersection. This is like the example we saw in chapter 6: three vectors can be linearly dependent even when no pair among them is parallel.

EXERCISE

Suppose we have m linear equations and n unknown variables. What do the following values of m and n say about whether there is a unique solution?

- a.) $m = 2, n = 2$
- b.) $m = 2, n = 7$
- c.) $m = 5, n = 5$
- d.) $m = 3, n = 2$

SOLUTION:

- a.) With two linear equations and two unknowns, there *can* be a unique solution. The two equations represent lines in the plane, and they will intersect at a unique point unless they are parallel.
- b.) With two linear equations and seven unknowns, there cannot be a unique solution. Assuming the 6-dimensional hyperplanes defined by these equations are not parallel, there will be a 5-dimensional space of solutions.
- c.) With five linear equations and five unknowns, there *can* be a unique solution, as long as the equations are independent.
- d.) With three linear equations and two unknowns, there *can* be a unique solution, but it requires some luck. This would mean that the third line happens to pass through the intersection point of the first two lines, which is unlikely but possible.

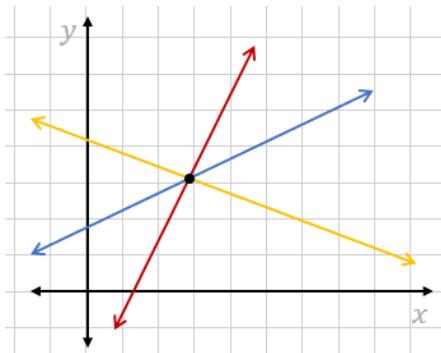


Figure 7.36 Three lines in the plane which happen to intersect at a point.

EXERCISE

Find 3 planes whose intersection is a single point, 3 planes whose intersection is a line, and 3 planes whose intersection is a plane.

SOLUTION

The planes $z - y = 0$, $z + y = 0$, and $z + x = 0$ intersect at the single point $(0,0,0)$. Most randomly selected planes will intersect at a unique point like this.

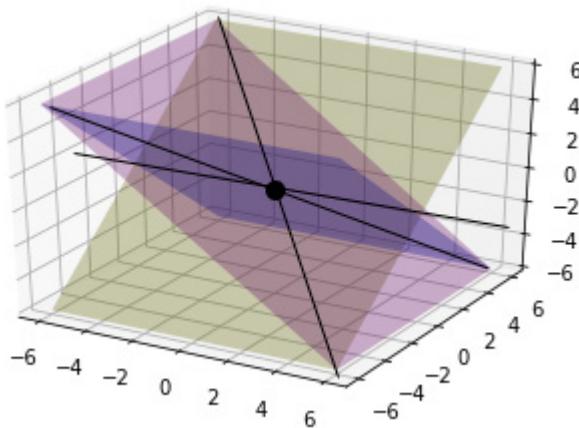


Figure 7.37 Three planes intersecting at a single point.

The planes $z - y = 0$, $z + y = 0$, and $z = 0$ intersect on a line, specifically the x axis. If you play with these equations, you'll find both y and z are constrained to be zero, but x doesn't even appear, so it has no constraints. Any vector $(x,0,0)$ on the x axis is therefore a solution.

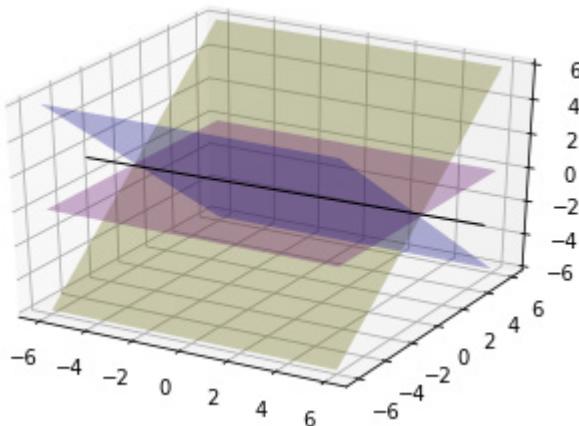


Figure 7.38 Three planes whose intersection points form a line.

Finally, if all three equations represent the same plane, then that whole plane is a set of solutions. For instance, $z - y = 0$, $2z + 2y = 0$, and $3z - 3y = 0$ all represent the same plane.

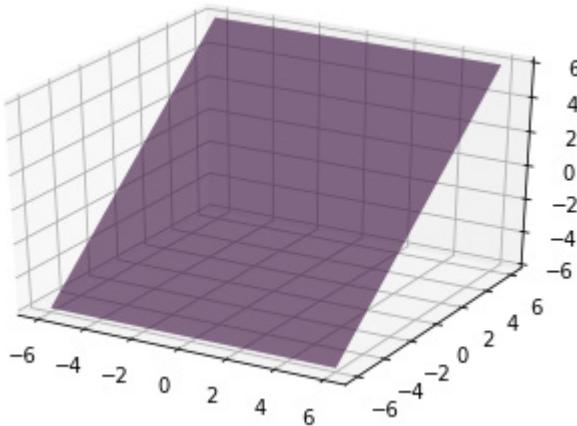


Figure 7.39 Three identical planes overlaid; their set of solutions is the whole plane.

EXERCISE

Without using Python, what is the solution of the system of linear equations in 5D? $x_5 = 3$, $x_2 = 1$, $x_4 = -1$, $x_1 = 0$, and $x_1 + x_2 + x_3 = -2$? Confirm the answer with NumPy.

SOLUTION

Since four of these linear equations specify the value of a coordinate, we know the solution will have the form $(0, 1, x_3, -1, 3)$. We need to do some algebra using the final equation to find out the value of x_3 . Since $x_1 + x_2 + x_3 = -2$, we know $0 + 1 + x_3 = -2$, and x_3 must be -3 . The unique solution point is therefore $(0, 1, -3, -1, 3)$. Converting this system to matrix form, we can solve it with NumPy to confirm we got it right.

```
>>> matrix =
np.array(((0,0,0,0,1),(0,1,0,0,0),(0,0,0,1,0),(1,0,0,0,0),(1,1,1,0,0)))
>>> vector = np.array((3,1,-1,0,-2))
>>> np.linalg.solve(matrix,vector)
array([ 0.,  1., -3., -1.,  3.])
```

MINI-PROJECT

In any number of dimensions, there is an identity matrix which acts as the identity map. That is, when you multiply the n -dimensional identity matrix I by any vector v , you get the same vector v as a result: $Iv = v$.

This means that $Iv = w$ is an easy system of linear equations to solve: one possible answer for v is $v = w$. The idea of this mini-project is that you can start with a system of linear equations $Av = w$ and multiply both sides by another matrix B such that $(BA) = I$. If that is the case, then you have $(BA)v = Bw$ and $Iv = Bw$ or $v = Bw$. In other words, if you have a system $Av = w$, and a suitable matrix B , then Bw is the solution to your system. This matrix B is called the *inverse matrix* of A .

Let's look again at the system of equations we solved in the preceding section.

$$\begin{pmatrix} 1 & 1 & -1 \\ 0 & 2 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -1 \\ 3 \\ 2 \end{pmatrix}$$

Use the NumPy function `numpy.linalg.inv(matrix)`, which returns the inverse of the matrix it is given, to find the inverse of the matrix on the left-hand side of the equation. Then, multiply both sides by this matrix to find the solution to the linear system. Compare your results with the results we got from NumPy's solver.

Hint: you may also want to use NumPy's built-in matrix multiplication routine, `numpy.matmul`, to make computations simpler.

SOLUTION

First, we can compute the inverse of the matrix using NumPy:

```
>>> matrix = np.array(((1,1,-1),(0,2,-1),(1,0,1)))
>>> vector = np.array((-1,3,2))
>>> inverse = np.linalg.inv(matrix)
>>> inverse
array([[ 0.66666667, -0.33333333,  0.33333333],
       [-0.33333333,  0.66666667,  0.33333333],
       [-0.66666667,  0.33333333,  0.66666667]])
```

The product of the inverse matrix with the original matrix gives us the identity matrix, having ones on the diagonal and zeroes elsewhere, albeit with some numerical error.

```
>>> np.matmul(inverse,matrix)
array([[ 1.0000000e+00,  1.11022302e-16, -1.11022302e-16],
       [ 0.0000000e+00,  1.00000000e+00,  0.00000000e+00],
       [ 0.0000000e+00,  0.00000000e+00,  1.00000000e+00]])
```

The trick is to multiply both sides of the matrix equation by this inverse matrix. Here I've rounded the values in the inverse matrix for the sake of readability. We already know that the first product on the left is a matrix and its inverse, so we can simplify accordingly.

$$\underbrace{\begin{pmatrix} 0.667 & -0.333 & 0.333 \\ -0.333 & 0.667 & 0.333 \\ -0.667 & 0.333 & 0.667 \end{pmatrix} \begin{pmatrix} 1 & -1 & 0 \\ 0 & -1 & -1 \\ 1 & 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}}_{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}} = \begin{pmatrix} 0.667 & -0.333 & 0.333 \\ -0.333 & 0.667 & 0.333 \\ -0.667 & 0.333 & 0.667 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0.667 & -0.333 & 0.333 \\ -0.333 & 0.667 & 0.333 \\ -0.667 & 0.333 & 0.667 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}$$

Figure 7.40 Multiplying both sides of the system by the inverse matrix, and simplifying.

This gives us an explicit formula for the solution (x,y,z) , all we need to do is carry out the matrix multiplication. It turns out `numpy.matmul` also works for matrix-vector multiplication:

```
>>> np.matmul(inverse, vector)
array([-1.,  3.,  3.])
```

This is the same as the solution we got earlier from the solver.

7.4 Changing basis by solving linear equations

The notion of linear independence of vectors is clearly related to the notion of independence of linear equations. The connection comes from the fact that solving a system of linear equations is the equivalent of re-writing vectors in a different basis. Let's explore what this means in 2D.

When we write coordinates for a vector like $(4,3)$, we are implicitly writing the vector as a linear combination of the standard basis vectors:

$$(4,3) = 4e_1 + 3e_2.$$

In the last chapter, you learned that the standard basis consisting of $e_1 = (1,0)$ and $e_2 = (0,1)$ is not the only basis available. For instance, a pair of vectors like $u_1 = (1,1)$ and $u_2 = (-1,1)$ form a basis for \mathbb{R}^2 . As any 2D vector can be written as a linear combination of e_1 and e_2 , so can any 2D vector be written as a linear combination of this u_1 and u_2 . For some c and d we can make the following equation true, but it's not immediately obvious what the values of c and d are.

$$c \cdot (1,1) + d \cdot (-1,1) = (4,2)$$

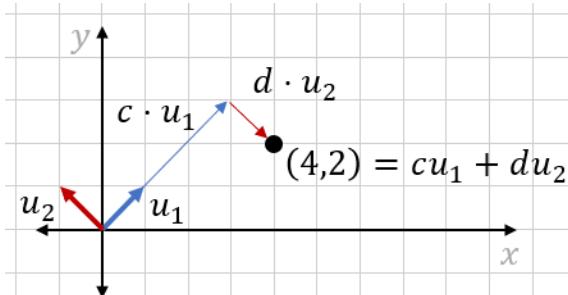


Figure 7.41 Writing $(4,2)$ as a linear combination of $u_1 = (1,1)$ and $u_2 = (-1,1)$.

As a linear combination, this equation is equivalent to a matrix equation, namely:

$$\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$$

This, too is a system of linear equations! In this case, the unknown vector is written (c,d) rather than (x,y) , and the linear equations hidden in the matrix equation are $c - d = 4$ and $c + d = 2$. There is a 2D space of vectors (c,d) that define different linear combinations of u_1 and u_2 , but only one will satisfy these two equations simultaneously.

Any choice of the pair (c,d) defines a different linear combination. As an example, let's look an arbitrary value of (c,d) , say $(c,d) = (3,1)$. The vector $(3,1)$ doesn't live in the same vector space as u_1 and u_2 -- it lives in a vector space of (c,d) pairs, each of which describe a different linear combination of u_1 and u_2 . The point $(c,d) = (3,1)$ describes a specific linear combination in our original 2D space: $3u_1 + 1u_2$ gets us to the point $(x,y) = (2,4)$.

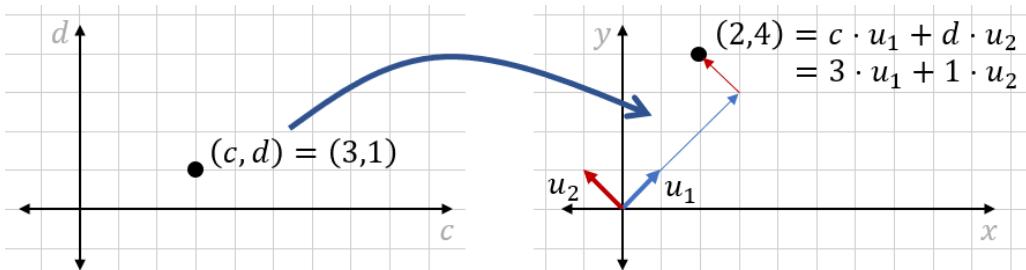


Figure 7.42 There is a 2D space of values of (c,d) . In particular $(c,d) = (3,1)$ yields the linear combination $3u_1 + 1u_2 = (2,4)$.

Recall that we trying to make $(4,2)$ as a linear combination of u_1 and u_2 , so this wasn't the linear combination we were looking for. For $cu_1 + du_2$ to equal $(4,2)$, we need to satisfy $c - d = 4$ and $c + d = 2$, as we saw above. Let's draw the system of linear equations in the c,d -

plane. Visually, we can tell that $(3, -1)$ is a point that satisfies both $c + d = 2$ and $c - d = 4$. This gives us the pair of scalars to use in a linear combination to make $(4, 2)$ out of u_1 and u_2 .

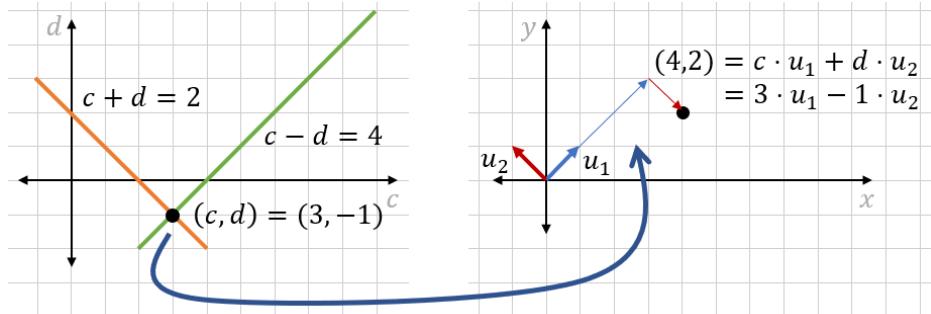


Figure 7.43 The point $(c,d) = (3,-1)$ satisfies both $c + d = 2$ and $c - d = 4$. Therefore, it describes the linear combination we were looking for.

Now we can write $(4,2)$ as a linear combination of two different pairs of basis vectors: $(4,2) = 4e_1 + 2e_2$ and $(4,2) = 3u_1 - 1u_2$. Remember, the coordinates $(4,2)$ are exactly the scalars in the linear combination $4e_1 + 4e_2$. If we had drawn our axes differently, u_1 and u_2 could just as well have been our standard basis; our vector would be $3u_1 - u_2$ and we would say its coordinates were $(3,1)$. To emphasize that coordinates are determined by our choice of basis, we can say that the vector has coordinates $(4,2)$ with respect to the standard basis, but it has coordinates $(3,-1)$ with respect to the basis consisting of u_1 and u_2 .

Finding the coordinates of a vector in a different basis is an example of a computational problem that is really a system of linear equations in disguise. It's an important example because every system of linear equations can be thought of in this way. Let's try another example, this time in 3D, to see what I mean.

7.4.1 Solving a 3D example

Let's start by writing down an example of a system of linear equations in 3D, and then we'll work on interpreting it. Instead of a 2-by-2 matrix and a 2D vector, we can start with a 3-by-3 matrix and a 3D vector.

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & -1 & -1 \\ 1 & 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ -7 \end{pmatrix}$$

The "unknown" here is a 3D vector; we need to find three numbers to identify it. Doing the matrix multiplication, we can break this up into three equations.

$$1 \cdot x - 1 \cdot y + 0 \cdot z = 1$$

$$0 \cdot x - 1 \cdot y - 1 \cdot z = 3$$

$$1 \cdot x + 0 \cdot y + 2 \cdot z = -7$$

We'll call this a system of three linear equations in three unknowns, and we'll call $ax + by + cz = d$ the standard form for a linear equation in 3D. In the next section, we'll look at the geometric interpretation for 3D linear equations. (It turns out they represent *planes* in 3D, as opposed to lines in 2D.)

For now, let's look at this system as a linear combination with coefficients to be determined. The matrix equation above is equivalent to the following.

$$x \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + y \begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix} + z \begin{pmatrix} 0 \\ -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ -7 \end{pmatrix}$$

Solving this equation is equivalent to asking the question, "what linear combination of $(1,0,1)$, $(-1,-1,0)$, and $(0,-1,2)$ yields the vector $(1,3,-7)$?" This is harder to picture than the 2D example, and it is harder to compute the answer by hand as well. Fortunately, we know NumPy can handle systems of linear equations in three unknowns, we simply pass a 3-by-3 matrix and 3D vector as inputs to the solver.

```
>>> import numpy as np
>>> w = np.array((1,3,-7))
>>> a = np.array(((1,-1,0),(0,-1,-1),(1,0,2)))
>>> np.linalg.solve(a,w)
array([ 3.,  2., -5.])
```

The values that solve our linear system are therefore $x = 3$, $y = 2$, and $z = -5$. In other words, these are the coefficients that build our desired linear combination. We can say that the vector $(1,3,-7)$ has coordinates $(3,2,-5)$ in the basis $(1,0,1)$, $(-1,-1,0)$, $(0,-1,2)$.

The story is the same in higher dimensions: as long as it is possible to do so, we can write a vector as a linear combination of other vectors by solving a corresponding system of linear equations. But, it's not always possible to write a linear combination, and not every system of linear equations has a unique solution, or even a solution at all. The question of whether a collection of vectors forms a basis is computationally equivalent to the question of whether a system of linear equations has a unique solution.

This profound connection is a good place to bookend Part 1, focused on linear algebra. There will be plenty of more linear algebra nuggets throughout the book, but they will be even more useful when we pair them with the core topic of Part 2: calculus.

7.4.2 Exercises

EXERCISE

How can you write the vector $(5,5)$ as a linear combination of $(10,1)$ $(3,2)$?

SOLUTION

This is equivalent to asking what numbers a and b satisfy the equation:

$$a \begin{pmatrix} 10 \\ 1 \end{pmatrix} + b \begin{pmatrix} 3 \\ 2 \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}$$

or what vector (a,b) satisfies the matrix equation:

$$\begin{pmatrix} 10 & 3 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}$$

We can find a solution with NumPy.

```
>>> matrix = np.array(((10,3),(1,2)))
>>> vector = np.array((5,5))
>>> np.linalg.solve(matrix,vector)
array([-0.29411765,  2.64705882])
```

This means the linear combination (which you can check!) is as follows.

$$-0.29411765 \cdot \begin{pmatrix} 10 \\ 1 \end{pmatrix} + 2.64705882 \cdot \begin{pmatrix} 3 \\ 2 \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}$$

EXERCISE

Write the vector $(3,0,6,9)$ as a linear combination of the vectors $(0,0,1,1)$, $(0,-2,-1,-1)$, $(1, -2, 0, 2)$, and $(0,0,-2,1)$.

SOLUTION

The linear system to solve is:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & -2 & -2 & 0 \\ 1 & -1 & 0 & -2 \\ 1 & -1 & 2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ 6 \\ 9 \end{pmatrix}$$

where the columns of the 4-by-4 matrix are the vectors we want to build the linear combination with. NumPy gives us the solution to this system:

```
>>> matrix = np.array(( (0, 0, 1, 0), (0, -2, -2, 0), (1, -1, 0, -2), (1, -1,
```

```

2, 1)))
>>> vector = np.array((3,0,6,9))
>>> np.linalg.solve(matrix,vector)
array([ 1., -3.,  3., -1.])

```

This means that the linear combination is

$$1 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} - 3 \cdot \begin{pmatrix} 0 \\ -2 \\ -1 \\ -1 \end{pmatrix} + 3 \cdot \begin{pmatrix} 1 \\ -2 \\ 0 \\ 2 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ 6 \\ 9 \end{pmatrix}$$

7.5 Summary

In this chapter you learned:

- How to model objects in a 2D video game as polygonal shapes built out of line segments.
- Given two vectors u and v , the points of the form $u + tv$ for any real number t lie on a straight line. In fact, any line can be described with this formula.
- Given real numbers a , b , and c , where at least one of a and b is non-zero, the points (x,y) in the plane satisfying $ax + by = c$ lie on a straight line. This is called the *standard form* for the equation of a line, and any line can be written in this form for some choice of a , b , and c . Equations for lines are called *linear equations*.
- Finding the point where two lines intersect in the plane is equivalent to finding the values (x,y) which simultaneously satisfy two linear equations. A collection of linear equations that we seek to solve simultaneously is called a *system of linear equations*.
- Solving a system of two linear equations is equivalent to finding what vector can be multiplied by a known 2-by-2 matrix to yield a known vector.
- NumPy has a built-in function `numpy.linalg.solve` which takes a matrix and a vector and solves the corresponding system of linear equations automatically, if possible.
- Some systems of linear equations cannot be solved. For instance, if two lines are parallel, they may have no intersection points or infinitely many (which would mean they are the same line). This means there is no (x,y) value that satisfies both lines' equations simultaneously. A matrix representing such a system is called *singular*.
- Planes in 3D are the analogies of lines in 2D. They are the sets of points (x,y,z) satisfying equations of the form $ax + by + cz = d$.
- Two non-parallel planes in 3D intersect at infinitely many points, specifically the set of points they share form a 1D line in 3D. Three planes can have a unique point of intersection, which can be found by solving the system of three linear equations representing the planes.
- Lines in 2D and planes in 3D are both cases of *hyperplanes*, sets of points in n -dimensions which are solutions to a single linear equation.

- In n -dimensions, you need a system of at least n linear equations to find a unique solution. If you have exactly n linear equations and they have a unique solution, they are called *independent equations*.
- Figuring out how to write a vector as a linear combination of a given set of vectors is computationally equivalent to solving a system of linear equations. If the set of vectors is a basis for the space, this is always possible.

A

Loading and Rendering 3D Models with OpenGL and PyGame

Beyond chapter 3, when we start writing programs that transform and animate graphics, I begin to use OpenGL and PyGame instead of Matplotlib. This appendix gives an overview of how to set up a game loop in PyGame and render 3D models in successive frames. The culmination is an implementation of a `draw_model` function, which renders a single image of a 3D model like the teapot we use in chapter 4.

The goal of `draw_model` is to encapsulate the library-specific work, so you don't have to spend a lot of time wrestling with OpenGL. But, if you want to understand how the function works, feel free to follow along with this appendix, and play with the code yourself. Let's start with our octahedron from chapter 3 and recreate it with PyOpenGL and PyGame.

A.1 Recreating the octahedron from Chapter 3

The first step to get working with the PyOpenGL and PyGame libraries is to install them. I recommend using pip, as follows:

```
> pip install PyGame
> pip install PyOpenGL
```

The first thing I'll show you is how to use these libraries to recreate work we've already done: rendering a simple 3D object.

In a new Python file called `octahedron.py`, we start with a bunch of imports. The first few come from the two new libraries, and the rest should be familiar from chapter 3. In particular, we'll continue to use all of the 3D vector arithmetic functions we've already built, organized in the file `vectors.py` in the source code.

```
import pygame
from pygame.locals import *
from OpenGL.GL import *
from OpenGL.GLU import *
import matplotlib.cm
from vectors import *
from math import *
```

While OpenGL does have automatic shading capabilities, we'll continue to use our shading mechanism from last chapter. We can use a blue color map from Matplotlib to compute colors of shaded sides of the octahedron.

```
def normal(face):
    return(cross(subtract(face[1], face[0]), subtract(face[2], face[0])))

blues = matplotlib.cm.get_cmap('Blues')

def shade(face,color_map=blues,light=(1,2,3)):
    return color_map(1 - dot(unit(normal(face)), unit(light)))
```

Next, we have to specify the geometry of the octahedron and the light source. Again, this is the same as in chapter 3.

```
light = (1,2,3)
faces = [
    [(1,0,0), (0,1,0), (0,0,1)],
    [(1,0,0), (0,0,-1), (0,1,0)],
    [(1,0,0), (0,0,1), (0,-1,0)],
    [(1,0,0), (0,-1,0), (0,0,-1)],
    [(-1,0,0), (0,0,1), (0,1,0)],
    [(-1,0,0), (0,1,0), (0,0,-1)],
    [(-1,0,0), (0,-1,0), (0,0,1)],
    [(-1,0,0), (0,0,-1), (0,-1,0)],
]
```

Now it's time for some unfamiliar territory. We're going to show the octahedron as a PyGame game window, which requires a few lines of boilerplate. Here, we start the game, set the window size in pixels, and tell PyGame to use OpenGL as the graphics engine.

```
pygame.init()
display = (400,400)
window = pygame.display.set_mode(display, DOUBLEBUF|OPENGL) ① ②
```

- ① We will ask PyGame to show our graphics in a 400-by-400 pixel window.
- ② Let PyGame know that we are using OpenGL for our graphics; indicate that we want to use a built-in optimization called double-buffering (which is not important to understand for our purposes).

In our simplified example in Chapter 3, we drew the octahedron from the perspective of someone looking from a point far up the z-axis. We computed which triangles should be visible to such an observer, and projected them to 2D by removing their z-axis. OpenGL has built-in functions to configure our perspective even more precisely.

```

gluPerspective(45, 1, 0.1, 50.0) ①
glTranslatef(0.0,0.0, -5)        ②
glEnable(GL_CULL_FACE)          ③
glEnable(GL_DEPTH_TEST)         ④
glCullFace(GL_BACK)             ⑤

```

- ① Describe our perspective looking at the scene: we will have a 45 degree viewing angle and an aspect ratio of 1. This means the vertical units and the horizontal units will display as having the same size. As an optimization, the numbers 0.1 and 50.0 put limits on the z-coordinates that will be rendered – no objects further than 50.0 units from the observer or closer than 0.1 units will show up.
- ② We observe the scene from 5 units up the z-axis, meaning we move the scene down by vector (0,0,-5).
- ③ Enable an OpenGL option that automatically hides polygons oriented away from the viewer, saving us some work we already did in chapter 3.
- ④ Enable an OpenGL option that automatically hides polygons which are facing us but behind other polygons. For the sphere this wasn't a problem, but for more complex shapes it could be.

Finally, we can implement the main code that will draw our octahedron. Since our eventual goal will be animating objects, we'll actually write code that draws it over and over repeatedly. These successive drawings, like frames of a movie, will show the same octahedron over time. And, like any video of any stationary object, the result will be indistinguishable from a static picture. To render a single frame, we'll loop through the vectors, decide how to shade them, draw them with OpenGL, and update the frame with PyGame. Inside of an infinite while-loop, this process can be automatically repeated as fast as possible as long as the program runs.

```

clock = pygame.time.Clock()                                ①
while True:
    for event in pygame.event.get():                      ②
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    clock.tick()                                         ③
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glBegin(GL_TRIANGLES)                                ④
    for face in faces:
        color = shade(face,blues,light)
        for vertex in face:
            glColor3fv((color[0], color[1], color[2]))  ⑤
            glVertex3fv(vertex)                          ⑥
    glEnd()
    pygame.display.flip() #7

```

- ① Initialize a clock to measure the advancement of time for PyGame.
- ② In each iteration, check the events PyGame has received and quit if the user has closed the window.
- ③ Indicate to the clock that time should elapse.
- ④ Instruct OpenGL that we are about to draw triangles.
- ⑤ For each vertex of each face (triangle), set the color based on the shading.
- ⑥ Specify the next vertex of the current triangle.
- ⑦ Indicate to PyGame that the newest frame of the animation is ready, and make it visible.

Running this, we see a 400 pixel by 400 pixel PyGame window appear, containing a figure that looks like the one from chapter 3.

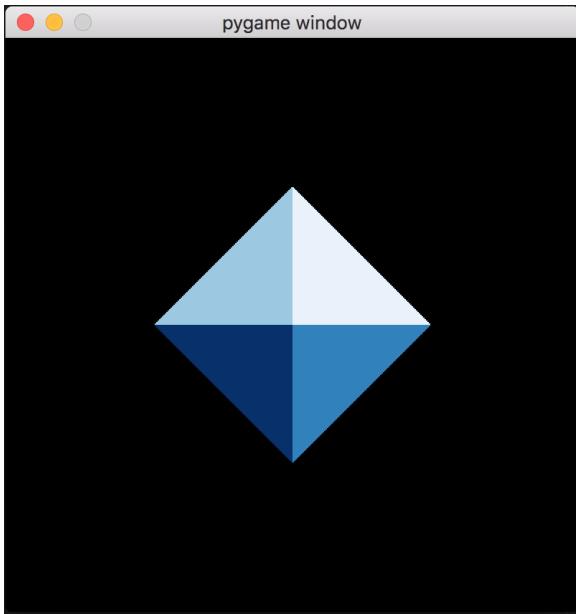


Figure A.1 The octahedron rendered in a PyGame window.

If you want to prove that something more interesting is happening, you can include the following line at the end of the `while True` loop:

```
print(clock.get_fps())
```

This prints instantaneous quotes of the rate (in frames per second, or “fps”) at which PyGame is rendering and re-rendering the octahedron. For a simple animation like this, PyGame should approximately reach its default maximum frame-rate of 60 frames per second.

But what’s the point of rendering so many frames if nothing changes? Once we include a vector transformation with each frame, we will see the octahedron move in various ways. For now, we can cheat by moving the “camera” with each frame instead of actually moving the octahedron.

A.2 Changing our perspective

The `glTranslatef` function above tells OpenGL the position from which we want to see the 3D scene we’re rendering. Similarly, there is a `glRotatef` function which lets us change the angle at which we observe the scene. Calling `glRotatef(theta, x, y, z)` rotates the whole scene by an angle `theta` about an axis specified by the vector `(x,y,z)`.

Let me clarify what I mean by “rotating by an angle about an axis.” You can think of the familiar example of the Earth rotating in space. The earth rotates by 360 degrees every day, or 15 degrees every hour. The *axis* is the invisible line that the Earth rotates around: it

passes through the north and south pole — the only two points which aren't rotating. For the earth, the axis of rotation is not directly upright, rather it is tilted by 23.5 degrees.

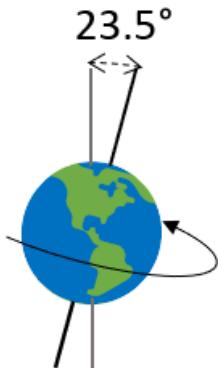


Figure A.2 A familiar example of an object rotating about an axis, the Earth's axis of rotation is tilted at 23.5 degrees relative to its orbital plane.

The vector $(0,0,1)$ points along the z-axis, so calling `glRotatef(30,0,0,1)` would rotate the scene by 30 degrees about the z-axis. Likewise, `glRotatef(30,0,1,1)` would rotate the scene by 30 degrees, but instead about the axis $(0,1,1)$, which is 45 degrees tilted between the y and z axes. If we call `glRotatef(30,0,0,1)` or `glRotatef(30,0,1,1)` after `glTranslatef(...)` in the octahedron code, we see the octahedron rotated.

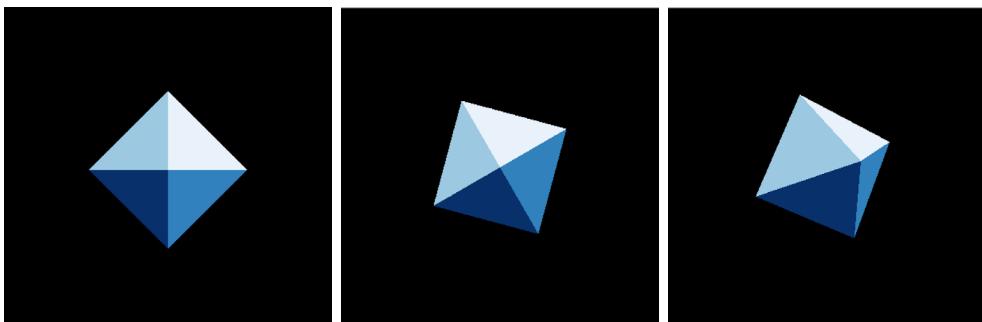


Figure A.3 The octahedron, as seen from three different, rotated perspectives using the `glRotatef` function from OpenGL.

Notice that the shading of the four visible sides of the octahedron has not changed. This is because none of the vectors have changed: the vertices of the octahedron and the light source are all the same. We have only changed the position of the “camera” relative to the octahedron. When we actually change the position of the octahedron, we'll see the shading change too.

To animate the rotation of the cube, we can call `glRotate` with a small angle every frame. For instance, if PyGame draws the octahedron at about 60 frames per second, and we call `glRotatef(1,x,y,z)` every frame, the octahedron will rotate about 60 degrees every second about the axis (x,y,z) . So, adding `glRotatef(1,1,1,1)` within the infinite while loop right before `glBegin` causes the octahedron to rotate by a degree per frame about an axis in the direction $(1,1,1)$.

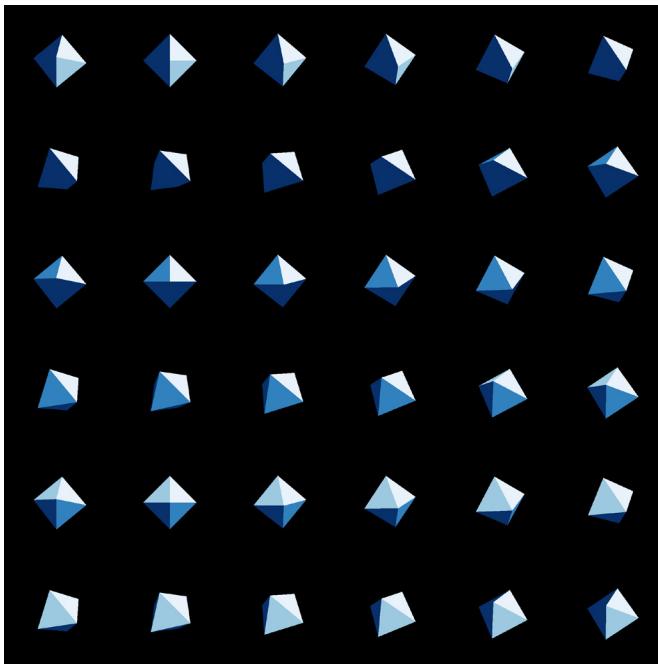


Figure A.4 Every 10th frame of our octahedron rotating at one degree per frame. After 36 frames, the octahedron has completed a full rotation.

This rotation rate is only accurate if PyGame draws exactly 60 frames per second. In the long run this may not be true: if a complex scene requires more than a sixtieth of a second to compute all vectors and draw all polygons, the motion will actually slow down. To keep the motion of the scene constant regardless of the frame rate, we can use PyGame's clock.

Let's say we want our scene to rotate by a full rotation (360 degrees) every 5 seconds. PyGame's clock thinks in milliseconds, which are thousandths of a second. For a thousandth of the time, the angle rotated will be divided by 1000.

```
degrees_per_second = 360./5.
degrees_per_millisecond = degrees_per_second / 1000.
```

The PyGame clock object we created has a `tick()` method, which both advances the clock and returns the number of milliseconds since `tick()` was last called. This gives us a reliable number

of milliseconds since the last frame was rendered, and lets us compute the angle that the scene should be rotated in that time.

```
milliseconds = clock.tick()
glRotatef(milliseconds * degrees_per_millisecond, 1,1,1)
```

Calling `glRotatef` like this every frame will guarantee that the scene rotates exactly 360 degrees every five seconds. In the file “rotate_octahedron.py” in the source code, you can see exactly how this code is inserted.

With the ability to move our perspective over time, we’ve already got better rendering capabilities than we developed in chapter 3. Now, we’ll turn our attention to drawing a more interesting shape than an octahedron or a sphere.

A.3 Loading and rendering the Utah teapot

As we manually identified the vectors outlining a 2D dinosaur in chapter 2, we could manually identify the vertices of any 3D object, organize them into triples representing triangles, and build the surface as a list of triangles. Artists who design 3D models have specialized interfaces for positioning vertices in space and then saving them to files. In this chapter, we’ll use a famous pre-built 3D model: the Utah teapot. The rendering this teapot is the “Hello World” program for graphics programmers: a simple, recognizable example for testing.

The teapot model is saved in the file “teapot.off” in the source code, where the “.off” extension stands for “Object File Format.” This is a plaintext format specifying the polygons making up the surface of a 3D object and the 3D vectors which are vertices of the polygon. The `teapot.off` file looks something like this:

Listing: A schematic of the teapot.off file

```
OFF
1
480 448 926
2
0 0 0.488037
3
0.00390625 0.0421881 0.476326
0.00390625 -0.0421881 0.476326
0.0107422 0 0.575333
...
4 324 306 304 317
4 306 283 281 304
4 283 248 246 281
...
```

- ➊ The first line “OFF” indicates that this file follows the Object File Format.
- ➋ The second line has three numbers: the number of vertices, faces, and edges of the 3D model, in that order.
- ➌ The next 480 lines of this file specify 3D vectors for each of the vertices. Each of these lines has three numbers, which are the x, y, and z coordinates. The order of these vertices matters because it is referenced below!
- ➍ The next 448 lines of this file specify the 448 faces of the model. The first number of each line tells us what kind of polygon the face is. The number “3” would indicate a triangle, “4” a quadrilateral, “5” a pentagon, and so on. Most of the teapot’s faces turn out to be quadrilaterals. The next numbers on each line tell us the indices of the vertices from above that form the corners of the given polygon.

In the file teapot.py in the source code, I built functions `load_vertices()` and `load_polygons()` that load the vertices and faces (polygons) from the `teapot.off` file. The first returns a list of 440 vectors which are all the vertices for the model. The second returns a list of 448 lists: each one contains vectors which are vertices of one of the 448 polygons making up the model. Finally, I include a third function `load_triangles()` which breaks up the polygons with four or more vertices so our entire model is built out of triangles.

I've left it as a mini-project for you to dig deeper into my code or to try to load the `teapot.off` file as well. For now I'll continue with the triangles loaded by my `teapot.py` file so we can get to drawing and playing with our teapot more quickly. The other step I'll skip is organizing the PyGame and OpenGL initialization into a function so that we don't have to repeat it every time we're drawing a model. In `draw_model.py` you'll find the following function:

```
def draw_model(faces, color_map=blues, light=(1,2,3)):
    ...
```

It takes in the faces of a 3D model (assumed to be correctly oriented triangles), a color map for shading, and a vector for the light source, and draws the model accordingly. Like our code to draw the octahedron, it draws whatever model is passed in over and over in a loop. In `draw_teapot.py`, I put these together as follows

Listing: `draw_teapot.py` loads the teapot triangles and passes them to `draw_model`

```
from teapot import load_triangles
from draw_model import draw_model

draw_model(load_triangles())
```

The result is an overhead view of a teapot: you can see the circular lid, the handle on the left, and the spout on the right. If we include suitable calls to `glRotatef` inside `draw_model`, we can see the same teapot from different angles.

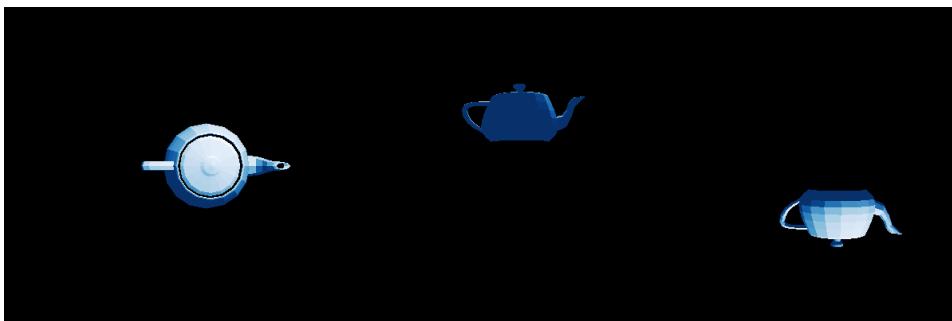


Figure A.5 Rendering of the original teapot model (leftmost) and after rotating our perspective with `glRotatef` (right two)

NOTE

Notice that in the second image, we have rotated our perspective so that we're looking at the "dark side" of the teapot – that is, the light source is pointed at the other side of the teapot from where we are looking. If the teapot itself were to have rotated, we'd see the visible side of the teapot illuminated. When we successfully move the teapot in 3D, we should see the shading update accordingly.

Now that we can render a shape that's more interesting than a simple geometric figure, it's time to play! When you return to your regularly scheduled programming in chapter 4, you'll learn about mathematical transformations you can do on all of the vertices of the teapot at once to move and distort it in 3D space. I've also left you some exercises below if you want to do some guided exploration of the rendering code.

A.4 Exercises

EXERCISES

Modify the `draw_model` function to be able to display the input figure from any rotated perspective. Specifically, give the `draw_model` function a keyword argument `glRotatefArgs` which provides a tuple of four numbers corresponding to the four arguments of `glRotatef`. With this extra information, add an appropriate call to `glRotatef` within the body of `draw_model` to execute the rotation.

SOLUTION

In the source code, see `draw_model.py` for the solution and `draw_teapot_glrotatef.py` for example usage.

EXERCISE

If we call `glRotatef(1, 1, 1, 1)` every frame, how many seconds does it take for the scene to complete a full revolution?

SOLUTION

The answer depends on the framerate. This call to `glRotatef` rotates the perspective by one degree each frame. At 60 frames per second, it would rotate 60 degrees per second, and complete a full rotation of 360 degrees in 6 seconds.

MINI-PROJECT

Implement the `load_triangles()` function from above, which loads the teapot from the `teapot.off` and produces a list of triangles in Python. Each triangle should be specified by a three 3D vectors. Then, pass your result to `draw_model()` as above and confirm that you see the same result.

SOLUTION

In the source code, you can find `load_triangles()` implemented in the file `teapot.py`. As a hint, you can turn the quadrilaterals into pairs of triangles by connecting their opposite vertices.

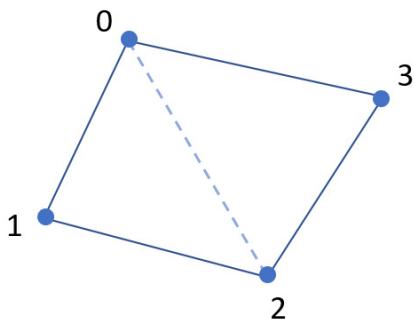


Figure A.6 Indexing four vertices of a quadrilateral, two triangles are formed by vertices 0, 1, 2 and 0, 2, 3 respectively.

MINI-PROJECT

Animate the teapot by changing the arguments to `gluPerspective` and `glTranslatef`. This will help you visualize the effects of each of the parameters.

SOLUTION

In `animated_octahedron.py` in the source code, an example is given for rotating the octahedron by $360/5 = 72$ degrees per second by updating the angle parameter of `glRotatef` every frame. You can try similar modifications yourself, with either the teapot or the octahedron.
