Matrix Methods and Programs

compiled on Thursday 2nd October, 2025

MATRIX METHODS & PROGRAMS

David F. Gleich

PREFACE

This document is a work-in-progress set of notes associated with CS515 at Purdue University. I hope to turn it into a book at some point. There are typos. There are mistakes. Please let me know if you find them. It's associated with the lectures at CS515 at Purdue and I am extremely grateful to all the questions students have asked over the years that have led me to develop this material.

As a warning, there are portions of the material I have based on notes by others. I believe I have documented these in my original notes and I hope that these citations and references made the transition to this integrated document. (I will be going through and checking!) These will likely be revised before I would seek publication of the book, but if you see material that you feel merits a citation to your own work, please do let me know!

The typography style and sectioning style is based on a combination of Nick Trefethen (Numerical Linear Algebra and Pseudospectra with Embree) and Edward Tufte.

The idea for writing is to illustrate the relevance of the analogies

computers		mathematics
&	↔	&
programs		matrices
&	↔	&
codes		algebra
&	↔	&
systems		analysis

or put differently, we use

matrices mathematics algebra analysis
to
describe study improve understand
the behavior of
computers programs codes systems
on interesting problems

I'm not happy with the title yet, but it's the best I have so far.

Another part of this book is that I wish to use better terminology besides what I affectionately call "dead white guy names." The terminology of the field is riddled with methods named after people

Gaussian elimination Jacobi's method Richardson's method
Vandermonde matrix Krylov subspace.

Mostly, these people are dead. And white men. (Of course there are exceptions.) Yet Gaussian elimination was known to the Chinese millenia before Gauss. So the names tend to stick to the person who managed to popularize the ideas rather than the ones who necessarily discovered them. This has been widely observed.[1] I like the following phrasing:

Methods and ideas tend to be named after the
last people to discover them.

The real issue is that these terms contribute to a culture of jargon, dogma, and arcana that isn't helpful to learn, understand, and compare ideas. Towards that end, another goal with this book is to attempt to give helpful names to some of these methods. For example, the Krylov subspace is really a subspace of matrix powers. Why isn't the term "matrix powers subspace" better? If length is an issue, "power subspace" is even shorter. It's also closely related to a monomial basis of polynomials, so mononomial basis would be another possible choice. These are all discussed throughout the text and we have lecture A that lists better phrases to consider using along with the current term.

[1] From Wikipedia "Stigler's law of eponymy, proposed by University of Chicago statistics professor Stephen Stigler in his 1980 publication Stigler's law of eponymy, states that no scientific discovery is named after its original discoverer." And of course, Stigler also wasn't the first one to mention this idea.

CONTENTS

PART I **MATRIX PROBLEMS & STRUCTURE**

PART II **SIMPLE ITERATIVE ALGORITHMS**

PART III **FINITELY TERMINATING ALGORITHMS**

PART IV **ANALYSIS**

PART V **SUBSPACE METHODS**

PART VI **ADVANCED PROBLEMS**

PART VII **EIGENVALUE ALGORITHMS**

PART VIII **APPLICATION DERIVATIONS**

APPENDIX

MISCELLANEOUS PROBLEMS 0

I need to find a place to put these.

EXERCISES

1. Let $\boldsymbol{C}$ be an $n \times k$ matrix with $n \geq k$. (i) Show that $\boldsymbol{A} = \left[\begin{smallmatrix} \boldsymbol{C} & \boldsymbol{I} \\ -\boldsymbol{I} & \boldsymbol{C}^T \end{smallmatrix}\right]$ is invertible. Let $\mathbf{b} = \left[\begin{smallmatrix} \mathbf{f} \\ \mathbf{g} \end{smallmatrix}\right]$, give an algorithm to solve $\boldsymbol{A}\left[\begin{smallmatrix} \mathbf{x} \\ \mathbf{y} \end{smallmatrix}\right] = \mathbf{b}$. (iii) Give an explicit form for the inverse (this may involve the inverse of other matrices.) This problem was inspired by algebra in my paper on Erasure Coded Eigenvalue problems. (Hint, show this for $\boldsymbol{C} = 0$, then try $\boldsymbol{C}$ as a rank-1 matrix $\boldsymbol{C} = \mathbf{u}\mathbf{v}^T$.)
2. Let $\boldsymbol{A}$ be an $n \times k$ matrix with $n \geq k$. Show that $\boldsymbol{A}^T(\boldsymbol{I} + \boldsymbol{A}\boldsymbol{A}^T)^{-1} - (\boldsymbol{I} + \boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T = 0$. Hint: use the rank-k update formula on the large inverse. Alternative hint: try when $\boldsymbol{A}$ is a rank-1 matrix $\boldsymbol{A} = \mathbf{u}\mathbf{v}^T$.
3. Let $\boldsymbol{A}$ be an $n \times k$ matrix with $n \geq k$. Show that $(\boldsymbol{I} + \boldsymbol{A}\boldsymbol{A}^T)^{-1} + \boldsymbol{A}(\boldsymbol{I} + \boldsymbol{A}^T mA)^{-1}\boldsymbol{A}^T = \boldsymbol{I}$. Hint: use the rank-k update formula on the large inverse. Alternative hint: try when $\boldsymbol{A}$ is a rank-1 matrix $\boldsymbol{A} = \mathbf{u}\mathbf{v}^T$.
4. Let $\boldsymbol{A}$ be a $k \times k$ matrix. Give a closed form expression for the matrix

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{I} \\ \boldsymbol{I} & 0 \end{bmatrix}^{-1}.$$

MATRIX PROBLEMS & STRUCTURE

I

WHAT IS A MATRIX? 1

Ask any American someone on the street of about the same age as I am "What is a matrix" and they are likely to look just a bit confused. They are probably thinking you meant to ask "What is The Matrix", in reference to the 1999-ish movie staring Keanu Reeves.

Now, ask someone in a engineering or science school "What is a matrix" and you'll probably get some of these answers:

1. A matrix is a 2-dimensional array, or table, of numbers.
2. A matrix is a linear transformation.
3. A matrix is a coordinate representation of a linear transformation between vector spaces.
4. A matrix is an element of $\mathbb{R}^{m,n}$ or $\mathbb{C}^{m,n}$ (or some other field such as $\mathbf{F}_2$, which arises frequently in cryptography).
5. It's a table of numbers where linear algebraic manipulations make sense.
6. I know it when I see it.
7. A matrix stores vectors.
8. Something used to simplify mathematical calculations.

These are all fantastic answers. The first one is right out of Wikipedia[1]

> *In mathematics, a matrix (pl.: matrices) is a rectangular array or table of numbers, symbols, or expressions, with elements or entries arranged in rows and columns [. . .]*

Moreover, it is a key fact in a linear algebra class that a linear transformation, or more precisely, a linear transformation between finite dimensional vector spaces, can be represented as a m by n matrix. The space of these matrices is often written as $\mathbb{R}^{m,n}$ or $\mathbb{C}^{m,n}$ when the elements come from the real ($\mathbb{R}$) or complex ($\mathbb{C}$) number systems but it can be any field or even – with some changes – elements from a ring. There is actually a second part of the Wikipedia definition:

> *[. . .] with elements or entries arranged in rows and columns, which is used to represent a mathematical object or property of such an object.*

This is a key for this book. Not just *any* table of numbers gives rise to a matrix. The numbers must have some relationship to a mathematical representation. Wikipedia, however, does not quite capture the nature of the mathematical representation.

The definition of a matrix we will use in this book is:

Learning objectives
1. Realize the difference between a table of data and a matrix.
2. Translate simple problems into matrix equations (such as least squares).
3. Review our notation for matrices, vectors, scalars, etc.

[1] Matrix (mathematics). `https://en.wikipedia.org/wiki/Matrix_(mathematics)`, accessed on 1 August 2024.

A matrix is a table of numbers where linear algebraic manipulations make sense.

This definition implies or subsumes many of the others answers.[2]

The ones that are even more general are:

- A matrix is a 2-dimensional array, or table, or numbers.
- A matrix is an element of $\mathbb{R}^{m,n}$ or $\mathbb{C}^{m,n}$ (or some other field such as $\mathbf{F}_2$, which arises frequently in cryptography).

What these miss is the essence of the *mathematics* or *computations* underlying matrices. Let's see an example to make this point clear.

1.1 A STARTING EXAMPLE

Is the following a matrix?

$$\begin{bmatrix} 4 & 9 & 6 & 0 & 4 & 5 & 3 \\ 4 & 9 & 4 & 1 & 9 & 9 & 7 \\ 4 & 9 & 4 & 6 & 0 & 1 & 0 \\ 4 & 9 & 6 & 1 & 7 & 6 & 1 \\ 4 & 9 & 4 & 8 & 7 & 9 & 8 \\ 4 & 9 & 6 & 2 & 3 & 9 & 9 \\ 4 & 9 & 4 & 6 & 0 & 1 & 3 \\ 4 & 9 & 6 & 9 & 4 & 3 & 2 \\ 4 & 9 & 4 & 9 & 0 & 2 & 5 \\ 4 & 9 & 4 & 6 & 0 & 0 & 5 \\ 4 & 9 & 4 & 6 & 0 & 0 & 9 \end{bmatrix}$$

It's an element of $\mathbb{R}^{11,7}$. So under a strict syntactic view and assuming the most general definition of a matrix, we could answer yes. However, this particular set of numbers arises from a table of telephone numbers, expanded by digit into columns. This information has *no linear algebraic structure*: the sum of two phone numbers is not another phone number. Also, it does not represent a linear transformation. We would not expect solving linear systems of equations with this "matrix" to yield interesting answers. Thus, for the purposes of this book, this data is *not a matrix*.

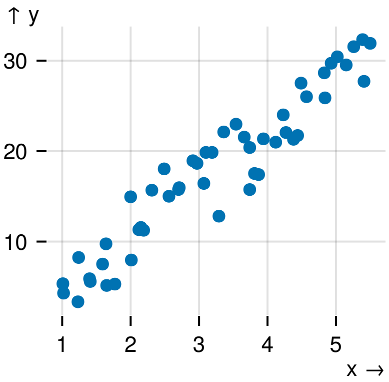
On the other hand, we might argue that there *is* low-rank structure in this matrix. Note that the first two columns are multiplies of each other. In this case, this structure arises because of the shared prefixes of telephone numbers, and thus, it is non algebraic or non mathematical in origin. Simply put, it is a curiosity of how how we choose to represent this information – the phone numbers – as a matrix. We would expect and hope *true structure* to be invariant or agnostic to such decisions.

[2] Please think about this, but I claim this version handles cases 2, 3, 5, 6, 8.

1.2 ANOTHER EXAMPLE

Let's try and answer this question a different perspective and see a few places where matrices arise. This will also help introduce us to our notation in class.

Suppose we have some data and we'd like to fit a linear model to it.



The data[3] for our problem are pairs: $(y_1, x_1), \ldots, (y_N, x_N)$. We can assemble them into a matrix in a few ways

$$X = \begin{bmatrix} y_1 & x_1 \\ y_2 & x_2 \\ \vdots & \\ y_N & x_N \end{bmatrix} \text{ or } X = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \\ x_N & y_N \end{bmatrix} \text{ or } X = \begin{bmatrix} x_1 & x_2 & \ldots & x_N \\ y_1 & y_2 & \ldots & y_N \end{bmatrix} \text{ or } \ldots$$

Are these matrices? Let's keep going before answering this!

Our goal is to find coefficients (c_1, c_2) such that $y = c_2 x + c_1$ is a good fit to the data. There are a few ways that we could measure fit. We will be expendient and insist that for each data point, we want:

$$(y_i - (c_2 x_i + c_1))^2$$

to be as small as possible for all datapoints.[4] This gives us a way to measure how good c_1, c_2 are. Our goal is now to minimize the function:

$$f(c_1, c_2) = \sum_{i=1}^{N} (y_i - (c_2 x_i + c_1))^2.$$

Let us use notation to eliminate the indices.

Let $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{c}$ be the vectors:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \qquad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

[3] These data are

x	y
1.02	4.3
1.01	5.34
1.23	3.34
1.24	8.24
1.4	5.89
1.41	5.59
1.59	7.51
1.65	5.15
1.64	9.75
1.77	5.3
2.01	7.96
2.0	14.95
2.15	11.56
2.12	11.32
2.19	11.25
2.31	15.68
2.49	18.04
2.56	15.01
2.7	15.76
2.71	15.96
2.91	18.94
2.97	18.65
3.07	16.43
3.1	19.86
3.19	19.86
3.29	12.81
3.36	22.12
3.54	22.98
3.66	21.55
3.74	20.4
3.74	15.75
3.81	17.53
3.87	17.41
3.94	21.36
4.12	20.99
4.23	24.01
4.27	22.06
4.38	21.31
4.44	21.73
4.49	27.52
4.57	26.02
4.83	28.66
4.84	25.89
4.93	29.72
5.02	30.43
5.15	29.53
5.26	31.55
5.39	32.33
5.5	31.92
5.41	27.72

then

$$f(\mathbf{c}) = \left\|\mathbf{y} - \mathbf{x}c_2 - \mathbf{e}c_1\right\|^2.$$

Here, the vector $\mathbf{e}$ is just a vector of all ones, and $\|\cdot\|$ is the 2-norm, or Euclidean norm, of a vector:

$$\|\mathbf{z}\| = \sqrt{\sum_{i=1}^{n} |z_i|^2}.$$

Now, let $\boldsymbol{A}$ be the matrix:

$$\boldsymbol{A} = \begin{bmatrix} \mathbf{e} & \mathbf{x} \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \\ 1 & x_N \end{bmatrix}. \tag{1.1}$$

Then we can write our function f as:

$$f(\mathbf{c}) = \left\|\mathbf{y} - \boldsymbol{A}\mathbf{c}\right\|^2.$$

This type of problem is an instance of what is called a *least squares* problem. The data to the problem are encoded into a matrix $\boldsymbol{A}$ and the goal is to produce coefficients $\mathbf{c}$ that constitute a linear relationship.

EXAMPLE 1.1 Hint that there is a matrix. Note that we can reparameterize the line as:

$$y = d_2(x-1) + d_1$$

in which case we have

$$c_2 = d_2, c_1 = d_1 - d_2$$

or

$$\mathbf{c} = \underbrace{\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}}_{=\boldsymbol{T}} \mathbf{d}$$

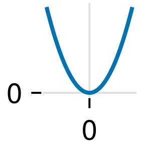
which where $\boldsymbol{T}$ is definitely a matrix! In this case, we'd have:

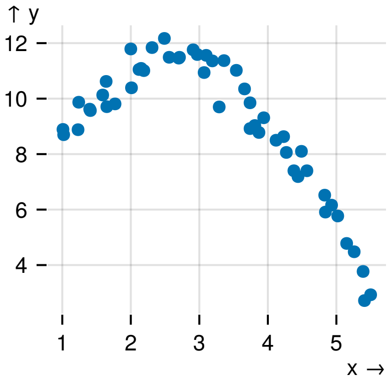
$$\|\mathbf{y} - \boldsymbol{A}\boldsymbol{T}\mathbf{d}\|$$

if we wanted to parameterize in terms of $\boldsymbol{T}$. ◆

Suppose instead we had the data[5] in the following picture

[4] This is a general instances of a squared loss approximation. If we want two values such that $a \approx b$ then with squared loss, we want $(a-b)^2$. These loss functions are often illustrated link this plot:





Then it's clear from the picture that a linear model isn't appropriate. We can still solve the problem we just posed, but there may be other models of our data that are appropriate. Such as a quadratic:

$$c_3 x^2 + c_2 x + c_1.$$

By going through the same type of steps, we can write the resulting problem in terms of the matrix:

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & & \\ 1 & x_N & x_N^2 \end{bmatrix}. \tag{1.2}$$

The final problem is then to find $\mathbf{c}$ to minimize:

$$f(\mathbf{c}) = \|\mathbf{y} - A\mathbf{c}\|^2.$$

Summary of least squares examples

In this case, we've translated *two* distinct problems into the same general form:

$$\text{find } \mathbf{c} \text{ to make } \|\mathbf{y} - A\mathbf{c}\|^2 \text{ as small as possible.}$$

This idea underlies our book. There are a great many problems in science, engineering, biology, and everywhere that can be turned into common matrix problems.

[5] Here, the data are.

x	y
1.02	8.7
1.01	8.89
1.23	8.88
1.24	9.87
1.4	9.62
1.41	9.57
1.59	10.13
1.65	9.71
1.64	10.62
1.77	9.81
2.01	10.39
2.0	11.79
2.15	11.09
2.12	11.05
2.19	11.01
2.31	11.84
2.49	12.17
2.56	11.49
2.7	11.46
2.71	11.49
2.91	11.76
2.97	11.59
3.07	10.94
3.1	11.56
3.19	11.36
3.29	9.7
3.36	11.37
3.54	11.02
3.66	10.35
3.74	9.85
3.74	8.92
3.81	9.03
3.87	8.78
3.94	9.31
4.12	8.5
4.23	8.63
4.27	8.06
4.38	7.4
4.44	7.19
4.49	8.1
4.57	7.4
4.83	6.52
4.84	5.91
4.93	6.16
5.02	5.77
5.15	4.78
5.26	4.48
5.39	3.77
5.5	2.93
5.41	2.72

1.3 COMMON MATRIX PROBLEMS

The three canonical and most common matrix problems are:

- Linear systems of equations: find $\mathbf{x}$ such that $\boldsymbol{A}\mathbf{x} = \mathbf{b}$
- Least squares problems: find $\mathbf{x}$ to minimize $\|\boldsymbol{A}\mathbf{x} - \mathbf{b}\|^2$
- Eigenvalue problems: find $\mathbf{x}, \lambda$ such that $\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x}$

These problems have deep relationships.

Solving least squares via a linear system.

We can turn the least squares problem

$$\text{find } \mathbf{x} \text{ to minimize } \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\|^2$$

into a related linear system.[6]

The first thing we need to do is understand how to find the minimum point of the least squares problem. There are a few ways to do this. One of the easiest is to think back to calculus class and about how we can find the extreme points of a simple quadratic function. Let $s(x) = 1/2ax^2 + b * x + c$ be a simple quadratic. This function only has a single extreme point when $s(x)$ is a minimum. Then, via calculus, we can find the extreme points by finding places where the derivative is zero. Here, $s'(x) = ax + b = 0$ gives the minimum point $x = -b/a$.

For least squares, we have :

$$f(\mathbf{x}) = \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\|^2 = (\boldsymbol{A}\mathbf{x} - \mathbf{b})^T(\boldsymbol{A}\mathbf{x} - \mathbf{b})$$

This idea that we can find the extreme points by looking for points where the derivative is zero generalizes to multivariate functions such as our $f(\mathbf{x})$ for least squares. This is because $f(\mathbf{x})$ for least squares is a smooth, convex functions. That means that $f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1-\alpha)f(\mathbf{y})$ when $0 \leq \alpha \leq 1$.

EXAMPLE 1.2 Showing that this property $f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1-\alpha)f(\mathbf{y})$ is a good exercise in working with vectors. The key step is to show:

$$\alpha^2\mathbf{x}^T\boldsymbol{A}^T\boldsymbol{A}\mathbf{x} + 2\alpha(1-\alpha)\mathbf{x}^T\boldsymbol{A}^T\boldsymbol{A}\mathbf{y} + (1-\alpha)^2\mathbf{y}^T\boldsymbol{A}^T\boldsymbol{A}\mathbf{y} \leq \alpha\mathbf{x}^T\boldsymbol{A}^T\boldsymbol{A}\mathbf{x} + (1-\alpha)\mathbf{y}^T\boldsymbol{A}^T\boldsymbol{A}\mathbf{y}$$

This can be done via by showing that difference is less than 0.

$$\begin{aligned} \text{LHS} - \text{RHS} &= \alpha(\alpha-1)\mathbf{x}^T\boldsymbol{A}^T\boldsymbol{A}\mathbf{x} + \alpha(\alpha-1)\mathbf{y}^T\boldsymbol{A}^T\boldsymbol{A}\mathbf{y} - 2\alpha(\alpha-1)\mathbf{x}^T\boldsymbol{A}^T\boldsymbol{A}\mathbf{y} \\ &= \alpha(\alpha-1)(\mathbf{x}-\mathbf{y})^T\boldsymbol{A}^T\boldsymbol{A}(\mathbf{x}-\mathbf{y}) \\ &= \alpha(\alpha-1)\|\boldsymbol{A}(\mathbf{x}-\mathbf{y})\| \\ &\leq 0. \end{aligned}$$

◆

The derivate or *gradient* of $f(\mathbf{x})$ is[7]

$$f'(\mathbf{x}) = 2\boldsymbol{A}^T\boldsymbol{A}\mathbf{x} - 2\boldsymbol{A}^T\mathbf{b}.$$

[6] This is not necessarily the best way to solve a least squares problem. We'll see better ways in the future!

[7] This isn't hard to work out, but is a bit tedious if you haven't seen it before. Try a problem where $\boldsymbol{A}$ is 3×2 to get started and just work element-wise.

This is zero when

$$\boldsymbol{A}^T\boldsymbol{A}\mathbf{x} = \boldsymbol{A}^T\mathbf{b}.$$

The result is a linear system of equations that *solve* a least squares problem.

From an eigenvector problem to a linear system

Suppose that we know that λ is an eigenvalue of $\mathbf{x}$. In that case, we may want to find the eigenvector. This can be done by solving the linear systems of equations:

$$(\boldsymbol{A} - \lambda\boldsymbol{I})\mathbf{x} = 0.$$

This is a *singular* system of linear equations that we will revisit in the future. So it isn't the standard for a linear system of equations, but it is a valid problem.

RECAP

So to get back to the starting question: what is a matrix?

For our purposes:

A matrix is a table of numbers where linear algebraic manipulations make sense on the rows or columns.

This extents the previous note to explain that the linear algebraic manipulations should make sense on the rows or columns.

Solving a least-squares problem with phone numbers doesn't make any sense. But solving this where the data come from experiments makes a good deal of sense. This is because in theorem 1.1, we show that there was a transformation matrix $\boldsymbol{T}$ that translated between two different formulations of the problem.

These problems have been stated and studied for centuries. The algorithms are decades old. Why, then, is this still an interesting subject? The reason that we study the subject of matrix computations, and indeed, the reason that this subject continues to be interesting is that our goal is to use the *structure* of the problem in order to solve the underlying problem better. New applications bring *new structures* with them. These often fit poorly with existing algorithms or methods or ideas – and then require us to think about how to make them better.

In this case, better may mean any of these:

faster   more accurately   more reliably   . . . .

We will see this soon!

EXERCISES

1. Consider a matrix of social security numbers. The matrix is a few thousand by 9. Each row is a distinct number and each column has one of the social security numbers in the same order they appear on the card. Explain why this is or isn't a matrix, using the ideas from class.
2. Consider a matrix from the social security database consisting of the GPT-5 embedding of everyone's social security number. Each row is a set of 3072 numbers corresponding to the embedding of the string of their social security number. Explain why this is or isn't a matrix, using the ideas from class.
3. Consider a collection of 10000 greyscale images. Each image is 64 by 64 pixels, or 4096 distinct numbers. Suppose we create a matrix where each column represents a 32 by 32 pixel corner of each image. So the matrix is 1024 rows by 40000 columns. Explain why this is or isn't a matrix, using the ideas from class.
4. Consider a matrix of demographic information for many all undergrad CS applicants at Purdue. Each student is a row. The columns represent:
   - 0/1 (had a highschool GPA >= 3.75)
   - 0/1 (had a highschool GPA >= 3.25)
   - 0/1 has more than three letters of recommendation
   - number of times the word "excellent" "top-tier" "best" appears in letters of recommendation
   - Flesch-Kincaid Grade Level of personal statement

   Explain why this is or isn't a matrix, using the ideas from class.
5. An audio file is just a big long vector of information. (Well, two vectors for a stereo audio file if you want to be very precise, but let's suppose we just have one signal.) Suppose we create a matrix where each column represents 44100 sequential pieces of information from the big vector, i.e.,
$$\boldsymbol{A} = \begin{bmatrix} x_1 & x_{44101} & x_{88201} & \cdots \\ \vdots & \vdots & \vdots & \cdots \\ x_{44100} & x_{88200} & x_{132300} & \cdots \end{bmatrix}.$$
6. Let $\boldsymbol{A}$ be the matrix from any of the previous questions. Consider the new matrix $\boldsymbol{B} = \boldsymbol{A}^T\boldsymbol{A}$. Explain why this is or isn't a matrix, using the ideas from class. There *may not be a clear answer*.
7. Let $\boldsymbol{A}$ be the matrix from any of the previous questions. Consider the new matrix $\boldsymbol{E}$ formed by the eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$. To be concrete, let $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^T$ be the spectral decomposition. Let $\boldsymbol{E} = \boldsymbol{V}$ or the first k columns of $\boldsymbol{V}$ where columns are picked in order of largest eigenvalues. Explain why this is or isn't a matrix, using the ideas from class. Again, there *may not be a clear answer*.

8. A key focus in this book is on structure within the problem. Let $\boldsymbol{A}$ be the matrix from the linear or quadratic fits from (1.1) or (1.2). Show that the matrix $\boldsymbol{A}^T\boldsymbol{A}$ has some type of structure in the entries.
9. Describe and implement an algorithm to build the matrix $\boldsymbol{A}^T\boldsymbol{A}$ for the least squares example *without forming* the matrix $\boldsymbol{A}$. That is, you need to use the structure from the previous question in order to implement an optimized algorithm. Describe the runtime of your algorithm.
10. As an example of how we can solve a problem *better* using structure,

Maybe a block-diagonal linear system? Maybe a rank-1 least squares problem? Something with the mean matrix? Something with iterative refinement?

11. Consider the following problem,[8] Let $\boldsymbol{C}$ be an $n \times k$ matrix with $n \geq k$.
    (a) Show that $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{C} & \boldsymbol{I} \\ -\boldsymbol{I} & \boldsymbol{C}^T \end{bmatrix}$ is invertible.
    (b) Let $\mathbf{b} = \begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix}$, give an algorithm to solve $\boldsymbol{A}\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \mathbf{b}$.
    (c) Give an explicit form for the inverse (this may involve the inverse of other matrices.) (Hint, show this for $\boldsymbol{C} = 0$, then try $\boldsymbol{C}$ as a rank-1 matrix $\boldsymbol{C} = \mathbf{u}\mathbf{v}^T$.)

    This is what we mean by using structure to solve a problem better. We can build a simple algorithm to solve this type of system.

[8] I ran into this problem when working on erasure coded eigensolvers, but it's been abstracted to avoid the details of the paper. This abstraction is essentially how I solved it.

NOTATION 2

Let us begin by introducing basic notation for matrices and vectors.

Learning objectives

1. Consider skipping this section if you are familiar with notation already.
2. Recognize how we write and discuss matrices, vectors, and their entries and some common operations.

2.1 MATRICES

We'll use $\mathbb{R}$ to denote the set of real-numbers and $\mathbb{C}$ to denote the set of complex numbers.

We write the space of all $m \times n$ real-valued matrices as $\mathbb{R}^{m\times n}$. Each

$$\boldsymbol{A} \in \mathbb{R}^{m\times n} \quad \text{is} \quad \begin{bmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & & \vdots \\ A_{m,1} & \cdots & A_{m,n} \end{bmatrix} \quad \text{where} \quad A_{i,j} \in \mathbb{R}.$$

Sometimes, I'll write:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}$$

instead. With only a few exceptions, matrices are written as *bold, capital* letters, as in $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$. Occasionally, we'll use a capital Greek letter, as in $\boldsymbol{\Lambda}$ or $\boldsymbol{\Sigma}$. Matrix elements are written as sub-scripted, *unbold* letters. When clear from context,

$$A_{i,j} \text{ is written } A_{ij}$$

instead, e.g. A_{11} instead of $A_{1,1}$.

Another notation for $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is

$$\boldsymbol{A} : n \times n.$$

Sometimes this is nicer to write on the board.

IN CLASS I'll usually write matrices with just upper-case letters. If you are unsure if something is a matrix or an element, raise your hand and ask, or *quietly* ask a neighbor.

2.2 VECTORS

We write the set of length-n real-valued vectors as $\mathbb{R}^n$. Each

$$\mathbf{x} \in \mathbb{R}^n \quad \text{is} \quad \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad \text{where } x_i \quad \in \mathbb{R}.$$

Vectors are denoted by *lowercase, bold* letters, as in $\mathbf{x}, \mathbf{y}, \mathbf{z}$. As with matrices, elements are sub-scripted, *unbold* letters. Sometimes, we'll write vector elements as

$$x_i \text{ or } [x]_i \text{ or } x(i).$$

Usually, this choice is motivated by a particular application.

Throughout the class, vectors are column *vectors.*

IN CLASS I'll usually write vectors with just lower-case letters and *will try* to follow the convention of underlining them. This is much nicer than the little arrows vectors are sometimes written with.

2.3 SCALARS

Lower-case greek letters are scalars as in α, β, γ.

2.4 OPERATIONS

TRANSPOSE Let $\boldsymbol{A} : m \times n$, then

$$\boldsymbol{B} : n \times m = \boldsymbol{A}^T \text{ has } B_{i,j} = A_{j,i}.$$

$$\textit{Example } \boldsymbol{A} = \begin{bmatrix} 2 & 3 \\ 1 & 4 \\ 3 & -1 \end{bmatrix} \quad \boldsymbol{A}^T = \begin{bmatrix} 2 & 1 & 3 \\ 3 & 4 & -1 \end{bmatrix}$$

HERMITIAN (Also called conjugate transpose) Let $\boldsymbol{A} \in \mathbb{C}^{m \times n}$, then

$$\boldsymbol{B} \in \mathbb{C}^{n \times m} = \boldsymbol{A}^* = \boldsymbol{A}^H \text{ has } B_{i,j} = \overline{A}_{j,i}.$$

$$\textit{Example } \boldsymbol{A} = \begin{bmatrix} 2 & 3 & \\ i & 4 & \\ & 3 & -i \end{bmatrix} \quad \boldsymbol{A}^* = \begin{bmatrix} 2 & -i & 3 \\ 3 & 4 & i \end{bmatrix}$$

ADDITION Let $\boldsymbol{A} : m \times n$ and $\boldsymbol{B} : m \times n$, then

$$\boldsymbol{C} : m \times n = \boldsymbol{A} + \boldsymbol{B} \implies C_{i,j} = A_{i,j} + B_{i,j}.$$

$$\textit{Example } \boldsymbol{A} = \begin{bmatrix} 2 & 3 \\ 1 & 4 \\ 3 & -1 \end{bmatrix}, \boldsymbol{B} = \begin{bmatrix} 1 & -1 \\ 2 & 3 \\ -1 & 1 \end{bmatrix} \boldsymbol{A} + \boldsymbol{B} = \begin{bmatrix} 3 & -2 \\ 3 & 2 \\ 2 & 0 \end{bmatrix}.$$

SCALAR MULTIPLICATION Let $\boldsymbol{A} : m \times n$ and $\alpha \in \mathbb{R}$, then

$$\boldsymbol{C} : m \times n = \alpha \boldsymbol{A} + \boldsymbol{B} \implies C_{i,j} = \alpha A_{i,j}.$$

$$\textit{Example } \boldsymbol{A} = \begin{bmatrix} 2 & 3 \\ 1 & 4 \\ 3 & -1 \end{bmatrix}, 5\boldsymbol{A} = \begin{bmatrix} 10 & 15 \\ 5 & 20 \\ 15 & -5 \end{bmatrix}$$

MATRIX MULTIPLICATION Let $\boldsymbol{A} : m \times n$ and $\boldsymbol{B} : n \times k$, then

$$\boldsymbol{C} : m \times k = \boldsymbol{AB} \implies C_{i,j} = \sum_{r=1}^{n} A_{i,r} B_{r,j}.$$

MATRIX-VECTOR MULTIPLICATION Let $\boldsymbol{A} : m \times n$ and $\mathbf{x} \in \mathbb{R}^n$, then

$$\mathbf{c} \in \mathbb{R}^m = \boldsymbol{A}\mathbf{x} \implies c_i = \sum_{j=1}^{n} A_{i,j} x_j.$$

This operation is just a special case of matrix multiplication that follows from treating $\mathbf{x}$ and $\mathbf{c}$ as $n \times 1$ and $m \times 1$ matrices, respectively.

VECTOR ADDITION, SCALAR VECTOR MULTIPLICATION These are just special cases of matrix addition and scalar matrix multiplication where vectors are viewed as $n \times 1$ matrices.

2.5 COMMON MATRICES & VECTORS

The $n \times n$ identity matrix is :

$$\boldsymbol{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & \vdots \\ \vdots & \cdots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix}$$

This matrix is rarely written with it's explicit dimension as that can almost always be inferred by context. That is to say, the dimension of the identity matrix is whatever it needs to be such that the matrix equation makes sense. For clarity, we might sometimes write:

$$\boldsymbol{I}_n$$

to denote the $n \times n$ matrix explicitly. Thus,

$$\boldsymbol{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The identity matrix has the property that $\boldsymbol{AI} = \boldsymbol{A}$ for any matrix $\boldsymbol{A}$. It's like multiplying by 1.

We denote the ith column of the identity matrix by $\mathbf{e}_i$:

$$\mathbf{e}_i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad i\text{th position}$$

For instance,

$$\mathbf{e}_2 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T.$$

Using these vectors, we can write the ith column of any matrix as

$$\boldsymbol{A}\mathbf{e}_i.$$

It is, perhaps, alarming that $\mathbf{e}_i$ is frequently used without specifying its dimension. However, just like the identity matrix above, it is almost always possible to work out the dimension. If we believe it is helpful to specify it, we'll use $\mathbf{e}_i^{(n)}$.

Finally, the vector $\mathbf{e}$ will be used to denote the vector of all ones:

$$\mathbf{e} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

Except sometimes it will be used as an error vector for a problem.

Some people use the matrix $\boldsymbol{J}$ to represent the *matrix of all ones*

$$\boldsymbol{J} = \begin{bmatrix}\mathbf{e} & \mathbf{e} & \cdots & \mathbf{e}\end{bmatrix}.$$

This isn't needed however, as we can easily write this as the rank-1 matrix $\boldsymbol{J} = \mathbf{e}\mathbf{e}^T$, which – while a bit longer – reveals the elementary rank-1 structure.

2.6 MATRIX & VECTOR PARTITIONING

It is often useful to represent a matrix as a collection of vectors. In this case, we write

$$\boldsymbol{A} : m \times n = \begin{bmatrix}\mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n\end{bmatrix}$$

where each $\mathbf{a}_j \in \mathbb{R}^m$. *This form corresponds to a partition into columns.*

Alternatively, we may wish to partition a matrix into rows.

$$\boldsymbol{A} : m \times n = \begin{bmatrix}\mathbf{r}_1^T \\ \mathbf{r}_2^T \\ \vdots \\ \mathbf{r}_m^T\end{bmatrix} \quad \text{where each} \quad \mathbf{r}_i \in \mathbb{R}^n.$$

Using the column partitioning:

$$\boldsymbol{A}\mathbf{x} = \begin{bmatrix}\mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n\end{bmatrix}\begin{bmatrix}x_1 \\ x_2 \\ \vdots \\ x_n\end{bmatrix} = \sum_j x_j \mathbf{a}_j.$$

And with the row partitioning:

$$\boldsymbol{A}\mathbf{x} = \begin{bmatrix}\mathbf{r}_1^T \\ \mathbf{r}_2^T \\ \vdots \\ \mathbf{r}_m^T\end{bmatrix}\mathbf{x} = \begin{bmatrix}\mathbf{r}_1^T\mathbf{x} \\ \mathbf{r}_2^T\mathbf{x} \\ \vdots \\ \mathbf{r}_m^T\mathbf{x}\end{bmatrix}.$$

Another useful partitioned representation of a matrix is into blocks:

$$\boldsymbol{A} = \begin{bmatrix}\boldsymbol{A}_{1,1} & \boldsymbol{A}_{1,2} \\ \boldsymbol{A}_{2,1} & \boldsymbol{A}_{2,2}\end{bmatrix}$$

or

$$\boldsymbol{A} = \begin{bmatrix}\boldsymbol{A}_{1,1} & \boldsymbol{A}_{1,2} & \boldsymbol{A}_{1,3} \\ \boldsymbol{A}_{2,1} & \boldsymbol{A}_{2,2} & \boldsymbol{A}_{2,3} \\ \boldsymbol{A}_{3,1} & \boldsymbol{A}_{3,2} & \boldsymbol{A}_{3,3}\end{bmatrix}.$$

Here, the sizes "just have to work out." Formally, all $\boldsymbol{A}_{i,\cdot}$ must have the same number of rows and all $\boldsymbol{A}_{\cdot,j}$ must have the same number of columns. The sizes will usually be determined by something that results in a unique representation. If not, the statement may be *very flexible* and *forgiving* on the dimensions – alternatively, there could be an issue with the statement.

2.7 BACKGROUND & BASIC RESULTS

We expect people to have seen similar to the following statements. These will be reiterated as required, but will not be established within this book.

- Full rank, square linear systems have a unique solutions.
- The null space of a matrix is the set of vectors where $\boldsymbol{A}\mathbf{x} = 0$.
- A symmetric matrix has an orthogonal set of eigenvectors.
- Any square matrix can be written in the Jordan canonical form $\boldsymbol{A} = \boldsymbol{X}\boldsymbol{J}\boldsymbol{X}^{-1}$ for a set of generalized eigenvectors $\boldsymbol{X}$ and with Jordan blocks.[1]

[1] This will be use sparingly and only for analysis.

EXERCISES

1. Identify the following:
$$\mathbf{f}, z_1, \mathbf{x}_1, \alpha, \beta, \boldsymbol{C}, \boldsymbol{C}_1, \boldsymbol{\Sigma}, B_{i,j}, \mathbf{b}_{i,j}$$
2. What are the results of the computations $\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & -1 \end{bmatrix}\begin{bmatrix} 1 & 2 & 3 \\ 6 & 4 & 2 \\ -2 & -2 & -2 \end{bmatrix}$ and $\begin{bmatrix} 1 & 2 & 3 \\ 6 & 4 & 2 \\ -2 & -2 & -2 \end{bmatrix}\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & -1 \end{bmatrix}$? Describe the difference in multiplying a diagonal matrix on the left compared with the right and give a short proof of your result.
3. Describe an algorithm to implement a diagonal matrix multiplication from the left or the right. This should not use a general matrix multiplication.
4. Let $\mathbf{e}_i$ be the vector with all zeros and a 1 in the ith entry. Let $\boldsymbol{A}$ be an $m \times n$ matrix. Give an expression for the rth row of $\boldsymbol{A}$ as a result of a matrix vector product. Give an expression for the cth column of $\boldsymbol{A}$ as a result of a matrix vector product.
5. The vector $\mathbf{e} = \begin{bmatrix} 1 & \dots & 1 \end{bmatrix}^T$ (i.e. the all ones vector). $\mathbf{x}$ = ‘ones(1000,1)‘ $\mathbf{y}$ = ‘1:1000‘, what is $\mathbf{x}^T\mathbf{y}$ = ?
6. $\mathbf{x} = \begin{bmatrix} 1.5 & 2 & -3 \end{bmatrix}^T$. (Assume $\mathbf{e}$ is 4×1.) $\mathbf{e}\mathbf{x}^T$ =? $\mathbf{x}\mathbf{e}^T$ =?
7. $\mathbf{x} = \begin{bmatrix} -5 & 4 & 2 \end{bmatrix}^T$. (Assume $\mathbf{e}_i$ is 3×1.) $\mathbf{e}_2\mathbf{x}^T$ =? $\mathbf{x}\mathbf{e}_1^T$ =?

3 STRUCTURE IN MATRICES

Learning objectives.
1. How to understand *structure* in matrices and where it comes from.
2. What a Hankel matrix is.
3. What a sparse matrix is.

Terms: Hankel, tridiagonal, triangular, Hessenberg, sparse, bidiagonal, data-sparse, symmetric positive definite density, nnz

One of the goals of matrix computations is to exploit structure inside the problem. That is, there are algorithms that will work for essentially any matrix that is a valid input for the problem. But you are unlikely to have any matrix! You have a matrix that arises in your application. The idea is that we should be able to take advantage of that structure in order to write better algorithms!

3.1 A FIRST EXAMPLE OF STRUCTURE

Consider those least squares problems we had. The matrix involved with fitting a quadratic polynomial to the data was

$$\boldsymbol{A} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & & \\ 1 & x_N & x_N^2 \end{bmatrix}.$$

One way to solve a least-squares problem (which we will derive eventually, see a future lecture!) is to convert it into a set of linear equations called the normal equations.

RECAP OF THE NORMAL EQUATIONS FOR LEAST-SQUARES [1]

To find $\mathbf{x}$ such that $\|\mathbf{b} - \boldsymbol{A}\mathbf{x}\|_2^2$ is minimized, we can solve the normal equations

$$\boldsymbol{A}^T\boldsymbol{A}\mathbf{x} = \boldsymbol{A}^T\mathbf{b}.$$

Let's look at the matrix $\boldsymbol{A}^T\boldsymbol{A}$ defined in the least squares problem with above:

$$\boldsymbol{A}^T\boldsymbol{A} = \begin{bmatrix} N & \sum_i x_i & \sum_i x_i^2 \\ \sum_i x_i & \sum_i x_i^2 & \sum_i x_i^3 \\ \sum_i x_i^2 & \sum_i x_i^3 & \sum_i x_i^4. \end{bmatrix}.$$

Again, this is a very specific type of matrix. Notice that the values are constant in certain regions:[2]

$$\boldsymbol{A}^T\boldsymbol{A} = \begin{bmatrix} y_1 & y_2 & y_3 \\ y_2 & y_3 & y_4 \\ y_3 & y_4 & y_5 \end{bmatrix}.$$

So in order to solve these types of equations, we do *not* need a general purpose means of solving a linear system of equations. What we need is just an algorithm to solve these specific types of inputs.

[1] There are better ways to solve these problems involving the QR decomposition. We will see that soon.

[2] Note that the original matrix involves $N + 1$ pieces of information: $1, x_1, \ldots, x_N$, but the resulting matrix has only 5 distinct pieces of information.

This type of input arises frequently, and so it has acquired a name. The general form is called a *Hankel* matrix.[3] Recall that one of our goals is to avoid naming things after people. Towards that end, we believe these are better named *left shift* matrices. Moreover, this immediately suggests a relationship with right shift matrices.

Definition 3.1
A matrix is a *left shift matrix* if $A_{i,j} = A_{i-1,j+1}$ whenever $i-1, j+1$ is a valid index into the matrix.

The following are straightforward results of this definition.

Theorem 3.2

- A square left shift matrix is symmetric.
- An $m \times n$ left shift matrix is defined by $m+n-1$ numbers.

Thus, as an example, we could work out how to solve linear equations with left shift matrices and show how to use that to fit 1 dimensional curves. This would *hopefully* make it better for this single purpose.

3.2 SPARSE STRUCTURE IN MATRICES

The next type of structure we'll discuss is *sparse structure* or simply put *sparsity*. Simply put, a sparse matrix consists of mostly zeros. Conversely, a dense matrix is a matrix that is not sparse.

Matrices with mostly zeros arise often when we are studying real-world systems. There are a large number of examples of this at the SuiteSparse Matrix Collection (`https://sparse.tamu.edu/`) which has a database of tens of thousands of examples of these real-world system and the matrices that result. We recently wrote the following passage in a grant application to explain the wide-scale presence of sparsity:

> Parsimony is a hallmark of scientific theories. They should be as simple as possible to explain the world. When these theories are realized in computer simulations, the result is often a sparse set of equations – a set where variables only have minimal dependencies on each other. This occurs because many simulations directly model a small unit of space and the simplest relationships have limited impact on their surroundings.

We'll see a few examples as we go along through our lectures. Let's get started with a simple one that comes up for a problem that has all the of the actual interesting detail removed.

[3] Who was Hankel? Is he really the first to look at these? Short answer: I don't know. Wikipedia has a little bit of information, `https://en.wikipedia.org/wiki/Hermann_Hankel`, so does the Encyclopedia of Mathematics, `https://www.encyclopediaofmath.org/index.php/Hankel_matrix`, but neither gets at the question of who was the first to specialize on these types of matrices.

EXAMPLE 3.3 Consider the following problem. Suppose you are sitting on the number line at 0, and you move left and right with equal probability (1/2) at each step. What is the expected length of time until you first hit the integer +6 or −4. [4]

We can solve this by letting $x_{\{i\}}$ be the expected length of time before you first hit either integer given that you start at state $\{i\}$. Note that i can range from −4 to +6. Also note that $x_{\{-4\}} = x_{\{+6\}} = 0$. Let's work out the others. Suppose we were to start at state $\{-3\}$. Then we have at least one move that results in two cases: we move to $\{-4\}$ with probability 1/2 and stop, or we move to $\{-2\}$ with probability 1/2 and continue. Consequently:

$$x_{\{-3\}} = 1 + \tfrac{1}{2}x_{\{-4\}} + \tfrac{1}{2}x_{\{-2\}}.$$

Likewise, we can apply the same analysis to get:

$$\begin{aligned} x_{\{-2\}} &= 1 + \tfrac{1}{2}x_{\{-3\}} + \tfrac{1}{2}x_{\{-1\}} \\ x_{\{-1\}} &= 1 + \tfrac{1}{2}x_{\{-2\}} + \tfrac{1}{2}x_{\{0\}} \\ &\vdots \\ x_{\{+5\}} &= 1 + \tfrac{1}{2}x_{\{+4\}} + \tfrac{1}{2}x_{\{+6\}}. \end{aligned}$$

This gives us an overall linear system of equations:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\{-4\}} \\ x_{\{-3\}} \\ x_{\{-2\}} \\ x_{\{-1\}} \\ x_{\{0\}} \\ x_{\{+1\}} \\ x_{\{+2\}} \\ x_{\{+3\}} \\ x_{\{+4\}} \\ x_{\{+5\}} \\ x_{\{+6\}} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Note that most of the elements are zero. This is an example of a sparse matrix. ◆

EXAMPLE 3.4 Consider the following revised setting.[5] Suppose you are sitting on the number line at 0, and you move right with probability p and stay put with probability q. With probability r, you restart at 0 at any value between 0 and your current position-1, with the restart chosen uniformly at random. What is the expected time until you hit +10? We also have $p + q + r = 1$. If $p = 1/2, q = 1/4, r = 1/4$, how long do we expect to take to reach +10?

Again, this can be solved by a similar linear system. First, we have $x_{\{0\}} = 1 + (1 - p)x_{\{0\}} + px_{\{+1\}}$. The other equations all proceed in a similar fashion.

Double check the small equation here doesn't cause spacing issues.

[4] More generally, this is an instance of first-hitting time in Markov chains, but we don't need to get into the formalities of that characterization and generalization of the problem.

[5] A system of equations closely related to this arose when were were studying the ability of algorithms to *make progress* when there are failures. In this case, p is the probability of progress on a given step and q and r model two different failure scenarios that take us back to previous levels of progress.

$$\begin{bmatrix} p & -p & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r & (1-q) & -p & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{2}r & -\frac{1}{2}r & (1-q) & -p & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{3}r & -\frac{1}{3}r & -\frac{1}{3}r & (1-q) & -p & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{4}r & -\frac{1}{4}r & -\frac{1}{4}r & -\frac{1}{4}r & (1-q) & -p & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{5}r & -\frac{1}{5}r & -\frac{1}{5}r & -\frac{1}{5}r & -\frac{1}{5}r & (1-q) & -p & 0 & 0 & 0 & 0 \\ -\frac{1}{6}r & -\frac{1}{6}r & -\frac{1}{6}r & -\frac{1}{6}r & -\frac{1}{6}r & -\frac{1}{6}r & (1-q) & -p & 0 & 0 & 0 \\ -\frac{1}{7}r & -\frac{1}{7}r & -\frac{1}{7}r & -\frac{1}{7}r & -\frac{1}{7}r & -\frac{1}{7}r & -\frac{1}{7}r & (1-q) & -p & 0 & 0 \\ -\frac{1}{8}r & -\frac{1}{8}r & -\frac{1}{8}r & -\frac{1}{8}r & -\frac{1}{8}r & -\frac{1}{8}r & -\frac{1}{8}r & -\frac{1}{8}r & (1-q) & -p & 0 \\ -\frac{1}{9}r & -\frac{1}{9}r & -\frac{1}{9}r & -\frac{1}{9}r & -\frac{1}{9}r & -\frac{1}{9}r & -\frac{1}{9}r & -\frac{1}{9}r & -\frac{1}{9}r & (1-q) & -p \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\{0\}} \\ x_{\{+1\}} \\ x_{\{+2\}} \\ x_{\{+3\}} \\ x_{\{+4\}} \\ x_{\{+5\}} \\ x_{\{+6\}} \\ x_{\{+7\}} \\ x_{\{+8\}} \\ x_{\{+9\}} \\ x_{\{+10\}} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Note that most of the elements are not zero. Is this system of equations sparse? We will discuss that. ◆

The case in theorem 3.3 has all the hallmarks of sparsity. There are at most 3 non-zero entries per row. Indeed, the problem is actually even more special. Because the entries are structured in the three *diagonals*, the matrix is called *tridiagonal.* This is another type of structure in matrices.

The case in theorem 3.4 has around half of the entries of the matrix listed. Yet, there are still many entries that are zero. This matrix is an example of a lower *Hessenberg* matrix – see section 3.5.

Definitions of sparsity

Is this matrix sparse? There are two competing ways to think about it.

Definition 3.5 (sparse, informal)
A matrix is sparse if there is enough structure in the zero entries to use advantageously in an algorithm.

Definition 3.6 (sparse, alternative)
A class of matrices is sparse if it has $o(mn)$ non-zero entries.[6]

Considering the matrix from theorem 3.4. This matrix is sparse by the informal definition as it's clear there are enough zeros to take advantage of. It is not sparse by the formal definitions as the number of nonzero entries is roughly $\frac{1}{2}n^2$. This shows how *sparsity* is not necessarily clear cut in terms of applications and theory.

Measuring sparsity

There is no single way measure the *sparsity* of a matrix. The key components are:

- the number of non-zero entries called the *number of nonzeros* or *nnz*
- the number of zero entries
- the total number of entries or mn for an $m \times n$ matrix .

[6] This is the "little-o" notation. What this means is that as $m, n \to \infty$, then the number of non-zeros in the matrix is any function that grows asymptotically more slowly than mn. As an example, if $m = n$, then $n^{1.5}$ is $o(n^2)$.

Note that any one of these can be inferred from the other two.

The *density* of an $m \times n$ matrix typically refers to the ratio:[7]

$$\text{density} = \frac{\text{number of nonzeros}}{\text{total possible entries}} = \frac{\text{nnz}}{mn}.$$

The *average entries* per row or per column of an $m \times n$ matrix are

$$\text{average nonzeros per row} = \frac{\text{number of nonzeros}}{m}$$

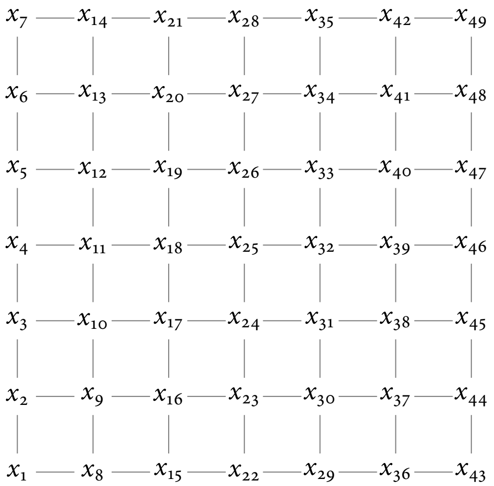
$$\text{average nonzeros per column} = \frac{\text{number of nonzeros}}{n}.$$

As matrices get bigger, these density behaves differently than the average nonzeros per row and columns. Suppose we have an $n \times n$ class of matrices with $n \log n$ non-zero entries. Then the average nonzeros per row grows as $\log n \to \infty$ but the density decays as $\log n/n \to 0$.

In most of the problems we study, an $n \times n$ matrix means we have n variables which governs the size of the matrix. In these settings, the average nonzeros per row or per column gives a better guide to the sparsity of the problem. One additional reason to prefer this measure is that it gives a rough guide to the amount of *work* needed to look at a single row or column.

3.2.1 *Grid and spatial structure*

An extremely common source of sparsity arises because we solve equations *on a grid* of points that represents a surface or a plane. In this case, variables are arranged on a *grid* and dependencies among the variables are represented by the grid adjacencies.


x_7 x_{14} x_{21} x_{28} x_{35} x_{42} x_{49}
x_6 x_{13} x_{20} x_{27} x_{34} x_{41} x_{48}
x_5 x_{12} x_{19} x_{26} x_{33} x_{40} x_{47}
x_4 x_{11} x_{18} x_{25} x_{32} x_{39} x_{46}
x_3 x_{10} x_{17} x_{24} x_{31} x_{38} x_{45}
x_2 x_9 x_{16} x_{23} x_{30} x_{37} x_{44}
x_1 x_8 x_{15} x_{22} x_{29} x_{36} x_{43}

In this scenario the variable x_1 may depend on or influence x_2, x_8. Likewise, the variable $x_3 2$ may depend on or influence $x_2 5, x_3 1, x_3 3, x_3 9$.

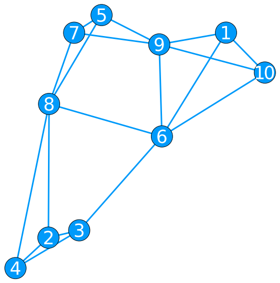
[7] These terms always need to be checked as terminology is highly variable.

More generally, the variable dependence follows the underlying geometry. This might be 3d. This might have non-grid structure. In the most common cases, the dependencies only reflect *nearby* points which produces sparsity.

Sparsity in data science

Sparse matrices arise frequently in data science, machine learning, and AI applications. The reason is that most real-world databases are not dense matrices. Image data are a notable exception and they are dense.

For instance, the adjacency matrix of the graph is used to compute the PageRank vector for the graph. For a graph with n nodes, this matrix will have nonzero entries for each edge. Typical graphs include social networks, where there are millions of nodes but each node only has a few thousand edges induced by acquaintance relationships.



$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

FIGURE 3.1 – A small contact network with the adjacency matrix for the network.

Likewise, in user behavior datasets, the rows are often users and the columns reflect behaviors. This could involve watching a movie. Since most people have only seen a few movies, these data are sparse. A well known instance of this matrix is the *Netflix* matrix of user ratings.

3.3 SYMMETRIC POSITIVE DEFINITE MATRICES

When we solved the least squares problem via the normal equations, the matrix $\boldsymbol{A}^T\boldsymbol{A}$ came up. It turns out that matrices with this form are extremely important and so they are called *symmetric positive semi-definite matrices*. More explicitly, a symmetric positive definite matrix is one that can be written:

$$\boldsymbol{B} = \boldsymbol{F}^T\boldsymbol{F}$$

for some matrix $\boldsymbol{F}$. Note that $\boldsymbol{F}$ need not be unique. Another way of characterizing symmeric positive semi-definite matrices is:

$$\mathbf{x}^T\boldsymbol{A}\mathbf{x} \geq 0 \text{ for all } \mathbf{x}.$$

Definition 3.7
A matrix $\boldsymbol{A}$ is symmetric positive definite if

$$\mathbf{x}^T \boldsymbol{A}\mathbf{x} > 0 \text{ for all } \mathbf{x} \neq 0.$$

3.4 FAST OPERATORS AND DATA-SPARSE MATRICES

Consider the matrix

$$\boldsymbol{C} = \begin{bmatrix} 1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ -\frac{1}{2} & 1 & -\frac{1}{2} & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & -\frac{1}{2} & 1 & -\frac{1}{2} \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 1 \end{bmatrix}$$

This is the same matrix as in theorem 3.3 except where every 0 entry has been replaced by 0.1. This matrix is not sparse. Yet, it's hardly different from the sparse matrix in that example.

From a programming perspective, the only difference between the two is that all the "unspecified" elements have been replaced by a different value. The Julia function to lookup an element in a sparse matrix implements this algorithm:[8]

```
function getindex(A::SparseMatrixCSC , i::Integer , j::Integer)
if (any entries in column j)
  if (element in row i is in column j)
     return value from row i and column j
   else
     return 0
   end
else
  return 0
end
end
```

This pseudocode could be trivially adapted to give us $\boldsymbol{C}$ instead by changing the two lines `return 0` to `return 0.1`.[9]

Why zero is special.

The reason that we call matrices with 0 *sparse* is because 0 is the multiplicative null element. Multiplying anything by 0 produces 0. Consequently, these elements drop out of any type of multiplication operations. Recall that the three fundamental problems we consider

linear systems $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ least squares $\min \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\|$ eigenvalues $\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x}$

[8] This function takes in a sparse matrix data structure, and two indices *i* and *j* of the element in the matrix. Their code is also much more general, explicit, and more element.

[9] In this case, we note that it'd likely be better to parameterize the default return value instead of implementing a new function for every possible different "not-specified" value.

all involve multiplying a matrix by a vector. Consequently, if the matrix has zeros, these matrix elements simply play no role in the multiplication operation. This is why the value of 0 is special and why sparse matrices must have zeros.

Data sparse.

The matrix $\boldsymbol{C}$ is still structured. This is an instance of what is called a *data sparse* matrix. It is a matrix whose entries can be inferred from a small amount of data. But where the resulting matrix is not itself sparse. This has implications for how an algorithm might use the matrix itself. For instance, looking at all of the elements of an $n \times n$ data sparse matrix might involve an operation for each element which would take n^2 time. However, there might be other operations that could be done faster.

Let

$$\boldsymbol{B} = \begin{bmatrix} 0.9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{2}{5} & 0.9 & -\frac{2}{5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{2}{5} & 0.9 & -\frac{2}{5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{2}{5} & 0.9 & -\frac{2}{5} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{2}{5} & 0.9 & -\frac{2}{5} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{2}{5} & 0.9 & -\frac{2}{5} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{2}{5} & 0.9 & -\frac{2}{5} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2}{5} & 0.9 & -\frac{2}{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2}{5} & 0.9 & -\frac{2}{5} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2}{5} & 0.9 & -\frac{2}{5} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9 \end{bmatrix}$$

Then $\boldsymbol{C} = \boldsymbol{B} + 0.1\mathbf{e}\mathbf{e}^T$. This gives us an expression for $\boldsymbol{C}$ as a rank-1 correction for $\boldsymbol{B}$. And $\boldsymbol{B}$ is sparse. Consider computing $\boldsymbol{C}\mathbf{x}$ for a given vector $\mathbf{x}$. This can be done by computing $\boldsymbol{B}\mathbf{x}$ and then adding $0.1\mathbf{e}^T\mathbf{x}$ to each element of $\boldsymbol{B}\mathbf{x}$ which is far more efficient than looking at all n^2 elements.

Sometimes data sparse matrices are called *fast operators* because they allow us to write a matrix-vector program or function that is much faster than looking at all of the elements of the matrix.

3.5 MATRIX STRUCTURES WITH A PATTERN

Many types of matrix structure include an explicit pattern.

RIGHT SHIFT A *right shift* matrix, also known as a Toeplitz matrix, is one where $A_{i,j} = A_{i+1,j+1}$ whenever the latter index is valid. For example

$$\begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \\ \beta_1 & \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_2 & \beta_1 & \alpha_1 & \alpha_2 \\ \beta_3 & \beta_2 & \beta_1 & \alpha_1 \\ \beta_4 & \beta_3 & \beta_2 & \beta_1 \end{bmatrix}.$$

Notice how each row *shifts to the right.* This gives rise to a c

CIRCULANT A *circulant matrix* is a right shift matrix where the elements "wrap around"

$$\begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \\ \alpha_2 & \alpha_3 & \alpha_4 & \alpha_1 \\ \alpha_3 & \alpha_4 & \alpha_1 & \alpha_2 \end{bmatrix}.$$

LEFT SHIFT We already defined *left shift* matrices above and noted that they are also called *Hankel matrices*. Formally, they have $A_{i,j} = A_{i-1,j+1}$ whenever $i-1, j+1$ is a valid index into the matrix.

DIAGONAL A diagonal matrix is one where only the main diagonal entries are set to non-zero values. An example is the identity matrix. where all the diagonal entries are 1.

TRIDIAGONAL A tridiagonal matrix is one where only three diagonals are set. This typically means the main diagonal and the diagonal above and below it. There are cases when this could mean another set of diagonals.

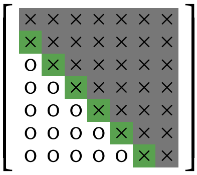
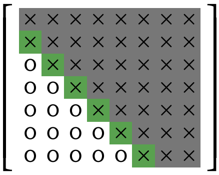
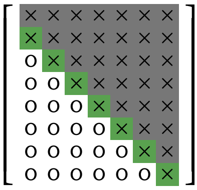
TRIANGULAR A triangular matrix gives a triangular region of zeros. This can be in the upper or lower triangular portion. The name flips and tells where the entries are, not the zeros, so

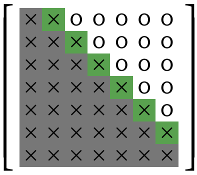
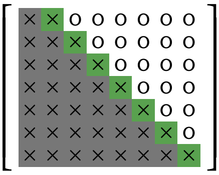
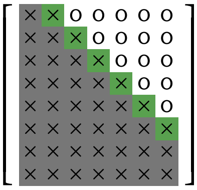
$$\textit{upper triangular} \text{ is } \begin{bmatrix} \times&\times&\times&\times&\times&\times&\times \\ o&\times&\times&\times&\times&\times&\times \\ o&o&\times&\times&\times&\times&\times \\ o&o&o&\times&\times&\times&\times \\ o&o&o&o&\times&\times&\times \\ o&o&o&o&o&\times&\times \\ o&o&o&o&o&o&\times \end{bmatrix} \text{ or } \begin{bmatrix} \times&\times&\times&\times&\times&\times&\times&\times \\ o&\times&\times&\times&\times&\times&\times&\times \\ o&o&\times&\times&\times&\times&\times&\times \\ o&o&o&\times&\times&\times&\times&\times \\ o&o&o&o&\times&\times&\times&\times \\ o&o&o&o&o&\times&\times&\times \\ o&o&o&o&o&o&\times&\times \end{bmatrix} \text{ or } \begin{bmatrix} \times&\times&\times&\times&\times&\times&\times \\ o&\times&\times&\times&\times&\times&\times \\ o&o&\times&\times&\times&\times&\times \\ o&o&o&\times&\times&\times&\times \\ o&o&o&o&\times&\times&\times \\ o&o&o&o&o&\times&\times \\ o&o&o&o&o&o&\times \\ o&o&o&o&o&o&o \end{bmatrix}$$

$$\textit{lower triangular} \text{ is } \begin{bmatrix} \times&o&o&o&o&o&o \\ \times&\times&o&o&o&o&o \\ \times&\times&\times&o&o&o&o \\ \times&\times&\times&\times&o&o&o \\ \times&\times&\times&\times&\times&o&o \\ \times&\times&\times&\times&\times&\times&o \\ \times&\times&\times&\times&\times&\times&\times \end{bmatrix} \text{ or } \begin{bmatrix} \times&o&o&o&o&o&o&o \\ \times&\times&o&o&o&o&o&o \\ \times&\times&\times&o&o&o&o&o \\ \times&\times&\times&\times&o&o&o&o \\ \times&\times&\times&\times&\times&o&o&o \\ \times&\times&\times&\times&\times&\times&o&o \\ \times&\times&\times&\times&\times&\times&\times&o \end{bmatrix} \text{ or } \begin{bmatrix} \times&o&o&o&o&o&o \\ \times&\times&o&o&o&o&o \\ \times&\times&\times&o&o&o&o \\ \times&\times&\times&\times&o&o&o \\ \times&\times&\times&\times&\times&o&o \\ \times&\times&\times&\times&\times&\times&o \\ \times&\times&\times&\times&\times&\times&\times \\ \times&\times&\times&\times&\times&\times&\times \end{bmatrix}$$

These are often square.

BULGED TRIANGULAR A *bulged triangular matrix*, which is also known as a *Hessenberg matrix*, is a triangular matrix with one additional diagonal adjacency to the other elements. These can be rectangular. Like triangular matrices, the upper or lower descriptor tells where the elements are

upper bulged triangular is  or  or 

lower bulged triangular is  or  or 

These are often *almost square* with one additional row or column.

Note that any of these matrix types can be sparse. They can also be combined. An upper bidiagonal is one possible name for an upper-triangular, tridiagonal matrix. It's best to always be explicit with these names though. An upper triangular, tridiagonal might also mean a matrix with the three distinct diagonals in the upper-triangular region specified.

3.6 BLOCK STRUCTURES

BIPARTITE a bipartite matrix has the form

$$\boldsymbol{A} = \begin{bmatrix} 0 & \boldsymbol{B} \\ \boldsymbol{B}^T & 0 \end{bmatrix}$$

for a square or rectangular matrix $\boldsymbol{B}$.

BLOCK DIAGONAL a block diagonal matrix consists of arbitrary matrices in diagonal blocks as in

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{B}_1 & & & \\ & \boldsymbol{B}_2 & & \\ & & \ddots & \\ & & & \boldsymbol{B}_k \end{bmatrix}.$$

3.7 OTHER CLASSES.

There are many other classes and structures that are relevant. An M-matrix is a square matrix where the inverse is non-negative. [10]

Definition 3.8
Let $\boldsymbol{A}$ be a square matrix, then $\boldsymbol{A}$ is an M-matrix if $\boldsymbol{A}^{-1}$ is elementwise non-negative.

[10] There are a few variations on these classes, so this may not be quite right, so I want to double-check.

The class of M-matrices arises frequently when dealing with Markov chains (although that is not why it is called an M-matrix). In fact, the matrix in theorem 3.3 is an M-matrix!

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}^{-1}$$

$$= \begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.9 & 1.8 & 1.6 & 1.4 & 1.2 & 1.0 & 0.8 & 0.6 & 0.4 & 0.2 & 0.1 \\
0.8 & 1.6 & 3.2 & 2.8 & 2.4 & 2.0 & 1.6 & 1.2 & 0.8 & 0.4 & 0.2 \\
0.7 & 1.4 & 2.8 & 4.2 & 3.6 & 3.0 & 2.4 & 1.8 & 1.2 & 0.6 & 0.3 \\
0.6 & 1.2 & 2.4 & 3.6 & 4.8 & 4.0 & 3.2 & 2.4 & 1.6 & 0.8 & 0.4 \\
0.5 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 4.0 & 3.0 & 2.0 & 1.0 & 0.5 \\
0.4 & 0.8 & 1.6 & 2.4 & 3.2 & 4.0 & 4.8 & 3.6 & 2.4 & 1.2 & 0.6 \\
0.3 & 0.6 & 1.2 & 1.8 & 2.4 & 3.0 & 3.6 & 4.2 & 2.8 & 1.4 & 0.7 \\
0.2 & 0.4 & 0.8 & 1.2 & 1.6 & 2.0 & 2.4 & 2.8 & 3.2 & 1.6 & 0.8 \\
0.1 & 0.2 & 0.4 & 0.6 & 0.8 & 1.0 & 1.2 & 1.4 & 1.6 & 1.8 & 0.9 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0
\end{bmatrix}$$

These matrices frequently arise when dealing with stochastic processes where the solution must be a probability, and hence, non-negative.

3.8 PERMUTATION MATRICES

A permutation matrix "shuffles" elements of a vector. Each column of a permutation matrix is a vector $\mathbf{e}_i$ and a permutation matrix must also be orthogonal.

Examples $\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$. This matrix expresses the permutation $2 \to 1, 3 \to 2, 1 \to 3$. We can see this by: $\boldsymbol{A}\begin{bmatrix} 0.5 \\ -0.5 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 1 \\ 0.5 \end{bmatrix}$.

3.9 ORTHOGONAL MATRICES

A matrix is orthogonal if $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{I}$. A key insight about orthogonal matrices is that *they do not change the 2-norm length of a vector.*[11] In this

[11] Formally, the result is let $\boldsymbol{A}$ be orthogonal, then $\|\boldsymbol{A}\mathbf{x}\|_2 = \|\mathbf{x}\|_2$. This is easy to prove so give it a try, but we will show it as well soon.

property, they generalize *rotations*. If I have a vector and simply *rotate* it around the origin, it stays the same length.[12]

The identity matrix $\boldsymbol{I}$ is orthogonal. We'll see more about orthogonal matrices soon – it's a very special structure!

EXERCISES

1. Consider the matrix from theorem 3.4. Devise and/or implement an algorithm to compute a matrix-vector product with this matrix in work that scales with n, instead of n^2, as the problems get bigger.
2. Consider a bipartite matrix $\boldsymbol{A} = \begin{bmatrix} 0 & \boldsymbol{B} \\ \boldsymbol{B}^T & 0 \end{bmatrix}$. Let $\mathbf{x} = \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix}$ be an eigenvector of $\boldsymbol{A}$ with eigenvalue λ. Show that there is an eigenvector $\mathbf{w}$ with eigenvalue $-\lambda$ also involving a combination of $\mathbf{y}$ and $\mathbf{z}$.
3. Show that the product of two diagonal matrices is also diagonal.
4. Show that the product of two upper triangular matrices is upper triangular.
5. Show that multiplying on the left by a diagonal matrix scales the rows.
6. Show that multiplying on the right by a diagonal matrix scales the columns.
7. Show that the inverse of
$$\begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix}$$
is
$$\begin{bmatrix} 1 & -a \\ 0 & 1 \end{bmatrix}.$$
and then, show that the inverse of
$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{A} \\ 0 & \boldsymbol{I} \end{bmatrix}$$
is
$$\begin{bmatrix} \boldsymbol{I} & -\boldsymbol{A} \\ 0 & \boldsymbol{I} \end{bmatrix}.$$
8. *Elementary matrices* Householder, who we will talk about in forthcoming lectures, has a few things named after him. He discussed the idea that any matrix:
$$\boldsymbol{I} - \sigma \mathbf{u}\mathbf{v}^T$$
should be called an elementary matrix. Show when an elementary matrix is invertible and give the inverse.
9. (a) Show that the product of two circulant matrices is circulant.
   (b) Show that the product of two circulant matrices commute with each other. That is, if $\boldsymbol{A}$ and $\boldsymbol{B}$ are circulant matrices, then $\boldsymbol{AB} = \boldsymbol{BA}$.

[12] TODO, get a little tikz picture of this... but maybe put it inline.

10. Let $\boldsymbol{C}$ be an $N \times N$ circulant matrix. Then there is an algorithm to compute $\boldsymbol{C}\mathbf{x}$ in time $O(N \log N)$. This is faster than the time $O(N^2)$ that a straightforward implementation takes. Now let $\boldsymbol{A}$ be an $m \times n$ left or right shift matrix. Show that we can compute $\boldsymbol{A}\mathbf{x}$ in time $O((m+n)\log(m+n))$ by giving an algorithm or procedure to do so.

A MATRIX MODEL OF VIRAL SPREAD 4

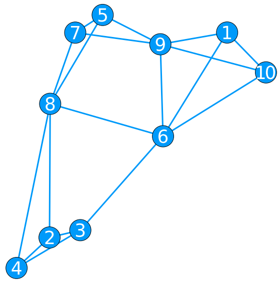
We are going to consider a simple model of a viral spreading process where each person[1]

- can be infected
- is infected.

We are also going to consider "time" where "time" represents some regular period such as a day or week. We also assume that each person's spreadable contacts are the same over that time period; a contact is spreadable if you see them long enough to possibly spread a virus to them. Put another way, "time" is long enough so that we see the same group of people over that period. So this isn't the group of everyone you see, but everyone you see long enough to possibly exchange viral material with!

These notes were written during the Fall 2020 semester of the COVID-19 pandemic. So called "armchair epidemiologists" were everywhere and the time called for everyone to be able to understand spread and policy and a host of complex issues. These notes should not be used for "armchair epidemiology" but rather to understand how the tools from this class might manifest in such a scenario.

The contacts among our people define a network or graph. Each node is a person. The edges of the network represent the spreadable contacts. Here's an example.



$$A = \begin{bmatrix} 0&0&0&0&0&1&0&0&1&1\\ 0&0&1&1&0&0&0&1&0&0\\ 0&1&0&1&0&1&0&0&0&0\\ 0&1&1&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&1&1&0\\ 1&0&1&0&0&0&0&1&1&1\\ 0&0&0&0&1&0&0&1&1&0\\ 0&1&0&1&1&1&1&0&0&0\\ 1&0&0&0&1&1&1&0&0&1\\ 1&0&0&0&0&1&0&0&1&0 \end{bmatrix}$$

FIGURE 4.1 – A small contact network with the adjacency matrix for the network.

Note here that the non-zero entries in the adjacency matrix correspond to edges. So $A(1,10)$ and $A(10,1)$ are both one because person 1 and person 10 are contacts. Now, at the moment, it's *premature* to call $\boldsymbol{A}$ a matrix. Right now, it's just a table of data that collects information on edges. But, we'll soon see the term *matrix* is appropriate.

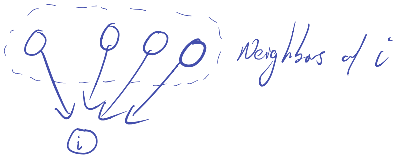
Back to the virus and how it spreads. We further assume a contact will cause an infection with probability $0 < \rho < 1$.[2]

4.1 A PROBLEMATIC BUT USEFUL STARTING MODEL

The probability that a person i is infected at time $t+1$ is the probability that i got the infection from a contact at time t. This corresponds with the following probability scenario.

[1] In standard mathematical epidemiology literature, this would be a susceptible, infected (SI) model. This is highly simplistic!

[2] If you see someone more often that you want to increase this probability for *some contacts*, the model we have would allow you do to this! Seems like a good homework problem to figure out where!



Each contact j in the neighbors of i infects i based on a simple random trial that occurs with probability $\rho P(j \text{ is infected at time } t)$. This is a simple exercise in probability.[3] What that reference explains is that it is easiest to look at the probability that i is not infected. Which corresponds with *all* of the "infection attempts" failing. We assume these are independent, so the failure to be infected is just the probability

$$\prod_{j \in \text{neighbors}(i)} (1 - \rho P(j \text{ is infected at time } t)).$$

This makes sense. If any neighbor is infected with probability 1 and $\rho = 1$, then you will be infected, so this quantity will be 0 (so there no chance you are not infected.) The probability that i is infected is simply the complement:

$$P(i \text{ is infected at time } t+1) = 1 - \prod_{j \in \text{neighbors}(i)} (1 - \rho P(j \text{ is infected at time } t)).$$

We then evaluate this for all i.

This describes a very simple evolution in terms of the adjacency matrix $\boldsymbol{A}$ that is easiest to explain in terms of code. Let $\mathbf{x}$ be the vector $P(i \text{ is infected at time } t)$ for all i and $\mathbf{y}$ be the vector $P(i \text{ is infected at time } t+1)$ for all i. Then

```
function evolve(x::Vector, p::Real, A::AbstractMatrix)
  log_not_infected = log.(1 .- p.*x)
  y = 1 .- exp.(A*log_not_infected)
  y = max.(y, x)
end
```

Here, we are using the product is the exponentiated sum of logs. Consequently, we can simultaneously evaluate *all* of the probabilities by taking the log and then *summing* using the adjacency matrix. This is because

$$[\boldsymbol{A}\mathbf{x}]_i = \sum_{j \in \text{neighbors(i)}} x_j.$$

The final max is useful if you have a boundary condition with a set of definitely infected nodes, but this could also be omitted.[4]

An interesting aside.

When I ran this, initially, I thought this would converge to all probabilities of 1. This does not happen. Instead it converges to a steady state I

[3] See https://www.khanacademy.org/math/ap-statistics/probability-ap/probability-multiplication-rule/a/probabilities-involving-at-least-one-success.

[4] Play around with it and see. It is a model, not a commandment! The idea is to modify it and understand what happens.

can't quite explain. A steady state corresponds with

$$\log(1 - P(i)) = \sum_{j\in\text{neighbors}(i)} \log(1 - \rho P(j)).$$

It is totally unclear to me *why* and *how* this iteration ought to converge and why this fixed point ought to exist. But it does—reliably so!

An approximation

But here is where linear algebra comes into play. Suppose we make the reasonable approximation that $\rho P(j \text{ is infected at time } t)$ is small. This means the chance of getting this from an arbitrary interaction is small. This is plausible at the start of an infection. Then that our expression looks like

$$(1-a)\cdot(1-b)\cdot(1-c)\cdots.$$

If a, b, c are fairly small, then products ab are even smaller, so we could use the approximation:

$$(1-a)\cdot(1-b)\cdot(1-c) \approx 1 - a - b - c.$$

Applied to our expression, this gives: Then note that

$$\prod_{j\in\text{neighbors}(i)} (1-\rho P(j \text{ is infected at time } t)) \approx 1-\rho \sum_{j\in\text{neighbors}(i)} P(j \text{ is infected at time } t)$$

This suggests an even simpler iteration.

```
function evolve_approx(x::Vector, p::Real, A::AbstractMatrix)
  y = p.*(A*x)
end
```

This is just a repeated matrix vector product! If $\mathbf{x}^{(t)}$ is the set of probabilities from this approximation at the tth step, then

$$\mathbf{x}^{(t+1)} = \rho \boldsymbol{A}\mathbf{x}^{(t)} = (\rho \boldsymbol{A})^{t+1}\mathbf{x}^{(0)}$$

where $\mathbf{x}^{(0)}$ is the *start* of everything.

4.2 FIXING THE PROBLEM

But there is a problem in the above formulation. This was hinted at in the interesting aside. If you get the infection with probability ρ, then over enough time, everyone would become infected. The probabilities in either model above, though, do not go to 1. This is because we forgot a piece: you *infect yourself* based on the probability in the prior iteration.

The adjustment is simple

$$\begin{aligned} &P(i \text{ is infected at time } t+1) \\ &\quad = \Big(1 - \underbrace{\prod_{j\in\text{neighbors}(i)} (1 - \rho P(j \text{ is infected at time } t)))}_{\text{infected via neighbors}} \underbrace{(1 - P(i \text{ is infected at time } t))}_{\text{actually infected in the previous step.}} \end{aligned}$$

and

```
function evolve_with_self(x::Vector,  p::Real, A::AbstractMatrix)
  log_not_infected = log.(1 .- p.*x)
  y = (1 .- exp.(A*log_not_infected).*(1 .- x))
  y = max.(y, x)
end
```

In this new model, the probabilities always go to one.[5]

The approximation again

Let's use that same idea and approximation to understand what will happen when the probabilities are small. Applying this to the adjusted formulation

$$P(i \text{ is infected at time } t+1) \approx \rho \sum_{j \in \text{neighbors}(i)} P(j \text{ is infected at time } t) + P(i \text{ is infected at time } t).$$

```
function evolve_with_self_approx(x::Vector,  p::Real, A::AbstractMatrix)
  y = rho*(A*x) + x
end
```

This is also just repeated matrix vector products, but with the matrix $\rho\boldsymbol{A} + \boldsymbol{I}$ instead of $\rho\boldsymbol{A}$. As in, if again $\mathbf{x}^{(t)}$ is the set of probabilities from this approximation at the tth step, then

$$\mathbf{x}^{(t+1)} = (\rho\boldsymbol{A} + \boldsymbol{I})\mathbf{x}^{(t)} = (\rho\boldsymbol{A} + \boldsymbol{I})^{t+1}\mathbf{x}^{(0)}$$

where $\mathbf{x}^{(0)}$ is the *start* of everything.

4.3 THE EIGENANALYSIS

As we will see in this class, the *eigenvectors of $\boldsymbol{A}$* determine the behavior of both powers of $\rho\boldsymbol{A}$ and $(\rho\boldsymbol{A}+\boldsymbol{I})$ as the powers get large. This information then suggests how epidemics spread on networks and a variety of other related behaviors.

4.4 A SLIGHTLY DIFFERENT MODEL

These models are not commandments. They encode slightly different and related ideas. Here's another way to understand this. What we are doing in the first (incorrect) model is evaluating the probability that node i is infected *by neighbors* at time t. Let $n_i^{(t)} = P(i \text{ is infected via neighbors at time } t)$. Then we have

$$n_i^{(t+1)} = 1 - \prod_{j \in \text{neighbors}(i)} (1 - \rho n_j^{(t)}) \approx \rho \sum_{j \in \text{neighbors}(i)} n_j^{(t)}.$$

As a matrix-vector iteration, the approximation is

$$\mathbf{n}^{(t+1)} = \rho\boldsymbol{A}\mathbf{n}^{(t)} = (\rho\boldsymbol{A})^{t+1}\mathbf{n}^{(0)}.$$

But what is $\mathbf{n}^{(0)}$, the starting condition? This has to do with what is often called a *boundary condition*. If we are in a scenario like the US, where the

[5] This is not a complicated argument, but it isn't the focus on this class.

virus is *everywhere* then we can reasonably set $\mathbf{n}^{(0)}$ to be a small constant to model the scenario where everyone has some small chance of being infected. Alternatively, if we are in a contact tracing scenario or a test and trace scenario like Purdue is trying to do, we would remove the contacts from the network that we know are infected and look at the probability that. Here, we simply take any nodes we *know* are infected, remove them from the network, but evaluate the probability that they infect their neighbors. For simplicity, suppose there is one node z infected. Then we set $n_j^{(0)} = \rho$ if j is a neighbor of z and 0 otherwise. The matrix $\boldsymbol{A}$ for this second scenario *does not include z*.

Now, this models transmission, but we know to know infection probabilities. These are just given by

$$P(i \text{ is infected by time } t) = 1 - \prod (1 - n_i^{(t)}) \approx \sum_{\ell=0}^{t} n_i^{(t)}.$$

Let $\mathbf{x}^{(t)}$ be the probability that i is infected by time t above for all nodes. Then we have

$$\mathbf{x}^{(t)} \approx \sum_{\ell=0}^{t} \mathbf{n}^{(t)} \approx \sum_{\ell=0}^{t} (\rho \boldsymbol{A})^{\ell} \mathbf{n}^{(0)}.$$

This last expression is known as a Neumann series, and admits a closed form solution when ρ is sufficiently small.[6] If ρ is small enough then

$$\sum_{\ell=0}^{\infty} (\rho \boldsymbol{A})^{\ell} \mathbf{n}^{(0)} = (\boldsymbol{I} - \rho \boldsymbol{A})^{-1} \mathbf{x}^{(0)}.$$

Put another way, your chance of ever being infected via neighbors from a starting set of probabilities is given by the the solution of a linear system of equations whose right hand side is the initial set of probabilities:

$$(\boldsymbol{I} - \rho \boldsymbol{A}) \mathbf{x}^{(\infty)} = \mathbf{n}^{(0)}.$$

This is something known as Katz centrality Katz [1953] that was derived to understand *social structure*. but again shows how simple matrix systems arise in a problem that is relevant to the times!

The idea to relate eigenvectors or Katz scores and epidemics arose, most recently, from a Tweet Dan Larrenmore at UC Boulder sent about how these determine sampling probabilities in one of their COVID testing papers Larremore et al. [2020]. In other scenarios, these also are called replicator dynamics from Rumi Ghosh Ghosh et al. [2014], which better model viral phenomenon. There is more history on understanding matrix diffusions with $\boldsymbol{A}$ as *non-conservative* diffusions that *create mass*, but I've forgotten where all I've read about this and how much (if any) I liked explaining in alternative ways. Some earlier references I'm aware of that would have inspired these thoughts are Saberi's work on (computer) viral spreading and eigenvalues Berger et al. [2005].

[6] Later in class, we'll see that ρ has to be smaller than the largest magnitude eigenvalue of $\boldsymbol{A}$.

CANDYLAND & WORKING WITH SPARSE MATRICES 5

Learning objectives
Learn how to perform basic operations with sparse matrices using the Candyland matrix as an example.

5.1 INTRO TO CANDYLAND

As we mentioned, there are many real-world problems that involve sparse matrices. In a few lectures, we'll see how discretizations of Laplacian operator on 2d grids will give us sparse matrices. For this class, we are going to continue working with random processes.

The game of Candyland is played on 133 squares. At each turn, a player draws a card from a deck of cards. This determines where they move to for the next turn. There is no interaction with other players (other than sharing the same deck). For our study here, we are going to model the game where we simply draw a random card from the total set at each time, so there is no memory at all in the game. This means that the resulting system is a Markov chain, or a memoryless stochastic process. While there is a great deal of formality we can get into with all of these things, the key thing to remember is that *what happens at each step can just be described by a matrix that tells you the probability of what happens next.*

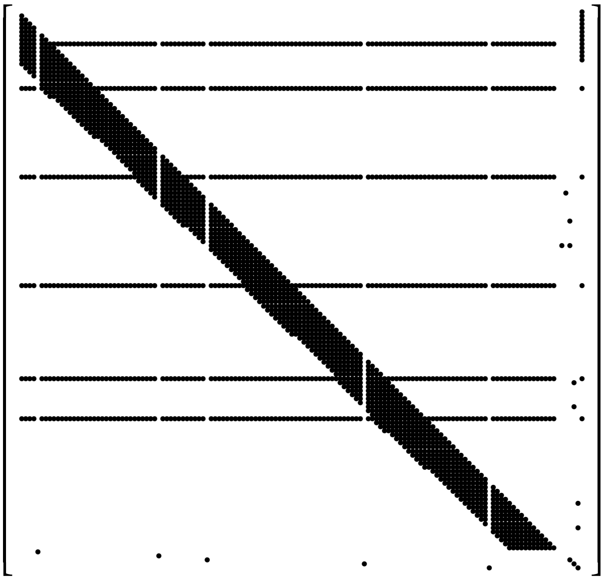
5.1.1 *The Candyland Model*

So we are going to create the matrix for Candyland that gives the probabilities of moving between the 133 squares, along with two special squares; one for the start (state 140) and one for the destination (134) . There are also a set of 5 special cases that involve exceptions to the rules (135,136,137,138,139).

In this case, the game of Candyland can be modeled with a 140×140 matrix $\boldsymbol{T}$.[1] If we show the matrix with a small • for each nonzero entry, then it looks like[2]

[1] The data files to recreate $\boldsymbol{T}$ are available on the the course website.

[2] TODO – Double check this one isn't transposed.

$\boldsymbol{T} =$  .

This is clearly sparse as most of the matrix is empty. This is because it's impossible to get between most pairs of squares in Candyland in a single move.

Let $\boldsymbol{T} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & \dots & \mathbf{t}_{140} \end{bmatrix}$ be the column-wise partition. Where $\mathbf{t}_j$ describes the probability of ending up at each of the 140 states given that we are in state j. Put another way, $T(i,j)$ is the probability of moving to state i given that you are in state j. Consequently, after one step of the game, the probability that the player is in any state can be read off from $\mathbf{t}_{140}$. This is because the player starts in state 140.

Now, what's the probability of being in any state after two-steps? We can use the matrix to work out this probability:

$$\begin{aligned}
&\text{Probability that player is in state } i \text{ after two steps}\\
&\quad = \sum_k \text{Probability that player is in state } k \text{ after one step and moves from } k \text{ to } i.\\
&\quad = \sum_k T(i,k)\mathbf{t}_{140}(k)
\end{aligned}$$

If we do this for all i, then we find that

$$\mathbf{p}_2 = \boldsymbol{T}\mathbf{t}_{140}$$

is the probability of the player being in any state after two steps. This is just a matrix-vector operation!

Now to figure out where the player is after any number of steps, we proceed iteratively:

$$\begin{aligned}
&\text{Probability that player is in state } i \text{ after three steps}\\
&\quad = \sum_k \text{Probability that player is in state } k \text{ after two steps and moves from } k \text{ to } i.\\
&\quad = \sum_k T(i,k)\mathbf{p}_2(k).
\end{aligned}$$

Again, by grouping everything together, we get:

$$\mathbf{p}_3 = \boldsymbol{T}\mathbf{p}_2 = \boldsymbol{T}^2\mathbf{t}_{140}.$$

By induction now, we get that the probability the player is in any state after k steps:

$$\mathbf{p}_k = \boldsymbol{T}^{k-1}\mathbf{t}_{140}.$$

The key point: in order to compute this probability, we only need to compute matrix-vector products with a sparse matrix.

5.1.2 *Computing expected length of a Candyland game*

The Candyland game ends when then the player is in state 134 in this particular model. Let X be the random variable that is the length of the Candyland game. Then we want to compute the expected value of X. Recall that the expected value of a discrete random variable is:

$$E[X] = \sum\nolimits_{i \text{ where } i \text{ is any possible value of } x} i \cdot (\text{probability that } X = i).$$

The probability that the game ends in 5 steps is[3]

$$[\boldsymbol{T}^4\mathbf{t}_{140}]_{134}.$$

Hence, the expected length of the Candyland game is:

$$E[X] = \sum_{i=1}^{\infty} i \cdot [\boldsymbol{T}^{i-1}\mathbf{t}_{140}]_{134}.$$

In practice, we can't run this until infinity, even though the game could, in theory, last a very long time. We can compute this via the following algorithm. [4]

```
Create a starting vector $\mathbf{p} = \mathbf{e}_{140}$ because we start in state 140.
EX ← 0
For length = 1 to maximum game length considered
    $\mathbf{p} \leftarrow \boldsymbol{T}\mathbf{p}$
    EX ← EX + length·$\mathbf{p}_{134}$
return EX
```

The key algorithm step is to compute the matrix-vector product $\boldsymbol{T}\mathbf{p}$

5.2 SPARSE MATRIX STORAGE: STORING ONLY THE VALID TRANSITIONS FOR CANDYLAND AND PERFORMING A MATRIX-VECTOR PRODUCT

The idea with sparse matrix storage is that we only store the nonzero entries of the matrix. Anything that is not stored is assumed to be zero. This is illustrated in the following figure.

[3] This is the 134 entry in the vector $\boldsymbol{T}^4\mathbf{t}_{140}$.

[4] In Julia, the code is

```
function candylandlength(T, ma
  n = size(T,1)
  (p = zeros(n))[140] = 1
  ex = 0.0
  for l=1:len
    p = T*p
    ex += length*p[134]
  end
  return ex
end
```

$$\begin{bmatrix} 0 & 16 & 13 & 0 & 0 & 0 \\ 0 & 0 & 10 & 12 & 0 & 0 \\ 0 & 4 & 0 & 0 & 14 & 0 \\ 0 & 0 & 9 & 0 & 0 & 20 \\ 0 & 0 & 0 & 7 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Indexed storage

I	3	3	5	5	2	1	2	4	4	1
J	5	2	6	4	3	3	4	6	3	2
V	14	4	4	7	10	13	12	20	9	16

The arrays I, J, and V store the row index, column index, and nonzero value associated with each nonzero entry in the matrix. There are 30 values in the arrays, whereas storing all the entries in the matrix would need 36 values. Although this isn't a particularly large difference, is is less data.

For the matrix $\boldsymbol{T}$ in the Candyland problem, there are 6816 entries in the arrays I, J, V whereas there would be 19600 entries in the matrix $\boldsymbol{T}$ had we stored all the zeros.

We can use this data structure to implement a matrix-vector product. Recall that

$$\mathbf{y} = \boldsymbol{A}\mathbf{x} \text{ means that } y_i = \sum_{j=1,\ldots,n} A_{i,j} x_j \text{ for all } i \,.$$

If $A_{i,j} = 0$ then it plays no role in the final summation and we can write the equivalent expression:

$$\mathbf{y} = \boldsymbol{A}\mathbf{x} \text{ means that } y_i = \sum_{j \text{ where } A_{i,j} \neq 0} A_{i,j} x_j \text{ for all } i \,.$$

This means that an algorithm simply has to implement this accumulation over all *nonzero* entries in the matrix. This is exactly what is stored in the arrays I, J, V.

The algorithm in Julia is:

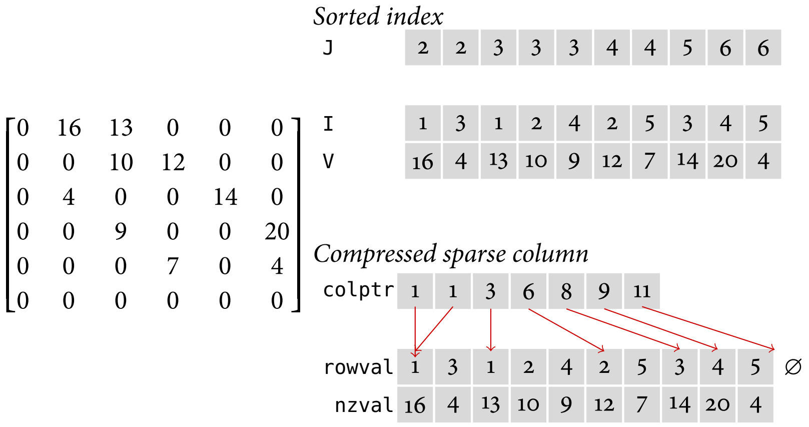
```julia
function indexed_sparse_matrix_vector_product(x,I,J,V,m,n)
  y = zeros(m)
  for nzi=1:length(I)
    i,j,v = I[nzi], J[nzi], V[nzi]
    y[i] += v*x[j]
  end
  return y
end
```

This algorithm can be translated to many other languages too.

5.3 ELIMINATING REDUNDANT DATA STORAGE: COMPRESSED SPARSE ROW AND COLUMN FORMATS

The idea with compressed sparse column storage is that some of the information in the full indexed information is redundant *if* we sort all the data by column.[5]

[5] We can also sort by row. That discussion is next.



This figure shows that when the data are sorted by increasing column index. Then there are multiple values with the same column in adjacent entries of the J array. We can *compress* these into a list of pointers. This means that we create a new array called `colptr` that stores the starting index for all the entries in I, V arrays associated with a given column.

$$\text{Entries of column } j \text{ are stored in } \begin{array}{l} \texttt{rowval}[\texttt{colptr}[j]] \ldots \texttt{rowval}[\texttt{colptr}[j+1]-1] \\ \texttt{nzval}[\texttt{colptr}[j]] \ldots \texttt{nzval}[\texttt{colptr}[j+1]-1] \end{array} .$$

This means if $\texttt{colptr}[j] = \texttt{colptr}[j+1]$ then there are no entries in the column. (See the example in column 1.) This

This structure enables efficient iteration over the elements of the matrix for matrix-vector products, just like indexed storage, with only minimal changes to the loop. In Julia, the algorithm is:[6]

```
function indexed_sparse_matrix_vector_product(x,colptr,rowval,nzval,m,n)
  y = zeros(m)
  for j=1:n
    for nzi=colptr[j]:colptr[j+1]-1
      i,v = rowval[nzi], nzval[nzi]
      y[i] += v*x[j]
    end
  end
  return y
end
```

Both Julia and Matlab use compressed sparse column formats for their preferred sparse matrix format.

ADVANTAGES OF COMPRESSED SPARSE COLUMN COMPARED WITH INDEXED STORAGE.

- less data / memory storage
- allows random access to column data
- allows per-column operations to be more efficient

[6] This algorithm is especially particular to using 0 based or 1-based indexing.

5.4 COMPRESSED SPARSE ROW

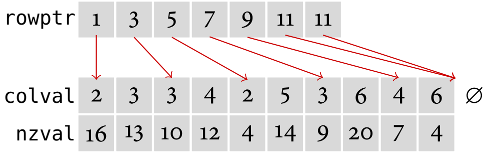
The compressed sparse row format adopts compression of the rows. If we sort row indices from an indexed format, then we can compress them into pointers just as in the compressed sparse column format.

Sorted row index

I	1	1	2	2	3	3	4	4	5	5
J	2	3	3	4	2	5	3	6	4	6
V	16	13	10	12	4	14	9	20	7	4

$$\begin{bmatrix} 0 & 16 & 13 & 0 & 0 & 0 \\ 0 & 0 & 10 & 12 & 0 & 0 \\ 0 & 4 & 0 & 0 & 14 & 0 \\ 0 & 0 & 9 & 0 & 0 & 20 \\ 0 & 0 & 0 & 7 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Compressed sparse row



rowptr	1	3	5	7	9	11	11				
colval	2	3	3	4	2	5	3	6	4	6	∅
nzval	16	13	10	12	4	14	9	20	7	4	

Entries of row i are stored in $\begin{array}{l} \texttt{colval}[\texttt{rowptr}[i]] \ldots \texttt{colval}[\texttt{rowptr}[j+1]-1] \\ \texttt{nzval}[\texttt{rowptr}[j]] \ldots \texttt{nzval}[\texttt{rowptr}[j+1]-1] \end{array}$.

If we were to implement the matrix-vector routine for compressed sparse row matrices, however, there is an interesting optimization possible because *all of the updates* to the output vector $\mathbf{y}$ happen in the same index.

```
function indexed_sparse_matrix_vector_product(x,rowptr,colval,nzval,m,n)
  y = zeros(m)
  for i=1:m
    yi = 0.0
    for nzi=rowptr[j]:rowptr[j+1]-1
      j,v = colval[nzi], nzval[nzi]
      yi += v*x[j]
    end
    y[i] = yi
  end
  return y
end
```

An important advantage of this CSR structure is that is is possible to parallelize the sparse matrix vector routine over the rows of the matrix.

5.5 DISADVANTAGES OF COMPRESSED STORAGE FORMATS

A major disadvantage of compressed sparse column and compressed sparse row formats is that they cannot be easily altered once created.[7] Adding a new nonzero element inside the matrix requires rebuilding the entire array. (Unless it is at the last column!) For this reason, a common

[7] If the number of elements that will change are known ahead of time, then an alternative is to simply insert them into the matrix structure with a "0" placeholder value. It is okay to have *zero* entries in the data structures, and it is okay to alter the values associated with each nonzero element.

paradigm is to use *indexed format* while creating the information for your matrix and then only convert to *compressed sparse* formats when it is time to analyze the matrix and it will be fixed for a reasonably long period of time.

Another disadvantage is that we often want to have random access to both rows and columns of a matrix. Compressed sparse column gives efficient random access to columns; compressed sparse row gives efficient random access to rows. But finding all the information for a given row in a compressed sparse column structure involves searching over all the elements. If both random row and random column access are needed, the *easiest* solution is simply to store the matrix both in CSC and CSR formats. This doubles the storage space.[8]

5.6 ALTERATIVE FORMATS

Most programming languages have a standard *hash table* or *dictionary* implementation.[9] These allow *arbitrary* key-value pairs to be inserted and give fast access and fast update times. This can be used as a sparse matrix data structure by using the key as an index tuple and the value as the non-zero value. This allows fast insertion and deletion of elements. It does not allow fast random access to rows and columns.

For fast insertion and fast random access to rows and columns, then we can use an array of hash tables.[10]

5.7 OPERATIONS WITH CSC MATRICES

COMPUTING THE TRACE

EXTRACTING A ROW

EXTRACTING A COLUMN

MULTICOLUMN MULTIPLICATION

SPARSE-MATRIX BY SPARSE-MATRIX MULTIPLICATION This is one of the most intricate procedures. We devote an entire future section to it.

[8] Note that storing a matrix in CSR can be accomplished by storing the transpose in CSC. Likewise, storing a matrix in CSC can be done by storing the *transpose* in CSR.

[9] In Julia, these are `Dict` types. In Python, these are also called dictionaries. In C++ the type is `unordered_map`. The matrix type in Julia would be: `Dict{Tuple{Int,Int},Float64()}` for `Float64` values in the matrix. This idea is implemented in the package `SparseMatrixDicts.jl`

[10] In Julia, the type would be `Vector{Dict{Int,Float64}}[]`.

6 MATRIX & VECTOR NORMS

Norms are used to measure the size of vectors and matrices. They are generalizations of the scalar function $|x|$, which determines the size or magnitude of a scalar value. For instance, if x is close to y, then we have $|x - y|$ is close to zero. In fact, if we take the idea of a vector norm and specialize it to a scalar, then we will reproduce exactly a scaled absolute value function (see exercise 2).

Learning objectives

1. Examples of vector norms.
2. Examples of matrix norms.
3. The submultiplicative property of a matrix norm.
4. The property that *all norms are equivalent*

So far, we have used the 2-norm of a vector. Let's work with them formally.

6.1 VECTOR NORMS

We begin with the two norm.

Definition 6.1
The Euclidean norm or 2-norm of a vector[1] is

$$\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} |x_i|^2} = \sqrt{\mathbf{x}^T\mathbf{x}}$$

This can be generalized a p-norm.

Definition 6.2
The p-norm of a vector is

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}.$$

This isn't created to generalize for generalizations sake. One of the common uses of norms is to argue that a sequence of vectors

$$\mathbf{x}_k \to \mathbf{y}$$

which can be handled by showing

$$\|\mathbf{x}_k - \mathbf{y}\| \to 0.$$

Depending on the value of p, this can be easy or difficult. For instance, when $p = 1$, then this is simply a sum of absolute values:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$$

[1] This definition includes absolute values. Yet, $x_i^2 \geq 0$ for all real values. We leave the absolute values because this then generalizes to complex values where we need a complex magnitude.

[2] It is a useful exercise to convince yourself that as $p \to \infty$, then the value of the norm will simply be the largest element by magnitude.

and $p = \infty$ can be defined via a limit:[2]

$$\|\mathbf{x}\|_\infty = \max_{i=1}^{n} |x_i|.$$

Now, we are going to define an extremely general notion of norm in order to state a few important results.

Definition 6.3
A vector norm on $\mathbf{x} \in \mathbb{R}^n$ is any function $f(\mathbf{x}) \to \mathbb{R}$ that satisfies:

1. $f(\mathbf{x}) \geq 0$ (*non-negative*)
2. $f(\mathbf{x}) = 0$ if and only if $\mathbf{x} = 0$ (*zero-sensitive*)
3. $f(\alpha\mathbf{x}) = |\alpha| f(\mathbf{x})$ for any scalar α (*linear scale or 1-homogeneous*),
4. $f(\mathbf{x}+\mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y})$ (*triangle inequality*).

Any p-norm with $p \geq 1$ satisfies these definitions. When $p < 1$, then we violate the triangle inequality.

There are some crazy norms too. For instance, the following function satisfies these three criteria:

$$f(\mathbf{x}) = \text{sum of largest two entries in } \mathbf{x} \text{ by magnitude.}$$

All norms are equivalent

Given that the idea of a norm can be very general. We wish to immediately show that for any result about *convergence to zero*, it does not matter which norm is used.

The following theorem guarantees that if $\mathbf{x}_k \to \mathbf{y}$ for any norm, then it will happen for all norms.

Informally, this theorem is known as the *all norms are equivalent* theorem.

Note that I think this is $1/C_1$ vs. C_1 the way it's defined.

Theorem 6.4 (All norms are equivalent)
Formally, let $f(\mathbf{x})$ and $g(\mathbf{x})$ be any pair of vector norms on $\mathbb{R}^n$, then there exist positive constants $L \leq U$ such that

$$Lf(\mathbf{x}) \leq g(\mathbf{x}) \leq Uf(\mathbf{x}).$$

Note that these constants can depend on the dimension n.

PROOF First, note that the theorem is immediately true if $\mathbf{x} = 0$ for any values of L and U. So in the remainder, we can can study the case where $\mathbf{x} \neq 0$ and hence, $f(\mathbf{x}) > 0$ and $g(\mathbf{x}) > 0$. To show the upper bound, this is equivalent to

$$\frac{f(\mathbf{x})}{g(\mathbf{x})} \leq U \quad \Leftrightarrow \quad \frac{\frac{1}{g(\mathbf{x})} f(\mathbf{x})}{\frac{1}{g(\mathbf{x})} g(\mathbf{x})} \leq U$$

We can now rewrite this as $f(\mathbf{y}) \leq U$ where $g(\mathbf{y}) = 1$ by using the linear scale property because $\frac{1}{g(\mathbf{x})} f(\mathbf{x}) = f(\mathbf{x}/g(\mathbf{x}))$ is *always* computed with a vector where the g-norm is 1. Hence we set

$$U = \begin{array}{ll} \underset{\mathbf{y}}{\text{maximize}} & f(\mathbf{y}) \\ \text{subject to} & g(\mathbf{y}) = 1. \end{array} \qquad \blacksquare$$

Now, U is finite because it is the value of f for some point in a closed set.

For instance, $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n\|\mathbf{x}\|_\infty$ is an instance with $C_1 = 1, C_2 = n$.

Quiz What are C_1 and C_2 such that $C_1\|\mathbf{x}\|_1 \leq \|\mathbf{x}\|_\infty \leq C_2\|\mathbf{x}\|_1$?

Consequently, suppose, for a vector norm $f(\mathbf{x})$, you show that $f(\mathbf{x}_k - \mathbf{y}) \to 0$. Then we know that $C_2 f(\mathbf{x}) \geq g(\mathbf{x})$ and also that $C_2 f(\mathbf{x}_k - \mathbf{y}) \to 0$. Since $g(\mathbf{x}) \geq 0$, then we must have $g(\mathbf{x}_k - \mathbf{y}) \to 0$ as well.

6.2 MATRIX NORMS

Vector norms measure the size or magnitude of a vector. Matrix norms do the same for a matrix. There are two important types of matrix norms: element-wise (or Frobenius norms) and operator norms. Just like vector norms, there is a general condition for all matrix norms.

Definition 6.5
A matrix norm on $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ is any function $f(\boldsymbol{X}) \to \mathbb{R}$ that satisfies:

1. $f(\boldsymbol{X}) \geq 0$ (*non-negative*)
2. $f(\boldsymbol{X}) = 0$ if and only if $\boldsymbol{X} = 0$ (*zero-sensitive*)
3. $f(\alpha\boldsymbol{X}) = |\alpha| f(\boldsymbol{X})$ for any scalar α (*1-homogeneous*)
4. $f(\boldsymbol{X} + \boldsymbol{Y}) \leq f(\boldsymbol{X}) + f(\boldsymbol{Y})$ (*triangle inequality*).

6.2.1 *Element-wise norms*

Note that if vec $\boldsymbol{X}$ is any way of turning $\boldsymbol{X}$ into a vector by organizing the mn elements of $\boldsymbol{X}$ into a single array, then $f(\text{vec}(\boldsymbol{X}))$ is a matrix norm for any vector norm $f(\mathbf{x})$. These are called element-wise norms. The most common of which is the Frobenius norm.

Definition 6.6
The Frobenius norm of a matrix is

$$\|\boldsymbol{X}\|_F = \sqrt{\sum_{ij} |X_{ij}|^2} = \|\operatorname{vec}(\boldsymbol{X})\|_2 = \sqrt{\operatorname{trace}(\boldsymbol{A}^T\boldsymbol{A})}.$$

Here, we used $\operatorname{trace}(\boldsymbol{A}) = \sum_{i=1}^{\min(m,n)} A_{i,i}$, which is the sum of diagonal entries.

6.2.2 *Operator-induced norms*

Let $f(\mathbf{x})$ be any vector norm, then we can define a matrix norm via:

$$f(\boldsymbol{X}) = \underset{\mathbf{x}\neq 0}{\text{maximize}} \quad f(\boldsymbol{A}\mathbf{x})/f(\mathbf{x}) \ .$$

Proof that $f(\boldsymbol{X})$ is a matrix norm

1. $f(\boldsymbol{X}) \geq 0$ because f is a vector norm. 2. If $f(\boldsymbol{X}) = 0$, then $f(\boldsymbol{A}\mathbf{x})/f(\mathbf{x}) = 0$ for all vectors $\mathbf{x} \neq 0$. Since $f(\mathbf{e}_i) > 0$, then we must have $f(\boldsymbol{A}\mathbf{e}_i) = 0$ for all $\mathbf{e}_i$, so the matrix is entirely empty. Also, if $\boldsymbol{A} = 0$, then $\boldsymbol{A}\mathbf{x} = 0$ for any $\mathbf{x}$, and so $f(\boldsymbol{A}) = 0$. 3. $f(\alpha\boldsymbol{X}) = \underset{\mathbf{x}\neq 0}{\text{maximize}} \quad f(\alpha\boldsymbol{A}\mathbf{x})/f(\mathbf{x}) = |\alpha| f(\boldsymbol{X})$. 4. Note that $f((\boldsymbol{X}+\boldsymbol{Y})\mathbf{x}) \leq f(\boldsymbol{X}\mathbf{x}) + f(\boldsymbol{Y}\mathbf{x})$ be the vector-norm triangle inequality. Hence,

$$\begin{aligned} f(\boldsymbol{X}+\boldsymbol{Y}) = \underset{\mathbf{x}\neq 0}{\text{maximize}} \quad f((\boldsymbol{X}+\boldsymbol{Y})\mathbf{x})/f(\mathbf{x}) &\leq \underset{\mathbf{x}\neq 0}{\text{maximize}} \quad f(\boldsymbol{X}\mathbf{x})/f(\mathbf{x}) + f(\boldsymbol{Y}\mathbf{x})/f(\mathbf{x}) \\ &\leq \underset{\mathbf{x}\neq 0}{\text{maximize}} \quad f(\boldsymbol{X}\mathbf{x})/f(\mathbf{x}) + \underset{\mathbf{x}\neq 0}{\text{maximize}} \quad f(\boldsymbol{Y}\mathbf{x})/f(\mathbf{x}) \\ &\leq f(\boldsymbol{X}) + f(\boldsymbol{Y}) \end{aligned}$$

The operator induced norms are harder to reason about.

Let $f(\mathbf{x}) = \|\mathbf{x}\|_1$, then

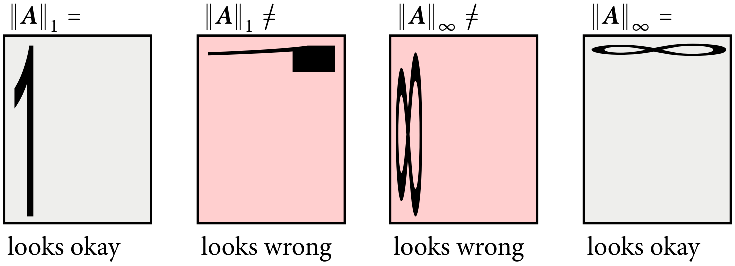
$$\|\boldsymbol{A}\|_1 = \max_{j=1}^{n} \sum_{i=1}^{m} |A_{ij}|$$

which is the maximum column 1-norm. If, instead, $f(\mathbf{x}) = \|\mathbf{x}\|_\infty$, then

$$\|\boldsymbol{A}\|_\infty = \max_{i=1}^{m} \sum_{j=1}^{n} |A_{ij}|$$

which is the maximum row 1-norm.

Here's my picture to remember these.


$\|\boldsymbol{A}\|_1 =$
$\|\boldsymbol{A}\|_1 \neq$
$\|\boldsymbol{A}\|_\infty \neq$
$\|\boldsymbol{A}\|_\infty =$
looks okay
looks wrong
looks wrong
looks okay

6.2.3 *Additional matrix norms*

This section can be skipped on a first reading.

There is a wide additional class of norms defined in terms of the singular values of a matrix. See other sections on the singular values and their definitions.[3]

An $m \times n$ real-valued or complex-valued matrix has $\min(m, n)$ non-negative *real* singular values. Let $\sigma_1, \ldots, \sigma_{\min(m,n)}$ be the singular values of a $m \times n$ matrix with $m \geq n$.

[3] TODO Insert reference when assembled into bigger document.

Definition 6.7 (The Nuclear Norm, the Trace Norm)
Let $\sigma_1, \ldots, \sigma_{\min(m,n)}$ be the singular values of an $m \times n$ matrix $\boldsymbol{A}$. Then the *nuclear norm* also called the *trace norm* is the matrix norm based on the function

$$f(\boldsymbol{A}) = \sum_i \sigma_i \text{ commonly denoted } \|\boldsymbol{A}\|_*.$$

Definition 6.8 (The Schatten Norms)
Let $\sigma_1, \ldots, \sigma_{\min(m,n)}$ be the singular values of an $m \times n$ matrix $\boldsymbol{A}$. Let $\mathbf{s}$ be the *vector* of singular values, ordered arbitrarily. Then the *Shatten p-norm* is the matrix norm based on the function

$$f(\boldsymbol{A}) = \|\mathbf{s}\|_p.$$

Definition 6.9 (The Ky-Fan Norms)
Let $\sigma_1, \ldots, \sigma_{\min(m,n)}$ be the singular values of an $m \times n$ matrix $\boldsymbol{A}$ where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,n)}$ by convention (that is, the elements are ordered in decreasing order in most conventions). Then the *Ky-Fan p-norm* is the matrix norm based on the function

$$f(\boldsymbol{A}) = \sum_{i=1}^{p} \sigma_i.$$

Note that both Shatten and Ky-Fan norms are *vector norms* applied to the vector of singular values $\mathbf{s}$. For Shatten norms, it is a p-norm. For Ky-Fan norms, it is the sum of the largest p elements. Indeed, any vector norm applied to the singular values of a matrix is a valid matrix norm.

6.3 ORTHOGONAL INVARIANCE

An important property of a norm is that it is orthogonally invariant. This property is a realization of two ideas:

- *norms measure lengths - orthogonal matrices generalize rotations*

When we rotate a vector, we simple change its orientation, but not its length. Consequently, we have the definition:

Definition 6.10 (orthogonally invariant)
Let $\mathbf{Q}$ be a square orthogonal matrix. Then a vector norm $f(\mathbf{x})$ is *orthogonally invariant* when

$$f(\mathbf{Qx}) = f(\mathbf{x}) \quad \text{or written as} \quad \|\mathbf{Qx}\| = \|\mathbf{x}\|.$$

Let $\boldsymbol{A}$ be an $m \times n$ matrix. Let $\boldsymbol{U}$ be a square $m \times m$ orthogonal matrix and let $\boldsymbol{V}$ be a square $n \times n$ orthogonal matrix. Then a matrix norm $f(\boldsymbol{A})$ is *orthogonally invariant* when

$$f(\boldsymbol{UAV}) = f(\boldsymbol{A}) \quad \text{or written as} \quad \|\boldsymbol{UAV}\| = \|\boldsymbol{A}\|.$$

6.4 THE SUBMULTIPLICATIVE PROPERTY

Note that operator-induced matrix norms satisfy the property that:

$$f(\boldsymbol{A}\mathbf{x}) \leq f(\boldsymbol{A})f(\mathbf{x})$$

which is handy for studying iterative algorithms! This property has the special name: *sub-multiplicative*.

Definition 6.11
A matrix-norm $f(\boldsymbol{A})$ is sub-multiplicative if:

$$f(\boldsymbol{AB}) \leq f(\boldsymbol{A})f(\boldsymbol{B}).$$

As you'll see on the homework, not all norms are sub-multiplicative. But we can always scale a norm to be sub-multiplicative.

6.5 MODULAR NORMS

These are a very recent idea. In many applications of optimization, machine learning, and artificial intelligence, we work with a collection of variables. In this collection of variables, some may represent *matrices* whereas other represent *vectors*. For instance, in a simple convolutional network (see)

EXERCISES

1. Let $\mathbf{x} \in \mathbb{C}^n$. Decompose $\mathbf{x}$ into the real and imaginary parts: $\mathbf{x} = \mathbf{y} + i\mathbf{z}$ where $\mathbf{y} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{R}^n$. Show that $\|\mathbf{x}\|_2 = \| \left[\begin{smallmatrix} \mathbf{y} \\ \mathbf{z} \end{smallmatrix} \right] \|_2$.
2. Let $\alpha \in \mathbb{R}^1$ be a scalar. Show that if $f(\alpha)$ is a norm on a scalar, then $f(\alpha) = \sigma|\alpha|$ for some σ. Hence, "vector norms" on scalars are just scaled absolute value functions.
3. Let $\boldsymbol{P}$ be a permutation matrix. So $\boldsymbol{P}\mathbf{x}$ reorders the elements of $\mathbf{x}$. Find a vector-norm function on length 2 vectors where $\|\mathbf{x}\| \neq \|\boldsymbol{P}\mathbf{x}\|$.
4. (This requires knowledge of the SVD.) Show that the Schatten and Ky-Fan norms are orthogonally invariant.
5. Consider the following function:

$$f(\boldsymbol{A}) = \max_{i,j} |A_{i,j}|.$$

(a) Show that f is a matrix norm. (Very easy!)

(b) Show that f does not satisfy the sub-multiplicative property.

(c) Show that there exists $\sigma > 0$ such that:

$$g(\boldsymbol{A}) = \sigma f(\boldsymbol{A})$$

is a sub-multiplicative matrix-norm.

(d) More generally, show that if $f(\boldsymbol{A})$ is any matrix norm that is not submultiplicative, then there exists a scalar σ such that $g(\boldsymbol{A}) = \sigma f(\boldsymbol{A})$ is submultiplicative. Consequently, there is nothing special.

6. Let $\|\boldsymbol{A}\|$ be a matrix norm and let $\mathbf{k} \neq 0$ be a real-valued vector. Consider the function:

$$f(\mathbf{x}) = \|\mathbf{x}\mathbf{k}^T\|.$$

(a) Show that f is a vector norm.

(b) Show that if $\|\boldsymbol{A}\|$ is a sub-multiplicative matrix norm, then the vector norm f is *consistent* with the matrix norm. That is:

$$f(\boldsymbol{A}\mathbf{x}) \leq \|\boldsymbol{A}\| f(\mathbf{x}).$$

7. The function

$$f(\mathbf{x}) = \text{ sum of two largest entries in } \mathbf{x} \text{ by absolute value}$$

is a vector norm. Use a computer to prepare a 3d plot of the unit-ball for this norm where $\mathbf{x} \in \mathbb{R}^3$.

THE FIELD OF VALUES 6

This chapter will be numbered after CS515 in 2023.

6.6 IMPORTANT CLASSES OF MATRICES

Definition 6.12 (Normal matrices)
A matrix is normal if $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^T$ (or conjugate transpose) or equivalently, if $\boldsymbol{A}$ has an orthonormal set of eigenvectors.

EXAMPLE 6.13 A circulant matrix is a non-symmetric and normal matrix. For example,

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{bmatrix} \text{ where } \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{A}^T\boldsymbol{A} = \begin{bmatrix} 14 & 11 & 11 \\ 11 & 14 & 11 \\ 11 & 11 & 14 \end{bmatrix}$$

◆

Definition 6.14 (Diagonalizable)
A matrix is diagonalizable if all of the eigenvalues are non-degenerate.

EXAMPLE 6.15 The following matrix is *non-normal* and *non-symmetric* but diagonalizable.

$$\boldsymbol{A} = \begin{bmatrix} 3 & 0 & 3 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Then the eigenpairs of $\boldsymbol{A}$ are

$$0, \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T \quad 2, \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T \quad 4, \begin{bmatrix} 9/10 & 0 & 1/10 \end{bmatrix}$$

where we leave the eigenvectors unnormalized for simplicity. ◆

EXAMPLE 6.16 The following matrix is both non-normal and non-diagonalizable

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

in fact, any triangular matrix with a constant diagonal is non-normal and non-diagonalizable. ◆

We have the following set of implications

Symmetric or Hermitian ⇒ Normal ⇒ Diagonalizable

EXERCISES

1. Let $\boldsymbol{A}$ be a symmetric matrix and let $\boldsymbol{D}^{-1}$ be a diagonal matrix. Show that any matrix $\boldsymbol{D}^{-1}\boldsymbol{A}$ is diagonalizable.
2. This example is from Trefthen and Embree's book on non-normal matrices.[4] Let

$$\boldsymbol{A} = \begin{bmatrix} 0 & 2 & 0 & \cdots & \\ 1/2 & 0 & 2 & 0 & \cdots \\ 0 & 1/2 & 0 & 2\ddots & \ddots \\ & & \ddots & \ddots & \ddots \end{bmatrix}$$

   Then $\boldsymbol{A}$ is diagonalizable.

[4] Add real citation

SIMPLE ITERATIVE ALGORITHMS

II

We are going to look at a number of algorithms for solving linear systems of equations and least squares problems. These are all going to be "simple" algorithms in that we are going to derive them by using a few simple ideas that result from studying the equations that define a linear system.

The algorithms we are going to study right now are all of the flavor:

$$\text{start} \to \text{improve} \to \text{improve} \to \cdots$$

or as I like to think of them

$$\text{guess} \to \text{check} \to \text{correct} \to \text{check} \to \text{correct} \to \cdots.$$

That is to say, these are going to be *iterative* algorithms. We will construct a sequence of vectors that *hopefully* converges to the solution of the linear system of equations of least squares problem.

The fundamental idea behind simple iterative algorithms is that the algorithm behaves *as a matrix raised to a power.*

Theorem 6.17 (Fundamental Theorem of Simple Iterative Methods) Let $\boldsymbol{M}$ be a matrix with spectral radius $\rho(\boldsymbol{M}) < 1$, then the sequence of matrices

$$\boldsymbol{M}^k \to 0 \text{ as } k \to \infty.$$

This simple result allows us to build algorithms that will enable us to solve all non-singular linear systems.

Add a section on the fundamental theorem of iterative methods that covers how if the vector b isn't in the space where $\boldsymbol{A}$ has an eigenvalue bigger than 1, the method will still converge.

Add notes on the value of α for richardson, improve this section overall and get more of the theory in it!

7 SIMPLE ITERATIVE METHODS

The goal in this next section is to build an algorithm to solve a linear system of equations. We begin with a brief review of what this means in order to get to our iterative algorithm.

Learning objectives

1. A recap of what it means to solve a linear system.
2. Why we only talk about solving full rank systems.
3. The Neumann series algorithm for solving some linear systems.
4. How to evaluate an approximate solution.

Note that this section could also come after discussing vector and matrix norms, as we will use those in our discussion of approximate solutions.

We assume that you might already be familiar with the Euclidean vector norm: $\|\mathbf{x}\| = \sqrt{\sum_i x_i^2}$ from past experiences.

7.1 REVIEW OF LINEAR SYSTEMS OF EQUATIONS

Let's start with some basic properties of linear systems of equations.[1] A linear system

$$\boldsymbol{A}\mathbf{x} = \mathbf{b}$$

represents a set of equations

$$\begin{aligned} A_{1,1}x_1 + A_{1,2}x_2 + \ldots + A_{1,n}x_n &= b_1 \\ A_{2,1}x_1 + A_{2,2}x_2 + \ldots + A_{2,n}x_n &= b_2 \\ &\vdots \\ A_{m,1}x_1 + A_{m,2}x_2 + \ldots + A_{m,n}x_n &= b_m. \end{aligned}$$

This is a relationship described by m equations and n unknowns. These come from an enormous diversity of scenarios as detailed in previous lectures and notes.

If there are fewer equations than unknowns $(m < n)$, then the system is called *underdetermined* and it may have 0, 1, or an infinite set of solutions. If $m = n$, the system is called *square* and the system can have 0, 1, or an infinite set of solutions. And if $m > n$, the system is called *overdetermined* and it can have 0, 1, or an infinite set of solutions.

The above expressions are all 0, 1, or an infinite number. As a small consideration, why can't we have *two* solutions but not an infinite number? This is a property of a linear set of equations that is part of what makes them special and *easy to solve*. Suppose we have two solutions $\mathbf{x}$ and $\mathbf{y}$

$$\boldsymbol{A}\mathbf{x} = \mathbf{b} \qquad \boldsymbol{A}\mathbf{y} = \mathbf{b} \qquad \mathbf{x} \neq \mathbf{y}.$$

Then *any combination of those solutions* is also a solution, such as

$$\boldsymbol{A}(\gamma\mathbf{x} + (1-\gamma)\mathbf{y}) = \gamma\boldsymbol{A}\mathbf{x} + (1-\gamma)\boldsymbol{A}\mathbf{y}) = \gamma\mathbf{b} + (1-\gamma)\mathbf{b} = \mathbf{b}$$

and we have this relationship for all γ. This is an infinite set of solutions.

How can we have zero solutions to an underdetermined system? This is because the above characterization did not prescribe anything about the *dependencies* among solutions. For instance, here are two equations

$$\begin{aligned} -x + y - z &= 2 \\ -x + y - z &= 3. \end{aligned}$$

[1] This section should be a review.

Note that these are the same equation with a different value. There is no solution. As a matrix $\boldsymbol{A}$, this scenario is a 2×3 matrix with rank 1.

EXAMPLE 7.1 Here is a fun case to consider. Let $\boldsymbol{A} = \mathbf{y}\mathbf{y}^T$. When does $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ have a solution? When does it have no solution? Describe a procedure to find the solution.

When does it have a solution? If $\boldsymbol{A} = \mathbf{y}\mathbf{y}^T$ then $\boldsymbol{A}\mathbf{x} = \mathbf{y}\mathbf{y}^T\mathbf{x} = \sigma\mathbf{y}$ for some scalar σ. So if $\mathbf{b} = \sigma\mathbf{y}$ for any scalar σ then the linear system is non-trivial and always has a solution (actually, an infinite number of solutions). If $\mathbf{b}$ has any other form, then there is not a solution.

A procedure to find a solution. Suppose we *know* $\boldsymbol{A} = \mathbf{y}\mathbf{y}^T$ for some unknown $\mathbf{y}$ and also, we *know* that $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ has a solution. Then by the above analysis, we *know* that $\mathbf{b} = \sigma\mathbf{y}$ for some σ and also that means $\mathbf{x} = \alpha\mathbf{b}$ for some α. So we compute $\mathbf{z} = \boldsymbol{A}\mathbf{b}$, which will give us a vector that we just have to scale to get the right answer. So pick any non-zero entry j in $\mathbf{z}$ and find $\sigma = b_j/z_j$. Then $\mathbf{x} = \sigma\mathbf{b}$ is a solution. ◆

For this reason, typically people have chosen to discuss equations in terms of *full rank* matrices. An $m \times n$ matrix is full rank if the rank is $\min(m, n)$. In this case, underdetermined problems $(m < n)$ always have an infinite number of solutions. Overdetermined problems $(m > n)$ have either 1 or 0 solutions. Square systems have only one *unique* solution.

Unfortunately, all this flexibility in terms of the number of solutions makes it hard to discuss algorithms. Consequently:

When we consider solving *linear systems, we always focus on the* square, full-rank *case.*

There are a large number of known ways to characterize when a square system of linear equations is full rank.

- rank($\boldsymbol{A}$) is n (or m since $m = n$)
- $\boldsymbol{A}$ is invertible
- the columns of $\boldsymbol{A}$ are linearly independent
- the rows of $\boldsymbol{A}$ are linearly independent
- the determinant of $\boldsymbol{A}$ is non-zero
- the eigenvalues of $\boldsymbol{A}$ are all non-zero
- the singular values of $\boldsymbol{A}$ are all non-zero.

7.2 THE INVERSE

When $\boldsymbol{A}$ is square and full rank, then there exists a matrix $\boldsymbol{Y}$ such that $\boldsymbol{A}\boldsymbol{Y} = \boldsymbol{I}$ and $\boldsymbol{Y}\boldsymbol{A} = \boldsymbol{I}$. This matrix $\boldsymbol{Y}$ is called the *inverse* and is usually written $\boldsymbol{A}^{-1}$.[2]

Given a linear system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, then we can multiply both sides by $\boldsymbol{Y}$ and get $\boldsymbol{Y}\boldsymbol{A}\mathbf{x} = \boldsymbol{Y}\mathbf{b}$ where $(\boldsymbol{Y}\boldsymbol{A}) = \boldsymbol{I}$, so we get $\mathbf{x} = \boldsymbol{Y}\mathbf{b}$ or $\mathbf{x} = \boldsymbol{A}^{-1}\mathbf{b}$. There are many, many interpretations of this statement.

[2] One of the key points of matrix computations as a field is that we should not compute $\boldsymbol{A}^{-1}$ to solve linear systems. There are better algorithms.

7.3 A FIRST METHOD

Most people learn the following result somewhere in the educational background for this class. Let x be a scalar, then

$$1 + x + x^2 + x^3 + \ldots = \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

when $|x| < 1$. That is, if $x^k \to 0$, then the infinite sequences converges to the value $1/(1-x)$, which we are going to write as $(1-x)^{-1}$.

It turns out that this same result holds for matrices as well, with a few additional conditions.[3]

Theorem 7.2 (The Matrix Geometric Series)

$$\text{If that } \boldsymbol{A}^k \to 0, \text{ then } \sum_{k=0}^{\infty} \boldsymbol{A}^k = (\boldsymbol{I} - \boldsymbol{A})^{-1}.$$

PROOF Our proof proceeds just by showing that a partial infinite sum becomes a better approximation to the inverse. Let $\boldsymbol{S}_\ell = \sum_{k=0}^{\ell} \boldsymbol{A}^k$ and consider

$$\boldsymbol{S}_\ell(\boldsymbol{I} - \boldsymbol{A}) = \sum_{k=0}^{\ell} \boldsymbol{A}^k = (\boldsymbol{I} - \boldsymbol{A}) + (\boldsymbol{A} - \boldsymbol{A}^2) + (\boldsymbol{A}^2 - \boldsymbol{A}^3) + \ldots = \boldsymbol{I} - \boldsymbol{A}^{\ell+1}.$$

Consequently,

$$\lim_{\ell \to \infty} \boldsymbol{S}_\ell(\boldsymbol{I} - \boldsymbol{A}) = \lim_{\ell \to \infty} \boldsymbol{I} - \boldsymbol{A}^{\ell+1} = \boldsymbol{I}$$

and we have finished the proof as this is an explicit form for the inverse.■

7.4 CHECKING A POSSIBLE SOLUTION

One great aspect about solving linear equations is that "guestimates" are easy to check.

Let $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ be the system we are trying to solve and let $\mathbf{y}$ be a potential solution. Then the following quantities all deal with how good $\mathbf{y}$ is:

$$\begin{aligned} \text{error} &= \mathbf{y} - \mathbf{x} \\ \text{error} &= \|\mathbf{y} - \mathbf{x}\| \\ \text{relative error} &= \|\mathbf{y} - \mathbf{x}\| / \|\mathbf{x}\| \end{aligned}$$

$$\begin{aligned} \text{residual} &= \mathbf{b} - \boldsymbol{A}\mathbf{y} \\ \text{residual} &= \boldsymbol{A}\mathbf{y} - \mathbf{b} \\ \text{residual} &= \|\boldsymbol{A}\mathbf{y} - \mathbf{b}\| \\ \text{relative residual} &= \|\boldsymbol{A}\mathbf{y} - \mathbf{b}\| / \|\mathbf{b}\| \end{aligned}$$

[3] This theorem is also called the Neumann series.

Note that there are terms that may refer to multiple quantities. These are often used interchangably where the definition is clear from context.

The *error* measures are the most useful quantities, however, they are not easily computable as they require *knowing* the solution $\mathbf{x}$. However, we can bound the error in terms of the residual.

Theorem 7.3
Let $\mathbf{y}$ be any vector and $\mathbf{x}$ be the solution of $A\mathbf{x} = \mathbf{b}$, then the *error* $\mathbf{e} = \mathbf{y} - \mathbf{x}$ and *residual* $\mathbf{r} = A\mathbf{y} - \mathbf{b}$ are related as follows:

$$A\mathbf{e} = \mathbf{r}.$$

PROOF By definition:

$$A\mathbf{e} = A\mathbf{y} - A\mathbf{x} = A\mathbf{y} - \mathbf{b}$$

because $A\mathbf{x} = \mathbf{b}$. ■

This results in the following bound.

Corollary 7.4
Using the notation from Theorem 7.3, let $\|\cdot\|$ be a sub-multiplicative norm.[4] Then

$$\|\mathbf{e}\| \leq \|A^{-1}\| \|\mathbf{r}\|.$$

The proof follows from $\mathbf{e} = A^{-1}\mathbf{r}$ and using $\|A^{-1}\mathbf{r}\| \leq \|A\|^{-1}\|\mathbf{r}\|$ for a sub-multiplicative norm.

What this means is that if we want the error to be small, then we want the residual to be small. And the residual is easy to compute!

7.5 OUR FIRST METHOD REVISITED

On reflection, there is a better way to introduce the algorithm involving the Neumann series of a matrix. This has to do with how we might check the solution of a linear system of equation.

Given some initial guess at a solution $\mathbf{x}_0$, then we are going to compute the residual: $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$. If we are close to a solution, this will be small. So let's just correct by the amount we need:

$$\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{r}_0.$$

Now, if we just repeatedly do this, then

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{r}_k = \mathbf{x}_k + \mathbf{b} - A\mathbf{x}_k = (I - A)\mathbf{x}_k + \mathbf{b}.$$

Quiz. Let $\mathbf{x}_0 = \mathbf{b}$. Show that $\mathbf{x}_k$ will converge to the solution $\mathbf{x}$ as $k \to \infty$. State conditions if necessary for this to converge.

Solution. By definition, $\mathbf{x}_1 = (I - A)\mathbf{b} + \mathbf{b}$ and $\mathbf{x}_2 = (I - A)^2\mathbf{b} + (I - A)\mathbf{b} + \mathbf{b}$. By induction, we have: $\mathbf{x}_k = \sum_{\ell=0}^{k}(I - A)^{\ell}\mathbf{b}$ and as $k \to \infty$, then $\mathbf{x}_k \to (I - H)^{-1}\mathbf{b}$ where $H = I - A$. But using that definition gives $(I - H)^{-1} = A^{-1}$. Hence this algorithm will converge if $\rho(I - A) < 1$ and it just corresponds to using the Neumann series itself.

[4] Not all matrix norms are sub-multiplicative, see the discussion of Matrix and Vector Norms.

7.5.1 *An example with our simple random walk between −4 and 6.*

Recall our linear system that modeled how long it took a random walk to exit through −4 and +6. This was the linear system of equations

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 1 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\{-4\}} \\ x_{\{-3\}} \\ x_{\{-2\}} \\ x_{\{-1\}} \\ x_{\{0\}} \\ x_{\{+1\}} \\ x_{\{+2\}} \\ x_{\{+3\}} \\ x_{\{+4\}} \\ x_{\{+5\}} \\ x_{\{+6\}} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

When we apply this method here starting from $\mathbf{x}^{(1)} = 0$ (the all zeros vector), we get a sequence of iterates $\mathbf{x}^{(k)}$ along with residuals $\mathbf{r}^{(k)}$. After a few hundred iterations, these have largely converged.

Value of iterate vector $\mathbf{x}^{(k)}$ *when* $k =$

1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90	100	200	300	400	500
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	1.5	2.0	2.4	2.8	3.1	3.4	3.6	3.9	5.9	7.1	7.9	8.3	8.6	8.8	8.8	8.9	8.9	9.0	9.0	9.0	9.0
0.0	1.0	2.0	2.8	3.5	4.1	4.8	5.3	5.8	6.3	10.0	12.0	14.0	15.0	15.0	16.0	16.0	16.0	16.0	16.0	16.0	16.0	16.0
0.0	1.0	2.0	3.0	3.9	4.8	5.5	6.3	7.0	7.7	13.0	16.0	18.0	19.0	20.0	20.0	21.0	21.0	21.0	21.0	21.0	21.0	21.0
0.0	1.0	2.0	3.0	4.0	4.9	5.9	6.7	7.6	8.4	15.0	18.0	21.0	22.0	23.0	23.0	24.0	24.0	24.0	24.0	24.0	24.0	24.0
0.0	1.0	2.0	3.0	4.0	5.0	5.9	6.9	7.7	8.6	15.0	19.0	21.0	23.0	24.0	24.0	25.0	25.0	25.0	25.0	25.0	25.0	25.0
0.0	1.0	2.0	3.0	4.0	4.9	5.9	6.7	7.6	8.4	15.0	18.0	21.0	22.0	23.0	23.0	24.0	24.0	24.0	24.0	24.0	24.0	24.0
0.0	1.0	2.0	3.0	3.9	4.8	5.5	6.3	7.0	7.7	13.0	16.0	18.0	19.0	20.0	20.0	21.0	21.0	21.0	21.0	21.0	21.0	21.0
0.0	1.0	2.0	2.8	3.5	4.1	4.8	5.3	5.8	6.3	10.0	12.0	14.0	15.0	15.0	16.0	16.0	16.0	16.0	16.0	16.0	16.0	16.0
0.0	1.0	1.5	2.0	2.4	2.8	3.1	3.4	3.6	3.9	5.9	7.1	7.9	8.3	8.6	8.8	8.8	8.9	8.9	9.0	9.0	9.0	9.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Value of residual vector $\mathbf{r}^{(k)}$ *when* $k =$

1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90	100	200	300	400	500
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$1e^{0}$	$5e^{-1}$	$5e^{-1}$	$4e^{-1}$	$4e^{-1}$	$3e^{-1}$	$3e^{-1}$	$3e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$9e^{-2}$	$5e^{-2}$	$3e^{-2}$	$2e^{-2}$	$1e^{-2}$	$7e^{-3}$	$4e^{-3}$	$3e^{-3}$	$2e^{-5}$	$1e^{-7}$	$8e^{-10}$	$5e^{-12}$
$1e^{0}$	$1e^{0}$	$8e^{-1}$	$8e^{-1}$	$6e^{-1}$	$6e^{-1}$	$5e^{-1}$	$5e^{-1}$	$5e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$7e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$5e^{-3}$	$4e^{-5}$	$2e^{-7}$	$2e^{-9}$	$1e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$7e^{-1}$	$7e^{-1}$	$6e^{-1}$	$4e^{-1}$	$2e^{-1}$	$1e^{-1}$	$9e^{-2}$	$5e^{-2}$	$3e^{-2}$	$2e^{-2}$	$1e^{-2}$	$7e^{-3}$	$5e^{-5}$	$3e^{-7}$	$2e^{-9}$	$1e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$6e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$6e^{-5}$	$4e^{-7}$	$2e^{-9}$	$2e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$6e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$6e^{-5}$	$4e^{-7}$	$2e^{-9}$	$2e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$6e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$6e^{-5}$	$4e^{-7}$	$2e^{-9}$	$2e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$7e^{-1}$	$7e^{-1}$	$6e^{-1}$	$4e^{-1}$	$2e^{-1}$	$1e^{-1}$	$9e^{-2}$	$5e^{-2}$	$3e^{-2}$	$2e^{-2}$	$1e^{-2}$	$7e^{-3}$	$5e^{-5}$	$3e^{-7}$	$2e^{-9}$	$1e^{-11}$
$1e^{0}$	$1e^{0}$	$8e^{-1}$	$8e^{-1}$	$6e^{-1}$	$6e^{-1}$	$5e^{-1}$	$5e^{-1}$	$5e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$7e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$5e^{-3}$	$4e^{-5}$	$2e^{-7}$	$2e^{-9}$	$1e^{-11}$
$1e^{0}$	$5e^{-1}$	$5e^{-1}$	$4e^{-1}$	$4e^{-1}$	$3e^{-1}$	$3e^{-1}$	$3e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$9e^{-2}$	$5e^{-2}$	$3e^{-2}$	$2e^{-2}$	$1e^{-2}$	$7e^{-3}$	$4e^{-3}$	$3e^{-3}$	$2e^{-5}$	$1e^{-7}$	$8e^{-10}$	$5e^{-12}$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

7.5.2 *An example with our least-squares problem*

Recall the quadratic fitting problem from the introduction lecture "What is a matrix" (lecture 1) This is reproduced at right.

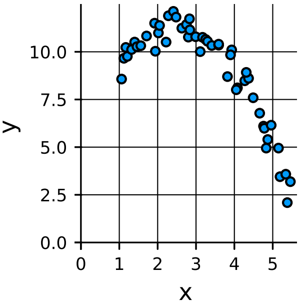


FIGURE 7.1 – We get a least squares problem to find a quadratic fit to this data $c_3x^2 + c_2x + c_1$. This gives a 3 × 3 linear system via the normal equations.

Consider the normal equations method of solving the least squares problem for the quadratic fit:

$$\begin{bmatrix} 50.0 & 162.63 & 616.468 \\ 162.63 & 616.468 & 2574.99 \\ 616.468 & 2574.99 & 11432.9 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 450.45 \\ 1329.6726 \\ 4549.258906 \end{bmatrix}$$

When we apply this method here starting from $\mathbf{c}^{(1)} = 0$ (the all zeros vector), we get a sequence of iterates $\mathbf{c}^{(k)}$ along with residuals $\mathbf{r}^{(k)}$. After less than one hundred iterations, this has produced 'NaN' on the computer – a hallmark that the algorithm cannot converge on the problem.

Value of solution vector $\mathbf{c}^{(k)} = [c_1\ c_2\ c_3]$ *when* k =

1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80
0	$4e^{2}$	$-3e^{6}$	$4e^{10}$	$-4e^{14}$	$5e^{18}$	$-6e^{22}$	$8e^{26}$	$-9e^{30}$	$1e^{35}$	$7e^{75}$	$5e^{116}$	$3e^{157}$	$2e^{198}$	$1e^{239}$	$8e^{279}$	NaN
0	$1e^{3}$	$-1e^{7}$	$2e^{11}$	$-2e^{15}$	$2e^{19}$	$-3e^{23}$	$3e^{27}$	$-4e^{31}$	$5e^{35}$	$3e^{76}$	$2e^{117}$	$1e^{158}$	$8e^{198}$	$5e^{239}$	$3e^{280}$	NaN
0	$4e^{3}$	$-6e^{7}$	$7e^{11}$	$-8e^{15}$	$1e^{20}$	$-1e^{24}$	$1e^{28}$	$-2e^{32}$	$2e^{36}$	$1e^{77}$	$8e^{117}$	$5e^{158}$	$4e^{199}$	$2e^{240}$	$2e^{281}$	NaN

The residuals show the same behavior and quickly grow to ∞.

Value of residual vector $\mathbf{r}^{(k)}$ *when* k =

1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80
$5e^{2}$	$-3e^{6}$	$4e^{10}$	$-4e^{14}$	$5e^{18}$	$-6e^{22}$	$8e^{26}$	$-9e^{30}$	$1e^{35}$	$-1e^{39}$	$-9e^{79}$	$-6e^{120}$	$-4e^{161}$	$-2e^{202}$	$-1e^{243}$	$-1e^{284}$	NaN
$1e^{3}$	$-1e^{7}$	$2e^{11}$	$-2e^{15}$	$2e^{19}$	$-3e^{23}$	$3e^{27}$	$-4e^{31}$	$5e^{35}$	$-6e^{39}$	$-4e^{80}$	$-2e^{121}$	$-1e^{162}$	$-1e^{203}$	$-6e^{243}$	$-4e^{284}$	NaN
$5e^{3}$	$-6e^{7}$	$7e^{11}$	$-8e^{15}$	$1e^{20}$	$-1e^{24}$	$1e^{28}$	$-2e^{32}$	$2e^{36}$	$-2e^{40}$	$-2e^{81}$	$-1e^{122}$	$-7e^{162}$	$-4e^{203}$	$-3e^{244}$	$-2e^{285}$	NaN

In this case, we can compute $\rho(\boldsymbol{I} - \boldsymbol{A}) = 12047.41494842964$, so the spectral radius is much larger than 1 and we would not expect the method to work.

What if this algorithm doesn't work? ================================

Suppose that $\rho(\boldsymbol{I} - \boldsymbol{A}) > 1$. Are we out of luck with using this method? Not entirely! Consider that we can *transform* the linear system into an equivalent system of equations:

$$\boldsymbol{A}\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \alpha\boldsymbol{A}\mathbf{x} = \alpha\mathbf{b}$$

Then the iteration is:

$$\mathbf{x}_{k+1} = (\boldsymbol{I} - \alpha\boldsymbol{A})\mathbf{x}_k + \alpha\mathbf{b} = \sum_{\ell=0}^{k} (\boldsymbol{I} - \alpha\boldsymbol{A})^{\ell}(\alpha\mathbf{b}) \to (\alpha\boldsymbol{A})^{-1}\alpha\mathbf{b}.$$

This method is called the Richardson method for solving a linear system of equations. It is credited to Lewis Fry Richardson. Among other things, Richardson decided to spend his time in the trenches during World War I dreaming up better uses for the people fighting the war. His solution was to have them forecast the weather and he came up with this method.

When will this method converge?

Based on our analysis of the Neumann series, this will converge if $\rho(\boldsymbol{I} - \alpha \boldsymbol{A}) < 1$. Let λ be an eigenvalue of $\boldsymbol{A}$. This means we need that $|1 - \alpha\lambda| < 1$ for all eigenvalues of $\boldsymbol{A}$.

This means we can always make this algorithm work for a symmetric positive definite matrix $\boldsymbol{A}$ because all of the eigenvalues are positive.

Move the remainder of this material to other sections

7.6 ANOTHER DERIVATION OF THE SAME ALGORITHM

7.6.1 *Notes 1*

Let's see *yet* another way to get at the same algorithm. This will involve some analysis of convex function.

Recall that a scalar quadratic function can be written:

$$f(x) = ax^2 + bx + c.$$

These look like bowls or lines (when $a = 0$).

Consider the problem

$$\underset{x}{\text{minimize}}\, ax^2 + bx + c$$

The solution is undefined is $a < 0$ (or just ∞). Otherwise, $x = -b/(2a)$ is the point that achieves the minimum. This can be found by looking for a point where the derivative is 0:

$$f'(x) = 2ax + b = 0 \Rightarrow x = -b/(2a).$$

A multivariate quadratic *looks* very similar.

7.6.2 *Notes 2*

Gradient Descent for $\boldsymbol{A}\mathbf{x} = \mathbf{b}$.

It turns out that for any positive definite matrix $\boldsymbol{A}$, that we can view it as the solution of an optimization problem

$$\underset{\mathbf{x}}{\text{minimize}} \quad \tfrac{1}{2}\mathbf{x}^T \boldsymbol{A}\mathbf{x} - \mathbf{x}^T \mathbf{b}.$$

This is because if $\boldsymbol{A}$ is positive semi-definite, then this problem is convex with a unique global minimizer. A convex function is just one that always lies below any line connecting two points. Formally, this is $f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \le \alpha f(\mathbf{x}) + (1-\alpha) f(\mathbf{y})$. A global minimizer is any point $\mathbf{x}^*$ where $f(\mathbf{x}^*) \le f(\mathbf{x})$ for any other point $\mathbf{x}$. Note that if $f(\mathbf{x})$ is convex and if we have two global minimizers, then any point on the line connecting them must be a minimizer by the property of convexity.

There is a stronger result to prove here too.

Theorem 7.5
Let $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \boldsymbol{A}\mathbf{x} - \mathbf{x}^T \mathbf{x}$. Then $f(\mathbf{x})$ is convex if $\boldsymbol{A}$ is symmetric positive definite.

PROOF From the definition

$$\begin{aligned} f(\alpha\mathbf{x}+(1-\alpha)\mathbf{y}) &= (\alpha\mathbf{x}+(1-\alpha)\mathbf{y})^T\boldsymbol{A}(\alpha\mathbf{x}+(1-\alpha)\mathbf{y}) - (\alpha\mathbf{x}+(1-\alpha)\mathbf{y})^T\mathbf{b} \\ &= \alpha(\alpha\mathbf{x}^T\boldsymbol{A}\mathbf{x}-\mathbf{x}^T\mathbf{b}+(1-\alpha)((1-\alpha)\mathbf{y}^T\boldsymbol{A}\mathbf{y}-\mathbf{y}^T\mathbf{y})+2\alpha(1-\alpha)\mathbf{x}^T\boldsymbol{A}\mathbf{y} = \dots \end{aligned}$$

■

EXERCISES

1. Let $\boldsymbol{A} = \boldsymbol{Y}\boldsymbol{Y}^T$ for some $n \times 2$ matrix $\boldsymbol{Y}$. Assume you have a routine to compute $\boldsymbol{X}^{-1}$ for a 2×2 matrix. Give a condition when $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ has at least one solution and give an algorithm to find it.
2. Show that using Richardson on the Poisson linear system will always converge if $\alpha < 1/4$.
3.

STEEPEST DESCENT & GRADIENT DESCENT 8

8.1 LINEAR SYSTEMS AND QUADRATIC FUNCTION MINIMIZATION

Learning objectives
1. Appreciate how linear systems are closely related to minimizing quadratic functions
2. Witness a computation of the gradient for a multivariate function in matrix algebra
3. See a characterization of a quadratic minimizer as the solution of a linear system
4. Generalize the algorithm to the steepest descent algorithm for solving a linear system

We are studying quadratic function minimization because this turns out to a good way to understand how to solve $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ for symmetric positive definite matrices $\boldsymbol{A}$. A full understanding of this will involve some analysis of convex functions. This is all *straightforward* for this case (if not simple), but it is an instance of a far more general theory. Some of the notes will make references to more general results that could be proved but are not relevant for the linear system case.

8.1.1 *Motivation from the scalar case*

————————————-

Recall that a scalar quadratic function can be written:

$$f(x) = ax^2 + bx + c.$$

These look like bowls or lines (when $a = 0$).

Consider the problem

$$\underset{x}{\text{minimize}}\, ax^2 + bx + c$$

The solution is undefined is $a < 0$ (or just ∞). Otherwise, $x = -b/(2a)$ is the point that achieves the minimum. This can be found by looking for a point where the derivative is 0:

$$f'(x) = 2ax + b = 0 \Rightarrow x = -b/(2a).$$

A multivariate quadratic *looks* very similar.

8.1.2 *The multivariate quadratic for* $\boldsymbol{A}\mathbf{x} = \mathbf{b}$

For $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, it turns out that for any positive definite matrix $\boldsymbol{A}$, that we can view it as the solution of an optimization problem

$$\underset{\mathbf{x}}{\text{minimize}} \quad \tfrac{1}{2}\mathbf{x}^T\boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}.$$

This is because if $\boldsymbol{A}$ is positive semi-definite, then this problem is convex with a unique global minimizer. A convex function is just one that always lies below any line connecting two points. Formally, this is $f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \le \alpha f(\mathbf{x}) + (1-\alpha)f(\mathbf{y})$. A global minimizer is any point $\mathbf{x}^*$ where $f(\mathbf{x}^*) \le f(\mathbf{x})$ for any other point $\mathbf{x}$. Note that if $f(\mathbf{x})$ is

There is a stronger result to prove here too.

convex and if we have two global minimizers, then any point on the line connecting them must be a minimizer by the property of convexity.

Theorem 8.1
Let $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$. Then $f(\mathbf{x})$ is convex if $\boldsymbol{A}$ is symmetric positive definite.

PROOF From the definition

$$
\begin{aligned}
f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) &= (\alpha\mathbf{x} + (1-\alpha)\mathbf{y})^T \boldsymbol{A}(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) - (\alpha\mathbf{x} + (1-\alpha)\mathbf{y})^T\mathbf{b} \\
&= \alpha(\alpha\mathbf{x}^T\boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b} + (1-\alpha)((1-\alpha)\mathbf{y}^T\boldsymbol{A}\mathbf{y} - \mathbf{y}^T\mathbf{y}) + 2\alpha(1-\alpha)\mathbf{x}^T\boldsymbol{A}\mathbf{y} \\
&= \alpha^2\mathbf{x}^T\boldsymbol{A}\mathbf{x} + (1-\alpha)^2\mathbf{y}^T\boldsymbol{A}\mathbf{y} + 2\alpha(1-\alpha)\mathbf{y}^T\boldsymbol{A}\mathbf{x} - \alpha\mathbf{x}^T\mathbf{b} - (1-\alpha)\mathbf{y}^T\mathbf{b}
\end{aligned}
$$

Our goal is to show that this is $\leq \alpha\mathbf{x}^T\boldsymbol{A}\mathbf{x} + (1-\alpha)\mathbf{y}^T\boldsymbol{A}\mathbf{y} - \alpha\mathbf{x}^T\mathbf{b} - (1-\alpha)\mathbf{y}^T\mathbf{b}$, and so the idea is to show that

$$\alpha^2\mathbf{x}^T\boldsymbol{A}\mathbf{x} + (1-\alpha)^2\mathbf{y}^T\boldsymbol{A}\mathbf{y} + 2\alpha(1-\alpha)\mathbf{y}^T\boldsymbol{A}\mathbf{x} - \alpha\mathbf{x}^T\boldsymbol{A}\mathbf{x} - (1-\alpha)\mathbf{y}^T\boldsymbol{A}\mathbf{y} \leq 0.$$

Note that we can simplify this to

$$(\alpha(\alpha-1))(\mathbf{x}^T\boldsymbol{A}\mathbf{x} + \mathbf{y}^T\boldsymbol{A}\mathbf{y} - 2\mathbf{x}^T\boldsymbol{A}\mathbf{y})$$

where we have $(\alpha(\alpha-1)) \leq 0$ and $(\mathbf{x}^T\boldsymbol{A}\mathbf{x}+\mathbf{y}^T\boldsymbol{A}\mathbf{y}-2\mathbf{x}^T\boldsymbol{A}\mathbf{y}) = (\mathbf{x}-\mathbf{y})^T\boldsymbol{A}(\mathbf{x}-\mathbf{y}) \geq 0$. Hence, the entire expression is ≤ 0, and we are done! ■

8.1.3 *The gradient*

Last time we proved that $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$ was a convex function. Let's show that the gradient of $f(\mathbf{x})$ is really the vector $\boldsymbol{A}\mathbf{x} - \mathbf{b}$.

EXAMPLE 8.2 Consider the function $f(\mathbf{x}) = \frac{1}{2}\begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} 3 & -1 \\ -1 & 4 \end{bmatrix} - \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} 7 \\ -6 \end{bmatrix} = 3/2x^2 + 2y^2 - xy - 7x + 6y$. Then the gradient is the vector

$$\begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} = \begin{bmatrix} 3x-y-7 \\ 4y-x+6 \end{bmatrix} = \begin{bmatrix} 3 & -1 \\ -1 & 4 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} 7 \\ -6 \end{bmatrix}.$$ ◆

More generally,

$$f(x_1, \ldots, x_n) = 1/2\sum_{ij} A_{ij}x_ix_j - \sum_i x_ib_i$$

We like thining of this in terms of the following table:

$A_{11}x_1^2$	$A_{12}x_1x_2$	$\cdots$	$A_{1n}x_1x_n$
$A_{21}x_2x_1$	$A_{22}x_2^2$	$\cdots$	$\vdots$
$\vdots$	$\ddots$	$\ddots$	$\vdots$
$A_{n1}x_nx_1\cdots$		$\cdots$	$A_{nn}x_n^2$

Now we have terms involving x_i in the ith row and ith column.

$$\partial f/\partial x_i = 1/2\sum_{j\neq i} A_{ij}x_i + A_{ii}x_i + 1/2\sum_{j\neq i} A_{ji}x_i - b_i = \text{ith row of } \boldsymbol{A}^T\mathbf{x} - b_i$$

The minimizer

The minimizer of a function is any point that is the lowest in some neighborhood. Formally, a point $\mathbf{x}^*$ is a local minimizer if $f(\mathbf{x}^*) \le f(\mathbf{x})$ for all $\mathbf{x}$ where $\|\mathbf{x} - \mathbf{x}^*\| \le \varepsilon$ for some positive value of ε. This just means that this is the lowest point in a neighborhood around the current point. The global minimizer $\mathbf{x}^*$ of a function is a point which is lower than everywhere else: $f(\mathbf{x}^*) \le f(\mathbf{x})$ for all $\mathbf{x}$.[1]

Convex functions are awesome because any local minimizer is a global minimizer!

This is easy to prove for continuous functions like the $f(\mathbf{x})$ that solves linear systems. Consider a point $\mathbf{x}$ and $\mathbf{y}$ where $\mathbf{x}$ is a local minizer and $\mathbf{y}$ is a global minimizer. Then along the line $\alpha\mathbf{x} + (1-\alpha)\mathbf{y}$ we must have that the function is bounded below by $\alpha f(\mathbf{x}) + (1-\alpha) f(\mathbf{y})$. Because $\mathbf{x}$ isn't a global min, we know that $f(\mathbf{y}) < f(\mathbf{x})$. Hence, that we *must* reduce the value of the function for all positive α compared with $f(\mathbf{x})$. This means that $f(\mathbf{x})$ couldn't have been a local minimizer. Hence, any local minimizer is a global minimizer of a continuous convex function.

Characterizing the minimizer

Any point where the gradient is zero is a global minimizer for a continuous convex function.

This is true generally, but it's super easy to show for our function for linear systems.

Theorem 8.3
[2] Let $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$ where $\boldsymbol{A}$ is symmetric, positive definite. Then the vector of partial derivatives is $\mathbf{g}(\mathbf{x}) = \boldsymbol{A}\mathbf{x} - \mathbf{b}$. Let $\mathbf{y}$ be a point where $\boldsymbol{A}\mathbf{x} - \mathbf{b} = 0$. Then $f(\mathbf{x}) \ge f(\mathbf{y})$.

PROOF Let $\mathbf{x} = \mathbf{y} + \alpha\mathbf{z}$. Then:

$$f(\mathbf{x}) = \tfrac{1}{2}(\mathbf{y}+\alpha\mathbf{z})^T \boldsymbol{A}(\mathbf{y}+\alpha\mathbf{z}) - (\mathbf{y}+\alpha\mathbf{z})^T\mathbf{b} = \tfrac{1}{2}\mathbf{y}^T\boldsymbol{A}\mathbf{y} + \tfrac{1}{2}\alpha^2\mathbf{z}^T\boldsymbol{A}\mathbf{z} + \alpha\mathbf{z}^T\boldsymbol{A}\mathbf{y} - \mathbf{y}^T\mathbf{b} - \alpha\mathbf{z}^T\mathbf{b}.$$

Now, recall that $\boldsymbol{A}\mathbf{y} = \mathbf{b}$ because the gradient is zero. Then we have:

$$f(\mathbf{x}) = \tfrac{1}{2}\mathbf{y}^T\boldsymbol{A}\mathbf{y} + \tfrac{1}{2}\alpha^2\mathbf{z}^T\boldsymbol{A}\mathbf{z} + \alpha\mathbf{z}^T\mathbf{b} - \mathbf{y}^T\mathbf{b} - \alpha\mathbf{z}^T\mathbf{b} = f(\mathbf{y}) + \tfrac{1}{2}\alpha^2\mathbf{z}^T\boldsymbol{A}\mathbf{z} \ge f(\mathbf{y}).$$

■

Finding the minimizer

If the gradient is not zero, then we can always reduce the function by moving a sufficiently small distance along the negative gradient.

In general, this is just an application of Taylor's theorem for multivariate function, but we can again proof this easily for us, and get a cool result along the way!

[1] For functions that aren't defined everywhere, this would be restricted to whereever the function is defined.

[2] This theorem generalizes to any function with a positive definite Hessian, but that's for an optimization class.

[3] For the moment, we'll let $\mathbf{g} = \mathbf{g}(\mathbf{x}) = \boldsymbol{A}\mathbf{x} - \mathbf{b}$ for a fixed $\mathbf{x}$.

Suppose $\mathbf{g}(\mathbf{x}) = \boldsymbol{A}\mathbf{x} - \mathbf{b} \neq 0$.[3] Then consider

$$\begin{aligned} f(\mathbf{x} - \alpha\mathbf{g}) &= \tfrac{1}{2}\mathbf{x}^T\boldsymbol{A}\mathbf{x} + \tfrac{1}{2}\alpha^2\mathbf{g}^T\boldsymbol{A}\mathbf{g} - \alpha\mathbf{g}^T\boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b} + \alpha\mathbf{g}^T\mathbf{b} \\ &= f(\mathbf{x}) + \tfrac{1}{2}\alpha^2\mathbf{g}^T\boldsymbol{A}\mathbf{g} - \alpha\mathbf{g}^T\boldsymbol{A}\mathbf{x} + \alpha\mathbf{g}^T\mathbf{b} \\ &= f(\mathbf{x}) - \alpha(\mathbf{g}^T\mathbf{g} - \tfrac{\alpha}{2}\mathbf{g}^T\boldsymbol{A}\mathbf{g}). \end{aligned}$$

So if this result is going to be true, we need $(\mathbf{g}^T\mathbf{g} - \frac{\alpha}{2}\mathbf{g}^T\boldsymbol{A}\mathbf{g}) \geq 0$ for α small enough. This corresponds with the condition that $\mathbf{g}^T\boldsymbol{A}\mathbf{g}/\mathbf{g}^T\mathbf{g} \leq 2/\alpha$. Let

$$\rho = \begin{array}{ll} \text{maximize} & \frac{\mathbf{x}^T\boldsymbol{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \\ \text{subject to} & \mathbf{x} \neq 0 \end{array} \qquad \text{then} \qquad \rho \geq \frac{\mathbf{g}^T\boldsymbol{A}\mathbf{g}}{\mathbf{g}^T\mathbf{g}} \text{ for any vector } \mathbf{g}.$$

This gives us an upper bound on the left hand side. Thus, we need $\rho \leq 2/\alpha$ or $\alpha \leq 2/\rho$ and we have descent,

$$f(\mathbf{x} - \alpha\mathbf{g}) = f(\mathbf{x}) - \alpha\,\underbrace{(\alpha/2\mathbf{g}^T\boldsymbol{A}\mathbf{g} - \alpha\mathbf{g}^T\mathbf{g})}_{\geq 0} \leq f(\mathbf{x}).$$

Note this is exactly the same bound we got out of the Richardson method too!

8.2 THE STEEPEST DESCENT ALGORITHM FOR SOLVING LINEAR SYSTEMS

We now need to turn these insights into an algorithm for solving a linear system of equations. The idea in steepest descent is that we use the insight from the last section: we are trying to minimize $f(\mathbf{x})$ and we can make $f(\mathbf{x})$ smaller by taking a step along the gradient $g(\mathbf{x})$.

8.2.1 *From Richardson to Steepest Descent*

Steepest descent on $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$ is just a generalization of Richardson's iteration:

$$\begin{aligned} \text{Richardson} \quad \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha\,\underbrace{(\mathbf{b} - \boldsymbol{A}\mathbf{x}^{(k)})}_{\text{residual}} \\ \text{Steepest Descent} \quad \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \alpha\,\underbrace{\mathbf{g}(\mathbf{x})}_{\text{gradient}} = \mathbf{x}^{(k)} - \alpha(\boldsymbol{A}\mathbf{x} - \mathbf{b}). \end{aligned}$$

This means that if $0 < \alpha < 2/\rho$ then the steepest descent method will converge.

8.2.2 *Picking a better value of α*

The idea with the steepest descent method is that we can pick α at each step and use $f(\mathbf{x})$ to inform this choice. This method arose from a completely different place from Richardson's method for solving $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ (which was based on the Neumann series).

Definition 8.4 (Steepest Descent Algorithm)
Let $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ be a symmetric, positive definite linear system of equations.

Finish this section...

8.2.3 *A coordinate-wise strategy.*

EXERCISES

1. The algorithms for coordinate and gradient descent are for solving $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ on a symmetric positive definite matrix $\boldsymbol{A}$ correspond to trying to minimize
$$f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T \boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$$
by, at each step, exactly minizing this function for a single coordinate (coordinate descent) or by exactly this function in the direction of the gradient (gradient descent). Let $\mathbf{x}_k$ represent the approximate solution after k steps. Let $\mathbf{g}_k$ represent the gradient $\boldsymbol{A}\mathbf{x}_k - \mathbf{b}$ at the kth step. Then the algorithms can be written:
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \frac{[\mathbf{g}_k]_i}{A_{i,i}}\mathbf{e}_i \qquad \mathbf{x}_{k+1} = \mathbf{x}_k - \frac{\mathbf{g}_k^T\mathbf{g}_k}{\mathbf{g}_k^T\boldsymbol{A}\mathbf{g}_k}\mathbf{g}_k$$
respectively.
Suppose that we wish to generalize coordinate descent to *two* coordinates. Then the update can be written:
$$\mathbf{x}_{k+1} = \mathbf{x}_k + \boldsymbol{E}_{ij}\mathbf{c}_k$$
where $\boldsymbol{E}_{ij} = \begin{bmatrix}\mathbf{e}_i & \mathbf{e}_j\end{bmatrix}$ and $\mathbf{c}_k = \begin{bmatrix} c_i \\ c_j \end{bmatrix}$. That is, we have to choose two coefficients: c_i, c_j in order to figure out the next step. Like coordinate descent, we are going to pick these to exactly minimize $f(\mathbf{x}_{k+1})$ as a function of c_i, c_j.
Give the resulting iterative algorithm.
2. (I'm not sure if this is true, and it could be difficult.). Let $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ be a diagonally dominant M matrix, but where $\boldsymbol{A}$ is not symmetric. This means that $\boldsymbol{A}^{-1} \geq 0$. Suppose also that $\mathbf{b} \geq 0$. Develop an algorithm akin to steepest descent for this problem. Ideas include looking at functions like $f(x) = \mathbf{e}^T\boldsymbol{A}\mathbf{x} - \mathbf{e}^T\mathbf{b}$

9 SIMULTANEOUS & SEQUENTIAL VARIABLE UPDATES (AKA JACOBI & GAUSS-SEIDEL)

Learning objectives
1. Viewing a linear system as a set of equations where we can solve for any variable given any equation it participates in.
2. Realize that Gauss-Seidel is just a "bug" in Jacobi

Now let's see another set of methods that can apply to solving $\boldsymbol{A}\mathbf{x} = \mathbf{b}$.

These, again, follow from different perspectives on what these equations mean. Consider that a system of linear equations represents a simultaneous solution of n individual equations

$$\boldsymbol{A}\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \begin{pmatrix} \mathbf{a}_1^T\mathbf{x} = b_1 \\ \mathbf{a}_2^T\mathbf{x} = b_2 \\ \cdots \\ \mathbf{a}_n^T\mathbf{x} = b_n \end{pmatrix} \text{ where } \boldsymbol{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \cdots \\ \mathbf{a}_n^T \end{bmatrix}$$

is the matrix with a vector for each row. We can use each of these equations to *solve* for any particular variable given the others. For example, given a possible solution $\mathbf{x}$, then

$$\mathbf{a}_1^T\mathbf{x} = b_1 \Leftrightarrow \sum_j A_{1j}x_j = b_1$$

and if $A_{1k} \neq 0$, then we must have

$$x_k = \frac{1}{A_{1k}}(b_1 - \sum_{j \neq k} A_{1j}x_j)$$

at a solution.[1]

Then one strategy to *improve* an inaccurate solution to $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ is to pick out a set of entries that can all be updated simultaneously and to do so.

9.1 A SIMPLE TWO VARIABLE EXAMPLE

Let's see an example. Suppose that

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ -2 & 1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

and we have a current guess $\mathbf{x} = \begin{bmatrix} 1 \\ 1/2 \end{bmatrix}$. Then for row 1, we have our choice of index k to update, 1 or 2. For the sake of example, let's use row 1 to update x_2 and then we'll use row 2 to update x_1. The update looks like:

$$\begin{aligned} x_1^{\text{new}} &= \frac{1}{-2}(1 - x_2^{\text{old}}) \\ x_2^{\text{new}} &= \frac{1}{2}(2 - x_1^{\text{old}}) \end{aligned}$$

[1] There must always exist such a k otherwise the system is singular!

For the guess $\mathbf{x} = \begin{bmatrix} 1 \\ 1/2 \end{bmatrix}$

$$\mathbf{x}^{\text{new}} = \begin{bmatrix} -1/4 \\ 1/2 \end{bmatrix}.$$

If we repeatedly use this formula, then we converge to the solution $\mathbf{x} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$ after a few iterations.

9.2 THE GENERAL EXAMPLE AND THE JACOBI ITERATION

If we think about running this on a general system, then we'll need to figure out which equation we use to update which variable. This is tricky, however, because the update to x_2 based on row 1 used an entry A_{12} that was non-zero. In general, this means that we cannot update x_2 from any of the other rows from 2 to n. In an ideal world (like the example above), we'd like to update $x_1, \ldots, x_n$ all at once.[2] That means that we need a distinct row i for each j such that $A_{ij} \neq 0$. We can encode this map in a matrix $\boldsymbol{M}$. Let $\boldsymbol{M}$ be an $n \times n$ matrix where:

$$M(i,j) = \begin{cases} 1 & \text{row } j \text{ is used to update } x_i \\ 0 & \text{otherwise.} \end{cases}$$

Then note that we must have exactly 1 entry in each row and column of $\boldsymbol{M}$.[3] We can use any matrix $\boldsymbol{M}$ where

$$M_{i,j} \neq 0 \text{ if and only if } A_{i,j} \neq 0.$$

In general these are not-so-easy to find. We'll return to this point in a moment. Let $M_i = j$ for convinience of notation. The iteration is thus:

$$x_1^{\text{new}} = \frac{1}{A_{M_1,1}}(b_{M_1} - \sum_{j \neq 1} A_{M_1,j} x_j)$$
$$\vdots$$
$$x_i^{\text{new}} = \frac{1}{A_{M_i,i}}(b_{M_i} - \sum_{j \neq i} A_{M_i,j} x_j)$$
$$\vdots$$
$$x_n^{\text{new}} = \frac{1}{A_{M_n,n}}(b_{M_n} - \sum_{j \neq n} A_{M_i,j} x_j)$$

We can state this using a set of matrices with some slight additional notation. Let $\boldsymbol{D}_M$ be the *diagonal* matrix where $[\boldsymbol{D}_M]i, i = A_{M_i,i}$. The idea with $\boldsymbol{D}_M$ is that we can write

$$\mathbf{x}^{\text{new}} = \boldsymbol{D}_M^{-1} \text{some vector}$$

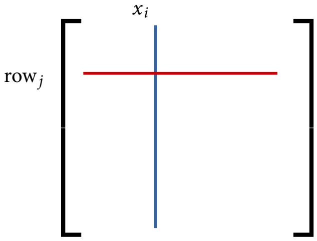


FIGURE 9.1 – A general setup for the equations described by rows of a system. We need $A_{j,i} \neq 0$ to update x_i based on equation j. We also cannot use equation j to update any other variables, nor can we use other equations to update x_i. The map matrix $\boldsymbol{M}$ encodes the set of updates.

[2] There are some really interesting and useful extensions that relax this assumption.

[3] Technically, and looking ahead just a few paragraphs, the matrix $\boldsymbol{M}$ is a permutation matrix, we will use this insight in a moment!

where entries of that vector correspond to $b_{M_i} - \sum_{j\neq i} A_{M_i,j} x_j)$. Consequently, let $\mathbf{b}_M$ be the vector $[\mathbf{b}_M]_i = b_{M_i}$. Also, let $\boldsymbol{N}_M$ be the matrix with entries:

$$[\boldsymbol{N}_M]_{i,j} = \begin{cases} 0 & i = j \\ A_{M_i,j} & i \neq j. \end{cases}$$

$$\mathbf{x}^{\text{new}} = \boldsymbol{D}_M^{-1}(\mathbf{b}_M - \boldsymbol{N}_M \mathbf{x}^{\text{old}}.$$

This iteration is called the Jacobi *method for solving a linear system of equations, although I think the name simultaneous variable updates or simultaneous variable relaxation communicates the idea better.*

9.3 IMPLEMENTATIONS OF JACOBI

```
"""
    jacobi_iteration_map_!(y,A,x,M) sets y to be the next Jacobi
    iteration from x with map M
"""
function jacobi_iteration(y,A,x,M=1:length(x))
  for i=1:length(x)
    y[i] -= (b[M[i] - A[M[i],:]'*x - A[M[i],i]*x[i]) / A[M[i],i]
  end
end
```

9.4 A 3X3 EXAMPLE WITH A PERMUTATION INSTEAD OF THE MAP

Most derivations of the Jacobi iteration assume that $\boldsymbol{D}$ is formed from the non-zero diagonal of the linear system of equations, but there is no such restriction in the derivation of the method. This is simply a notational convenience. All we need is a permutation matrix $\boldsymbol{P}$ such that $\boldsymbol{P} \odot \boldsymbol{A}$ is non-singular to build the matrix $\boldsymbol{M}$ for the above iteration to work.[4]

Consider the following linear system

$$\begin{bmatrix} 1 & 0 & -3 \\ 0 & -3 & 1 \\ -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \tag{9.1}$$

Then if we run the Jacobi update with

$$M_1 = 3 \qquad M_2 = 2 \qquad M_3 = 1 \text{ and } \mathbf{x}^{(0)} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T$$

We have

$$x_1^{\text{new}} = \frac{1}{A_{3,1} = -3}(1 - (A_{3,2} = 0) \cdot x_2 - (A_{3,3} = 1) \cdot x_3) = 0$$

$$x_2^{\text{new}} = \frac{1}{(A_{2,2} = -3)}(1 - (A_{2,1} = 0) \cdot x_1 - (A_{2,3} = 1) \cdot x_3) = 0$$

[4] Here $\odot$ is the element-wise, or Hadamard, products.

$$x_3^{\text{new}} = \frac{1}{(A_{1,3} = -3)}(1 - (A_{1,1} = 1) \cdot x_1 - (A_{1,2} = 0) \cdot x_2) = 0.$$

If we write this as a matrix update, we have:

$$\mathbf{x}^{(k+1)} = \begin{bmatrix} -3 & & \\ & -3 & \\ & & -3 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} -3 & & \\ & -3 & \\ & & -3 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \mathbf{x}^{(k)}.$$

To avoid the details with $\boldsymbol{M}$, let us simply *permute* the three rows of $\boldsymbol{A}$ so that row 3 comes first, then row 2, then row 1. Note that we simply reordered the rows, not the variables.

$$\begin{bmatrix} -3 & 0 & 1 \\ 0 & -3 & 1 \\ 1 & 0 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

In this case, we can simply split the matrix into it's two pieces:

$$\boldsymbol{D} = \text{ diagonal } = \begin{bmatrix} -3 & & \\ & -3 & \\ & & -3 \end{bmatrix} \qquad \boldsymbol{N} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

With this setup, we can easily write the iteration

$$\mathbf{x}^{(k+1)} = \boldsymbol{D}^{-1}\mathbf{b} - \boldsymbol{N}\mathbf{x}^{(k)}.$$

9.5 JACOBI WITH A PERMUTATION MATRIX

Let us derive the same update as above, using the permutation to show how it generalizes the previous algorithm. That's because the matrix $\boldsymbol{M}$ really is a permutation! A permutation matrix is just a way to reorder the rows or columns of a matrix. It reorders the rows if we multiply on the left.

For a permutation matrix $\boldsymbol{P}$, we have $P_{i,j} = 1$ if $\mathbf{y} = \boldsymbol{P}\mathbf{x}$ has $y_i = x_j$. That is, $P_{i,j}$ if i in the output was j in the input. So we can write:

$$\mathbf{b}_M = \boldsymbol{P}\mathbf{b} \text{ where } P_{i,j} = \begin{cases} 1 & j = M_i \\ 0 & \text{otherwise.} \end{cases}$$

We can also apply the same permutation to $\boldsymbol{A}$ as in

$$\boldsymbol{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix} \text{ then } \boldsymbol{PA} = \begin{bmatrix} \mathbf{a}_{M_1}^T \\ \mathbf{a}_{M_2}^T \\ \vdots \\ \mathbf{a}_{M_n}^T \end{bmatrix}.$$

This corresponds with solving a linear system

$$\boldsymbol{PA}\mathbf{x} = \boldsymbol{P}\mathbf{x}$$

or really, just re-ordering the equations we were given. Now note that

$$\boldsymbol{D}_M = \text{diagonal elements of } \boldsymbol{PA}\text{in a diagonal matrix.}$$

Then also,

$$\boldsymbol{N}_M = \boldsymbol{P}\boldsymbol{A} - \boldsymbol{D}_M.$$

This is why most textbooks do not describe the setup with using an equation to solve for an unknown, it's entirely equivalent to the following setup that is much closer to a typical description of Jacobi

1. Pick a permutation matrix $\boldsymbol{P}$ such that $\boldsymbol{P}\boldsymbol{A}$ has non-zero diagonal elements. 2. Let $\boldsymbol{D}$ be the diagonal elements of $\boldsymbol{P}\boldsymbol{A}$. 3. Let $\boldsymbol{N}$ be the matrix $\boldsymbol{P}\boldsymbol{A} - \boldsymbol{D}$

Then the Jacobi method implements the iteration:

$$\mathbf{x}^{(k+1)} = \boldsymbol{D}^{-1}(\boldsymbol{P}\mathbf{b} - \boldsymbol{N}\mathbf{x}^{(k)}).$$

This makes it much simplier to do the analysis below when we are interested in issues of convergence: permute the matrix, then look at the diagonal entries and t

EXAMPLE 9.1 Show that the linear system (9.1) will not converge using Jacobi with the *standard* permutation $\boldsymbol{P} = \boldsymbol{I}$. ◆

EXAMPLE 9.2 Note that the 3x3 least squares problem for our quadratic fitting example will not converge with Jacobi for *any* iteration. ◆

Interesting tangent

Of course, this begs the question, why do we need a *single* permutation matrix $\boldsymbol{P}$? Why can't we get away with using a sequence of iterations where we just ensure that each element is updated every so-often. I'm almost sure this has been studied, but don't know the reference off the top of my head. Or even a random pair at each step.

9.6 THE CONVERGENCE OF THE JACOBI METHOD

It's easy to determine the convergence of the Jacobi matrix with our knowledge of the spectral-radius of a matrix. Let's look at the error in the kth-step of the method:

$$\mathbf{x}^{(k+1)} - \mathbf{x} = \boldsymbol{D}_M^{-1}(\mathbf{b} - \boldsymbol{N}_M\mathbf{x}^{(k)}) - \mathbf{x}.$$

But note that we designed this so that $\mathbf{x}$ is a fixed point of the update, so $\mathbf{x} = \boldsymbol{D}_M^{-1}(\mathbf{b} - \boldsymbol{N}_M\mathbf{x})$ as well. This means that

$$\mathbf{x}^{(k+1)} - \mathbf{x} = \boldsymbol{D}_M^{-1}(\mathbf{b} - \boldsymbol{N}_M\mathbf{x}^{(k)} - \mathbf{b} + \boldsymbol{N}_M\mathbf{x}) = -\boldsymbol{D}_M^{-1}\boldsymbol{N}_M(\mathbf{x}^{(k)} - \mathbf{x}).$$

Consequently,

$$\mathbf{x}^{(k+1)} = (-\boldsymbol{D}_M^{-1}\boldsymbol{N}_M)^{k+1}(\mathbf{x}^{(0)} - \mathbf{x}).$$

This converges, for all starting points $\mathbf{x}^{(0)}$ if and only if $\rho(-\boldsymbol{D}_M^{-1}\boldsymbol{N}_M) < 1$.

9.7 A MISTAKE IN AN IMPLEMENTATION OF THE JACOBI METHOD

If we are solving large linear systems of equations, then we may have vectors with *billions* of entries! This means that storing another vector $\mathbf{x}^{\text{new}}$ may be expensive itself. In this case, I hope you can agree that it may occur to someone to try and save memory as follows:

> just update the solution to $\mathbf{x}$ with only a single set of memory!

For instance, consider the following implementation of the Jacobi iteration

```julia
"""
    jacobi_iteration!(y,A,x,M) updates x "like" Jacobi, but in-place, with map M
"""
function jacobi_iteration!(A,b,x,M=1:length(x))
  for i=1:length(x)
    x[i] -= (b[M[i]] - A[M[i],:]'*x - A[M[i],i]*x[i]) / A[M[i],i]
  end
end
```

This is wrong if the objective is to implement the Jacobi iteration. However, it turns out that this idea gives rise to a method called the Gauss-Seidel method.

$$
\begin{aligned}
x_1 &= \frac{1}{A_{M_1,1}}\Big(b_{M_1} - \sum_{j \neq M_1} A_{M_1,j} x_j\Big) \\
&\dots \\
x_i^{\text{new}} &= \frac{1}{A_{M_i,1}}\Big(b_{M_i} - \sum_{j \neq M_i} A_{M_i,j} x_j\Big) \\
&\dots
\end{aligned}
$$

9.8 ANALYZING GAUSS-SEIDEL

Test

9.9 THE GAUSS-SEIDEL AND STEEPEST DESCENT METHOD

Note, you can show that Gauss-Seidel converges on any symmetric positive definite matrix using the matrix $\boldsymbol{M} = \boldsymbol{I}$.

We can derive the Gauss-Seidel method as a mistake in Jacobi. Let's now consider what happens on a symmetric matrix $\boldsymbol{A}$ with unit diagonals. We have:

$$
\begin{aligned}
x_1^{\text{new}} &= \Big(b_1 - \sum_{j>1} A_{1,j} x_j^{\text{old}}\Big) \\
&\dots \\
x_i^{\text{new}} &= \Big(b_i - \sum_{j<i} x_j^{\text{new}} - \sum_{j>i} A_{1,j} x_j^{\text{old}}\Big)
\end{aligned}
$$

Recall the iteration for coordinate descent:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \gamma_k \mathbf{e}_i \text{ where } \gamma_k = -[\boldsymbol{A}\mathbf{x} - \mathbf{b}]_i / A_{i,i}$$

Written in terms of our problem

$$x_i^{\text{new}} = x_i - \sum_{j=1} A_{ij}\mathbf{x}^{\text{old}} + b_i = b_i - \sum_{j \neq i} A_{i,j} x_j^{\text{old}}.$$

This shows that if we update the ith variable, then we are doing a closely related update to Gauss-Seidel. To see that they are the same, remember how we arrived at Gauss-Seidel, we simply did the Jacobi update but *forgot* to allocate new memory. This means that, in the program, we have:

$$x_i^{\text{new}} = (b_i - \sum_{j<i} x_j^{\text{cur}} - \sum_{j>i} A_{1,j} x_j^{\text{cur}})$$

And now we can see that, expressed this way, Gauss-Seidel update is exactly the same as steepest descent.

EXERCISES

1. One of the common classes of matrices are called diagonally dominant. They have, or can be permuted to have, entries on on the diagonal such that:

$$\sum_{j \neq i} |A_{i,j}| \leq |A_{i,i}| \text{ for all } i.$$

Suppose that $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ is a diagonally dominant system. Show that the Jacobi iteration will converge to the solution when, for at least one row k, we have:

$$\sum_{j \neq k} |A_{k,j}| < |A_{k,k}|.$$

For this result to be true, we need to assume one additional property: the linear system is "fully coupled" in the sense that you cannot break the system up into two pieces where one depends on the other as in:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ 0 & \boldsymbol{C} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \end{bmatrix}.$$

A related assumption you may make is that the linear system you are trying to solve is called "irreducible" which means has the same definition as the ability to break it into two pieces, but is also associated with a relationship between any two equations. You may want to consult with other references as related theorems are proven in many textbooks.

Aside for those who want to dig deep. This structure above actually isn't really a problem as it can be determined by a simple connected components analysis on the graph corresponding to non-zeros of a matrix A. We could then solve $\boldsymbol{C}\mathbf{y} = \mathbf{c}$ and then $\boldsymbol{A}\mathbf{x} = \mathbf{b} - \boldsymbol{B}\mathbf{y}$ but for Jacobi to work, then, we would need both $\boldsymbol{A}$ and $\boldsymbol{C}$ to have one row where diagonal dominance holds with strict inequality. (And further assume that $\boldsymbol{A}$ and $\boldsymbol{C}$ themselves are also not decomposable.)

2. One of the common classes of matrices are called diagonally dominant. There are a number of subclasses of diagonal dominance. Here, we will study a related property. Consider a general linear system of equations:

   $$\boldsymbol{A}\mathbf{x} = \mathbf{b}.$$

   Let $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{N}$ represent a splitting of $\boldsymbol{A}$ into diagonal and non-diagonal terms. So $D_{ij} = A_{ij}$ if $i == j$ and $D_{ij} = 0$ otherwise. Then note that the off diagonal terms $\boldsymbol{N}$ satisfy $\boldsymbol{N} = \boldsymbol{A} - \boldsymbol{D}$. The only property we know about $\boldsymbol{A}$ is that there exists a vector norm $g(\mathbf{x})$ such that $g(\boldsymbol{N}\mathbf{x}) \leq \gamma g(\boldsymbol{D}\mathbf{x})$ for all $\mathbf{x}$ and for some fixed $\gamma < 1$, and we also have that $\boldsymbol{D}_{ii} \neq 0$. Show that the Jacobi iteration will converge to the solution for such a system.

3. Implement the Gauss-Seidel method. For each iteration through all n rows, you must use work proportional to the number of non-zeros in the matrix. Stop your iteration when the relative residual drops below 10^{-4}.

10 EIGENVALUES & THE POWER METHOD

Learning objectives
1. See a variety of ways to think about eigenvalues
2. Look at the power method

**Note, these notes are still being edited. There are a huge diversity of perspectives and geometric interpretations of eigenvalues and eigenvectors, so it's challenging to know how to show them. I'm working on some pictures to help. **

There are a variety of ways to derive and define the eigenvalues of a matrix $\boldsymbol{A}$. The most general definition of an eigenvalue of a matrix is a value λ such that $\det(\boldsymbol{A}-\lambda\boldsymbol{I}) = 0$. This definition, however, obscures much of the utility of eigenvalues of symmetric matrices (which are extremely common).

10.1 CRITICAL DIRECTIONS

** Still working on this section. Skip it now! **

The eigenvalues and eigenvectors of a symmetric, positive definite matrix $\boldsymbol{A}$ are critical directions in the quadratic function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\boldsymbol{A}\mathbf{x}$$

that are invariant to transformations.

For a symmetric matrix $\boldsymbol{A}$, then the eigenvalues of $\boldsymbol{A}$ are the *stationary points* of the following optimization problem:

$$\begin{array}{ll} \underset{\mathbf{x}}{\text{maximize}} & \mathbf{x}^T\boldsymbol{A}\mathbf{x} \\ \text{subject to} & \|\mathbf{x}\|^2 = 1 \end{array} \qquad (10.1)$$

10.2 STATIONARY POINTS

To go ahead and define something in terms of another definition: stationary points are those points where the Lagrangian of the problem has zero derivative. And what is the Lagrangian? It's a function that balances tradeoffs between the objective function $\mathbf{x}^T\boldsymbol{A}\mathbf{x}$ and the constraint $\|\mathbf{x}\|^2 = 1$

$$\mathcal{L}(\mathbf{x}, \lambda) = \mathbf{x}^T\boldsymbol{A}\mathbf{x} - \lambda \cdot (\mathbf{x}^T\mathbf{x} - 1).$$

The gradient of this function is just

$$\partial\mathcal{L}/\partial\mathbf{x} = 2\boldsymbol{A}\mathbf{x} - 2\lambda\mathbf{x}$$

$$\partial\mathcal{L}/\partial\lambda = \mathbf{x}^T\mathbf{x} - 1.$$

So at a stationary point, by definition, we have

$$\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x} \qquad \mathbf{x}^T\mathbf{x} = 1.$$

Conclusion: any stationary point of (10.1) is a pair:

$$(\mathbf{x}, \lambda) \text{ where } \boldsymbol{A}\mathbf{x} = \lambda \mathbf{x}$$

which implies that $(\boldsymbol{A} - \lambda \boldsymbol{I})\mathbf{x} = 0$ and also that $\det(\boldsymbol{A} - \lambda \boldsymbol{I}) = 0$.

Note that this analysis gives the same result for *minimizing* the problem instead of maxing

10.3 THE POWER METHOD TO FIND EIGENVALUES

Given that we have an optimization problem, one strategy to produce an algorithm is to seek a maximizer of $\mathbf{x}^T \boldsymbol{A}\mathbf{x}$ where $\|\mathbf{x}\| = 1$. Because the goal is a maximizer, we would do gradient *ascent* instead of gradient *descent*. However, this time we have a constraint that makes the problem more complicated. A simplistic strategy to handle this constraint is just to take a gradient step:

$$\mathbf{y} = \mathbf{x}^{(k)} + 2\gamma_k \boldsymbol{A}\mathbf{x}^{(k)}$$

and to project it back onto the feasible set:

$$\mathbf{x}^{(k+1)} = \operatorname{argmin}_{\mathbf{z}} \|\mathbf{z} - \mathbf{y}\| \text{ where } \|\mathbf{z}\| = 1.$$

A quick analysis similar to (10.1) shows that $\mathbf{z} = \gamma \mathbf{y}$ for some γ such that $\|\mathbf{z}\| = 1$. That is to say, we just take $\mathbf{y}$ and normalize it.

This gives us the iteration:

$$\mathbf{x}^{(k+1)} = \frac{\mathbf{x}^{(k)} + 2\gamma_k \boldsymbol{A}\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)} + 2\gamma_k \boldsymbol{A}\mathbf{x}^{(k)}\|}.$$

Again, we are interested in maximizing $\mathbf{x}^{(k+1)^T} \boldsymbol{A}\mathbf{x}^{(k+1)}$. This suggests taking γ_k large. In the limit as $\gamma_k \to \infty$ [1] then we find that

$$\mathbf{x}^{(k+1)} = (\boldsymbol{A}\mathbf{x})/\|\mathbf{x}\|.$$

This is the power method!

Definition 10.1 (the power method)
Let $\mathbf{x}^{(0)}$ be any vector. Then the power method is the iteration

$$\mathbf{x}^{(k+1)} = (\boldsymbol{A}\mathbf{x}^{(k)})/\|\mathbf{x}^{(k)}\|.$$

There are no *eigenvalues* in the power method. Instead, there are only eigenvectors. To get the eigenvalue, we need to look at the Rayleigh quotient

$$\lambda^{(k)} = \mathbf{x}^{(k)^T} \boldsymbol{A}\mathbf{x}^{(k)}.$$

This quantity can often be computed with minimal overhead because we need to compute the vector $\boldsymbol{A}\mathbf{x}^{(k)}$ to get the next iterate of the power method.

[1] — TODO – work out this derivation more. Can we show that $\gamma_k \to \infty$ is a natural step?

10.4 CONVERGENCE OF THE POWER METHOD

First, we need to show that the power method is really a simple algorithm. That is, we need to show that $\mathbf{x}^{(k)} = \boldsymbol{M}^k \mathbf{x}^{(0)}$ for some matrix. This type of simple statement will not quite be possible, we just need one slight correction to handle a sticky situation with the norm.

Theorem 10.2
Let $\mathbf{x}^{(k)}$ be the kth iterate of the power method starting from $\mathbf{x}^{(0)}$. Then $\mathbf{x}^{(k)} = \boldsymbol{A}^k \mathbf{x}^{(0)} / \|\boldsymbol{A}^k \mathbf{x}^{(0)}\|$.

PROOF This holds for $\mathbf{x}^{(1)}$ given that this is the explicit iteration. To show that it holds for all future iterations, we proceed inductively. Assume that it is true for the kth iteration: $\mathbf{x}^{(k)} = \boldsymbol{A}^k \mathbf{x}^{(0)} / \|\boldsymbol{A}^k \mathbf{x}^{(0)}\|$. This also means that $\mathbf{x}^{(k)} = \rho_k \boldsymbol{A}^k \mathbf{x}^{(0)}$ for some scalar ρ_k. Thus,

$$\mathbf{x}^{(k+1)} = \rho_k \boldsymbol{A}^{k+1} \mathbf{x}^{(k)} / \|\rho_k \boldsymbol{A}^{k+1} \mathbf{x}^{(k)}\|.$$

And now ρ_k cancels out of the equation and we are done. ■

This shows that we do have a simple algorithm and also that the long term behavior will be governed by matrix powers.

10.5 TERMINATION

A good way to terminate the iteration is to check if we satisfy the eigenvalue residual

$$\boldsymbol{A}\mathbf{x}^{(k)} - \lambda^{(k)} \mathbf{x}^{(k)} \approx 0$$

EXERCISES

1. This should be review, but is a case where eigenvalues are useful. Prove or disprove these statements.
   (a) The eigenvalues of an $n \times n$ real-valued matrix are always real.
   (b) The solution to $(\boldsymbol{A}^T \boldsymbol{A} + \gamma I)\mathbf{x} = \mathbf{b}$ is unique for any $\gamma > 0$.
   (c) Every $n \times n$ matrix can be written as the sum of two non-singular matrices.
   (d) An $n \times n$ real-valued matrix has an orthogonal set of eigenvectors.
   (e) An $n \times n$ matrix $\boldsymbol{A}$ and its transpose $\boldsymbol{A}^T$ have the same set of eigenvalues.
2. Let $\boldsymbol{A} = \begin{bmatrix} 0 & \boldsymbol{B} \\ \boldsymbol{B} & 0 \end{bmatrix}$. Consider starting the power method from $\mathbf{x} = \begin{bmatrix} \mathbf{c} \\ 0 \end{bmatrix}$. Describe the iterations of the power method $\mathbf{x}^{(k)}$ in terms of the matrix $\boldsymbol{B}$ and $\mathbf{c}$.

FINITELY TERMINATING ALGORITHMS III

ELIMINATION METHODS FOR LINEAR SYSTEMS 11

Learning objectives

1.

Thus far, we've seen methods that solve $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ via a sequence of vector changes. These methods have worked by updating $\mathbf{x}^{(k)}$ to $\mathbf{x}^{(k+1)}$. At no point did they consider changing the system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ into another system $\boldsymbol{B}\mathbf{y} = \mathbf{d}$, where $\mathbf{y}$ is somehow easier to find than $\mathbf{x}$ and we can compute $\mathbf{x}$ from $\mathbf{y}$ in a simple fashion.

The next class of methods we will look at will do exactly this! From $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, we will produce a sequence of systems that get progressively smaller by *eliminating* variables. The methods go by a variety of names: elimination, Gaussian elimination, LU decomposition, Cholesky factorization, and even more names including the Schur complement. However, the key idea is almost always the same.

11.1 VARIABLE ELIMINATION

Consider the case where we are solving a system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ then we can write this out and highlight the first row as follows:[1]

$$\boldsymbol{A} = \begin{bmatrix} \alpha & \mathbf{c}^T \\ \mathbf{d} & \boldsymbol{R} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} \beta \\ \mathbf{f} \end{bmatrix}.$$

Let the solution correspond to the elements

$$\mathbf{x} = \begin{bmatrix} \gamma \\ \mathbf{y} \end{bmatrix}$$

so that

$$\alpha\gamma + \mathbf{c}^T\mathbf{y} = \beta.$$

Then the idea behind all of the elmination methods is that, if we are given $\mathbf{y}$ by some type of oracle, we can compute γ from $\mathbf{y}$

$$\gamma(\mathbf{y}) = \frac{1}{\alpha}\left(\beta - \mathbf{c}^T\mathbf{y}\right).$$

This is neat, it says that if you had all by one solution of your system, it's easy to find missing element. [2]

We aren't quite done, however, because this hasn't simplified or changed or system at all. To do that, note that the remaining equations give the expression

$$\gamma\mathbf{d} + \boldsymbol{R}\mathbf{y} = \mathbf{f}.$$

This expression involves γ and $\mathbf{y}$. But we have γ as a function of $\mathbf{y}$ and so let's just substitute that in. The result is an expression purely in terms of $\mathbf{y}$

$$\gamma(\mathbf{y})\mathbf{d} + \boldsymbol{R}\mathbf{y} = \frac{1}{\alpha}\left(\beta - \mathbf{c}^T\mathbf{y}\right)\mathbf{d} + \boldsymbol{R}\mathbf{y} = \mathbf{f}.$$

[1] Of course, the curious will wonder what is special about the first row. As is common, there is nothing special about the first row and this could be done for any row. We'll return to this idea later.

[2] Careful readers will note that we need $\alpha \neq 0$ for this idea to work.

By re-arranging the equations, we arrive at the following linear system:

$$(\boldsymbol{R} - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T)\mathbf{y} = \mathbf{f} - \frac{\beta}{\alpha}\mathbf{d}.$$

EXAMPLE 11.1 Suppose we have:

$$\begin{bmatrix} -2 & -1 & -4 & -1 \\ -1 & -5 & -5 & -2 \\ 4 & 5 & 2 & 0 \\ -2 & -2 & -1 & 0 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 4 \\ 4 \\ -5 \\ 2 \end{bmatrix}$$

then

$$(\boldsymbol{R} - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T)\mathbf{y} = \begin{bmatrix} -4.5 & -3 & -1.5 \\ 3 & -6 & -2 \\ -1 & 3 & 1 \end{bmatrix} \mathbf{y} = \begin{bmatrix} 2 \\ 3 \\ -2 \end{bmatrix}$$

where $\mathbf{y} = \begin{bmatrix} -1 \\ -14/3 \\ 11 \end{bmatrix}$ and $\gamma = 7/3$ so $\mathbf{x} = \begin{bmatrix} 7/3 \\ -1 \\ -14/3 \\ 11 \end{bmatrix}$ ◆

If $\boldsymbol{A}$ was $n \times n$, then this method takes the system

$$(\boldsymbol{A}, \mathbf{b}) \quad \text{to} \quad (\boldsymbol{R} - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T, \mathbf{f} - \frac{\beta}{\alpha}\mathbf{d})$$

where this new system is $(n-1) \times (n-1)$. Hence, we arrive at an *easier* or smaller system to solve! To solve it, we can apply the same idea again until we get down to a 1×1 system. This algorithm is easy to implement on a computer that supports recursion.

```
function elimination_solve(A::Matrix, b::Vector)
  m,n = size(A)
  @assert(m==n, "the system is not square")
  @assert(n==length(b), "vector b has the wrong length")
  if n==1
    return [b[1]/A[1]]
  else
    R = A[2:end,2:end]
    c = A[1,2:end]
    d = A[2:end,1]
    alpha = A[1,1]
    y = elimination_solve(R-d*c'/alpha,
            b[2:end]-b[1]/alpha*d)
    gamma = (b[1] - c'*y)/alpha
    return pushfirst!(y,gamma)
  end
end
```

This idea is called *variable elimination*. We elminate the variable γ from the system of equation $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ by solving for its expression and them substituting that solution into the rest of the equations.

11.2 VARIABLE ELIMINATION AS A MATRIX PRODUCT

The really interesting part about variable elimination is that we can express it as a matrix expression! The following expression seems like magic. Essentially, by examining the above equation long enough, we can deduce an expression like the following. It allows us to express the elimination operation as a matrix itself. Let

$$\boldsymbol{A} = \begin{bmatrix}\alpha & \mathbf{c}^T \\ \mathbf{d} & \boldsymbol{R}\end{bmatrix} \text{ then } \underbrace{\begin{bmatrix}1 & 0 \\ -\mathbf{d}/\alpha & \boldsymbol{I}\end{bmatrix}}_{=\boldsymbol{L}_1}\underbrace{\begin{bmatrix}\alpha & \mathbf{c}^T \\ \mathbf{d} & \boldsymbol{R}\end{bmatrix}}_{=\boldsymbol{A}}\underbrace{\begin{bmatrix}1 & -\mathbf{c}^T/\alpha \\ 0 & \boldsymbol{I}\end{bmatrix}}_{=\boldsymbol{U}_1} = \begin{bmatrix}\alpha & 0 \\ 0 & \boldsymbol{R} - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T\end{bmatrix}.$$

Note that $\boldsymbol{L}_1$ is a non-singular matrix of the form:

$$\boldsymbol{L}_1 = \boldsymbol{I} - \mathbf{v}\mathbf{e}_1 \text{ where } v_1 = 0.$$

This means that $\boldsymbol{L}_1^{-1} = \boldsymbol{I} + \mathbf{v}\mathbf{e}_1$, which can be verified because $\boldsymbol{L}_1\boldsymbol{L}_1^{-1} = \boldsymbol{I}$. Likewise, $\boldsymbol{U}_1 = \boldsymbol{I} - \mathbf{e}_1\mathbf{u}^T$ where $\mathbf{u}_1 = 0$. It's inverse is $\boldsymbol{I} + \mathbf{e}_1\mathbf{u}^T$ as well. Using these matrices, we can transform:

$$\boldsymbol{A}\mathbf{x} = \mathbf{b} \to \boldsymbol{L}_1\boldsymbol{A}\boldsymbol{U}_1\boldsymbol{U}_{-1}\mathbf{x} = \boldsymbol{L}_1\mathbf{b}.$$

If we expand this block-wise, then we get:

$$\begin{bmatrix}\alpha & 0 \\ 0 & \boldsymbol{R} - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T\end{bmatrix}\begin{bmatrix}\gamma + \frac{1}{\alpha}\mathbf{c}^T\mathbf{y} \\ \mathbf{y}\end{bmatrix} = \begin{bmatrix}1 & 0 \\ -\mathbf{d}/\alpha & \boldsymbol{I}\end{bmatrix}\begin{bmatrix}\beta \\ \mathbf{f}\end{bmatrix},$$

which is exactly our reduced system.

Consequently, we can express our entire sequence of reductions as follows:

$$\boldsymbol{L}_{n-1}\boldsymbol{L}_{n-2}\cdots\boldsymbol{L}_1\boldsymbol{A}\boldsymbol{U}_1\boldsymbol{U}_2\cdots\boldsymbol{U}_{n-1} = \begin{bmatrix}\alpha_1 & & & \\ 0 & \alpha_2 & & \\ 0 & 0 & \ddots & \\ 0 & 0 & 0 & \alpha_n\end{bmatrix} = \boldsymbol{D}$$

or

$$\boldsymbol{A} = \boldsymbol{L}_1^{-1}\boldsymbol{L}_2^{-1}\cdots\boldsymbol{L}_{n-1}^{-1}\boldsymbol{D}\boldsymbol{U}_{n-1}^{-1}\boldsymbol{U}_{n-2}\cdots\boldsymbol{U}_1^{-1}.$$

It turns out that these elimination matrix $\boldsymbol{L}_i^{-1}$ and $\boldsymbol{U}_i^{-1}$ have some rather special properties that allow us to realize this form in an exceedingly simple way. For all i we have $\boldsymbol{L}_i^{-1} = (\boldsymbol{I} + \mathbf{v}\mathbf{e}_i)$ with $v_1 = v_2 = \ldots = v_i = 0$ and so

$$\boldsymbol{L}_i^{-1}\boldsymbol{L}_j^{-2} = (\boldsymbol{I}+\mathbf{v}_i\mathbf{e}_i^T)(\boldsymbol{I}+\mathbf{v}_j\mathbf{e}_j^T) = \boldsymbol{I}+\mathbf{v}_i\mathbf{e}_i^T+\mathbf{v}_j\mathbf{e}_j^T+\mathbf{v}_i\mathbf{e}_i^T\mathbf{v}_j\mathbf{e}_j^T = \boldsymbol{I}+\mathbf{v}_i\mathbf{e}_i^T+\mathbf{v}_j\mathbf{e}_j^T \text{ when } i < j.$$

This enables us to quickly compute these as follows:

```
function myreduce_all(A::Matrix)
  A = copy(A) # save a copy
  n = size(A,1)
  L = Matrix(1.0I,n,n)
  U = Matrix(1.0I,n,n)
  d = zeros(n)
  for i=1:n-1
    alpha = A[i,i]
    d[i] = alpha
    U[i,i+1:end] = A[i,i+1:end]/alpha
    L[i+1:end,i] = A[i+1:end,i]/alpha
    A[i+1:end,i+1:end] -=
      A[i+1:end,i]*A[i,i+1:end]'/alpha
  end
  d[n] = A[n,n]
  return L,U,d
end
L,U,d = myreduce_all(A)
L*Diagonal(d)*U - A
```

This is what is most commonly called the *LU* decomposition of a matrix.

> ***Definition 11.2 (LU Decomposition)***
> Let $\boldsymbol{A}$ be a non-singular matrix, then if we do not encounter a zero pivot,[3] we can write
>
> $$\boldsymbol{A} = \boldsymbol{LDU}$$
>
> where $\boldsymbol{L}$ and $\boldsymbol{U}$ have ones on the diagonal and $\boldsymbol{L}$ is lower-triangular and $\boldsymbol{U}$ is upper-triangular. Often, this will be simplified so that $\boldsymbol{A} = \boldsymbol{LU}$ where $\boldsymbol{D}$ is incorporated into $\boldsymbol{U}$.

11.3 PIVOTING

Suppose we start with the system

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}.$$

This has the solution $x = 1, y = 5$.

Then if we try the variable elimination approach, our first step is

$$\begin{gathered} 0x + y = 5 \\ x = \frac{5-y}{0} \\ \text{break!} \end{gathered}$$

This scenario involves division by zero because x really is not really a component of that system!

The solution is easy, if we wish to eliminate x from this equation, we need to use an equation that includes x.[4] In this case, we can simply swap rows

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} 6 \\ 5 \end{bmatrix}.$$

[3] We will see this in the very next paragraph!

[4] This is identical to how in Jacobi and Gauss-Seidel, if we wished to update the value for a variable x_i, we needed to use an equation that used the variable x_i.

After which we have

$$x + y = 6 \to x = 6 - y$$

$$y = 5$$

which we can quickly solve.

Computers need more structure in order to realize these same things. Pivoting is the idea they use to reorder the equations.

Definition 11.3 (Pivoting)
Pivoting reorders the equations (rows) of $\boldsymbol{A}$ in a linear system so that we can compute an LU-decomposition for any non-singular system.

We can always use pivoting to find a variable to eliminate.

Theorem 11.4
If $\boldsymbol{A}$ is non-singular, then at each step of an LU factorization, there must be a variable and equation pair that we can eliminate.

PROOF Assume by way of contradiction that we *cannot* find an equation (row) to eliminate a variable (column) in the kth step of an LU factorization. After $k - 1$ steps of an LU factorization we have

$$\boldsymbol{L}_{k-1}\boldsymbol{L}_{k-2}\cdots\boldsymbol{L}_1\boldsymbol{A}\boldsymbol{U}_1\boldsymbol{U}_2\cdots\boldsymbol{U}_{k-1} = \begin{bmatrix} d_1 & & & & \\ & d_2 & & & \\ & & \ddots & & \\ & & & d_{k-1} & \\ & & & & \boldsymbol{R} \end{bmatrix}$$

By our assumption, we are in the scenario where there is no equation (row) of $\boldsymbol{R}$ to eliminate the kth variable. The kth variable is involved in the first column of $\boldsymbol{R}$. This means that the first column of $\boldsymbol{R}$ is entirely zero. This implies that $\boldsymbol{R}$ is singular because it has a column that is entirely zero.

Now, $\boldsymbol{A}$ is non-singular, as are the products

$$\boldsymbol{L}_{k-1}\boldsymbol{L}_{k-2}\cdots\boldsymbol{L}_1 \text{ and } \boldsymbol{U}_1\boldsymbol{U}_2\cdots\boldsymbol{U}_{k-1}.$$

Consequently, the left-hand side is non-singular, which means the right hand side must be as well. However, our assumption implied that $\boldsymbol{R}$ was singular, which is how we arise at the contradiction. ■

This gives rise to the following algorithm for solving a system of linear equations.

```
function solve_with_simple_pivoting!(A::Matrix, b::Vector)
  m,n = size(A)
  @assert(m==n, "the system is not square")
  @assert(n==length(b), "vector b has the wrong length")
```

```
    if n==1
      return [b[1]/A[1]]
    else
      # let's make sure we have an equation
      # that we can eliminate!
      α = A[1,1]
      newrow = 1
      if α == 0
        for j=2:n # search the current column
          if A[j,1] != 0
            newrow = j
            break
          end
        end
        if newrow == 1
          error("the system is singular")
        end
      end
      # swap rows 1, and newrow
      if newrow != 1
        tmp = A[1,:]
        A[1,:] .= A[newrow,:]
        A[newrow,:] .= tmp
        b[1], b[newrow] = b[newrow], b[1]
      end
      D = A[2:end,2:end]
      c = A[1,2:end]
      d = A[2:end,1]
      α = A[1,1]
      y = solve_with_simple_pivoting!(D-d*c'/α, b[2:end]-b[1]/α*d)
      γ = (b[1] - c'*y)/α
      return pushfirst!(y,γ)
    end
end
```

11.4 PARTIAL PIVOTING

Add failure case example with singular matrix

EXERCISES

1. Suppose we wish to solve

$$\boldsymbol{M}\mathbf{x} = \mathbf{b}$$

and further suppose that you *already know* some of the values of $\mathbf{x}$. Let us permute and partition $\boldsymbol{M}$ to be a block system:

$$\boldsymbol{M}\mathbf{x} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}$$

where $\mathbf{x}_1$ is what you know.

(a) Show how to solve for $\mathbf{x}_2$ given $\mathbf{x}_1$. Under what conditions is this possible?

   (b) Now, suppose that you have a very kind, but very confused dog that happened to *eat* your flash memory stick holding the piece of $\mathbf{x}_1$ that you knew. However, you had saved your computed $\mathbf{x}_2$ on your Purdue account, and so you have a backup. (This means you can assume that computing $\mathbf{x}_2$ from $\mathbf{x}_1$ is *possible* for this problem if you determined it wasn't always possible above.) Can you get $\mathbf{x}_1$ back?

   (c) Combine these two parts to derive a single linear system to compute $\mathbf{x}_1$ without computing $\mathbf{x}_2$. The system you'll derive is called the *Schur complement*

2. Consider computing the LU decomposition on the matrix

$$\boldsymbol{A} = \boldsymbol{I} + \mathbf{c}\mathbf{e}_1 n^T + \mathbf{e}_n \mathbf{d}^T.$$

   Write down a closed form solution for the matrix $\boldsymbol{L}$ and $\boldsymbol{U}$ such that $\boldsymbol{A} = \boldsymbol{LDU}$.

3. Regarding triangular solves.

   (a) Implement backsolve and forward solve as functions in Julia. Show and document your code.

   (b) Construct a random upper-triangular linear system via:

```
A = triu(randn(n,n));
b = randn(n);
```

   Compare the performance of your backsolve to Julia's backslash method to solve a linear system.

   (c) Use your backsolve and forward solve code, along with Julia's `lu` factorization in order to implement your own linear solver. Present a paragraph or two (and a figure or two) comparing it's speed and accuracy to using the \ solver.

SYMMETRIC POSITIVE DEFINITE SYSTEMS & VARIABLE ELIMINATION 12

12.1 SYMMETRIC & SYMMETRIC POSITIVE DEFINITE SYSTEMS

Note that if $\boldsymbol{A}$ is symmetric, then $\mathbf{d} = \mathbf{c}$ and hence $\boldsymbol{R} - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T = \boldsymbol{R} - \frac{1}{\alpha}\mathbf{c}\mathbf{c}^T$ is symmetric as well.

In fact, if $\boldsymbol{A}$ is symmetric positive definite, then $\mathbf{z}^T\boldsymbol{A}\mathbf{z} > 0$ for any $\mathbf{z}$. We can show that $\boldsymbol{D} - \frac{1}{\alpha}\mathbf{c}\mathbf{c}^T$ is *also* symmetric positive definite too! To do so, we will show that $\mathbf{g}^T\boldsymbol{R}\mathbf{g} - \frac{1}{\alpha}(\mathbf{c}^T\mathbf{g})^2 > 0$ for any $\mathbf{g}$. We consider using a specially chosen vector $\mathbf{z}$ applied to the equation for $\boldsymbol{A}$

$$\mathbf{z}^T\boldsymbol{A}\mathbf{z} = \begin{bmatrix} \rho \\ \mathbf{g} \end{bmatrix}^T \begin{bmatrix} \alpha & \mathbf{c}^T \\ \mathbf{c} & \boldsymbol{R} \end{bmatrix} \begin{bmatrix} \rho \\ \mathbf{g} \end{bmatrix} = \alpha\rho^2 + 2\rho\mathbf{c}^T\mathbf{g} + \mathbf{g}^T\boldsymbol{D}\mathbf{g}.$$

At the moment, ρ is still arbitrary. We can choose it to be anything. However, our goal is to pick ρ such that we learn about $\mathbf{g}^T\boldsymbol{R}\mathbf{g} - \frac{1}{\alpha}(\mathbf{c}^T\mathbf{g})^2$. To do so, let $\rho = -(\mathbf{c}^T\mathbf{g})/\alpha$. Then

$$\alpha\rho^2+2\rho\mathbf{c}^T\mathbf{g}+\mathbf{g}^T\boldsymbol{D}\mathbf{g} = (\mathbf{c}^T\mathbf{g})^2/\alpha-2(\mathbf{c}^T\mathbf{g})^2/\alpha+\mathbf{g}^T\boldsymbol{D}\mathbf{g} = \mathbf{g}^T\boldsymbol{D}\mathbf{g}-(\mathbf{c}^T\mathbf{g})^2/\alpha > 0$$

as required.

This means that if we *eliminate* a variable on a symmetric positive definite system. The remaining system is still symmetric positive definite.

An important consequence of this result is that it means that minimizing a quadratic function is *preserved* under variable elimination.

Suppose we are able to run the LU decomposition of a matrix with no pivoting. In other words, suppose that $\alpha =$ `A[i,i]` is non-zero at each step. In this case, we produce:

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{U}.$$

Now, because $\boldsymbol{A}$ is symmetric, we have

$$\boldsymbol{A} = \boldsymbol{A}^T = (\boldsymbol{L}\boldsymbol{D}\boldsymbol{U})^T = \boldsymbol{U}^T\boldsymbol{D}\boldsymbol{L}^T.$$

This strongly hints that $\boldsymbol{L} = \boldsymbol{U}^T$. This result is indeed correct, as can be verified by looking at the each step of the algorithm on a symmetric $\boldsymbol{A}$ and noting that we preserve symmetry at each step as shown above. However, in general, we cannot assume that any symmetric matrix can be decomposed like this without pivoting. [1]here is a more general $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ factorization that can be computed in such cases. A simple counter example is: $\boldsymbol{A} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$.

[1] T

12.2 THE CHOLESKY DECOMPOSITION

The Cholesky decomposition is the LU decomposition, without pivoting, applied to symmetric positive definite matrix. For a symmetric positive definite matrix, we can show that pivoting is not required. This is actually a corollary of one of the definitions of what it means to be a positive definite matrix.

Consequently, for a symmetric positive definite matrix we always have

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T.$$

Moreover, because $\boldsymbol{A}$ is positive definite, we have $D_{i,i} > 0$. This happens because after each step of the reduction, the reduced matrix is *also* positive definite. The diagonal entries of a positive definite matrix are always positive, and these determine the entries of the diagonal $\boldsymbol{D}$. As shown above, after each elimination step, the matrix remains positive definite as well.

Because $\boldsymbol{D}$ is strictly positive, we can take the square root of each entry and compute

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}^{1/2}(\boldsymbol{D}^{1/2}\boldsymbol{L})^T = \boldsymbol{F}\boldsymbol{F}^T \text{ or } \boldsymbol{F}^T\boldsymbol{F}.$$

This gives the Cholesky routine

```
""" Compute the Cholesky factorization A = FF' and return F """
function (A::Matrix)
  A = copy(A) # save a copy
  n = size(A,1)
  F = Matrix(1.0I,n,n)
  d = zeros(n)
  for i=1:n-1
    alpha = A[i,i]
    d[i] = sqrt(alpha)
    F[i+1:end,i] = A[i+1:end,i]/alpha
    A[i+1:end,i+1:end] -= A[i+1:end,i]*A[i,i+1:end]'/alpha
  end
  d[n] = sqrt(A[n,n])
  return F*Diagonal(d), d
end
PD = A'*A
F,d = myreduce_all_cholesky(PD)
F*F' - A
```

Compare with the previous LU code to see the subtle differences.

What happens if your matrix is not symmetric positive definite? Then at some point in the decomposition, you will have $\alpha < 0$. This is actually one of the fastest ways to test if a matrix is symmetric positive definite as it avoid all eigenvalue computations.

13 GENERAL VARIABLE ELIMINATION

This is a more general discussion on the ideas in the previous sections. When we are elminating

14 PIVOTING & VARIABLE ELIMINATION

Partial pivoting, where we select the equation to use to eliminate based on the largest entry in the row, is by far the most common choice. The key downside to partial pivoting, however, is that it does not allow us to compute the *rank* of the matrix from the decomposition. That is, we might hope that if $\boldsymbol{PA} = \boldsymbol{LDU}$, then we could determine the rank $\boldsymbol{A}$ from the non-zeros of $\boldsymbol{D}$. The following examples shows why this does not work.

EXAMPLE 14.1 Add an example where partial pivoting fails early or late. ◆

There are two other common types of pivoting

Definition 14.2 (Rook pivoting)
This is wrong In variable elimination with rook pivoting, at each step, we choose either the equation with the *largest* magnitude entry for the next variable, or we choose the *variable* in the next equation with the largest magnitude entry.

Fix it!

Definition 14.3 (Complete pivoting)
In variable elimination with complete pivoting, at each step, we choose the equation and variable pair with the *largest* magnitude entry over all those that haven't yet been eliminated.

14.1 ROOK PIVOTING AS A RANK REVEALING DECOMPOSITION

The goal of this section is to show that rook pivoting results in a rank revealing factorization of a matrix.

14.2 COMPLETE PIVOTING AND A MATRIX DECOMPOSITION

For complete pivoting, we can view this as a matrix decomposition

```
function _find_pivot(A, eqsleft, varsleft)
  bestpivot = (first(eqsleft), first(varsleft))
  bestval = abs(A[bestpivot[1], bestpivot[2]])
  for j in varsleft
    for i in eqsleft
```

```
      if abs(A[i,j]) > bestval
        bestval = abs(A[i,j])
        bestpivot = (i,j)
      end
    end
  end
  return bestpivot, bestval
end
function complete_elimination(A)
  eqsleft = Set(1:size(A,1))
  varsleft = Set(1:size(A,2))
  eqorder = Int[]
  varorder = Int[]
  while length(eqsleft) > 0 && length(varsleft) > 0
    pivot, val = _find_pivot(A, eqsleft, varsleft)
    # TODO, finish this...
    push!(eqorder, pivot[1])
    push!(varorder, pivot[2]) # save
  end
end
```

Be

This section was inspired by a discussion with Nick Trefethen, with yet another viewpoint on variable elimination

ELIMINATION METHODS FOR LEAST SQUARES 15

15.1 THE SIMPLE ELIMINATION SOLVE

We can also use variable elimination for least squares problems. Consider

$$\text{minimize} \quad \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\|.$$

Partition $\boldsymbol{A} = \begin{bmatrix} \mathbf{a} & \boldsymbol{C} \end{bmatrix}$ and $\mathbf{x} = \left[\begin{smallmatrix} \gamma \\ \mathbf{y} \end{smallmatrix}\right]$. Then

$$\text{minimize} \quad \|\gamma\mathbf{a} + \boldsymbol{C}\mathbf{y} - \mathbf{b}\|.$$

We proceed as follows, suppose we know $\mathbf{y}$. Let $\mathbf{d} = \boldsymbol{C}\mathbf{y} - \mathbf{b}$. Then this is just the one varible least squares problem

$$\text{minimize} \quad \|\gamma\mathbf{a} - \mathbf{d}\|.$$

If we explain what this is, then we are looking for the best *scaling* of the vector $\mathbf{a}$ to get us as close to possible to $\mathbf{d}$.

> **TODO** Add figure that explains this

A little bit of thinking yields the following insight: the scaling of $\mathbf{a}$ is *closest* to $\mathbf{d}$ when the difference $\gamma\mathbf{a} - \mathbf{d}$ is *orthogonal* to $\mathbf{a}$. If this weren't the case, then we could decrease the distance by moving a little bit in any direction. Hence, the solution γ must satisfy the relationship:

$$\mathbf{a}^T(\gamma\mathbf{a} - \mathbf{d}) = 0 \quad \text{or} \quad \gamma = \frac{1}{\mathbf{a}^T\mathbf{a}}\mathbf{a}^T\mathbf{d}.$$

Now, we proceed as follows and substitute $\gamma(\mathbf{y})$ into our original least squares problem

$$\text{minimize} \quad \|\gamma(\mathbf{y})\mathbf{a} + \boldsymbol{C}\mathbf{y} - \mathbf{b}\| \quad \to \quad \text{minimize} \quad \|\tfrac{1}{\mathbf{a}^T\mathbf{a}}\mathbf{a}^T(\boldsymbol{C}\mathbf{y} - \mathbf{b})\mathbf{a} + \boldsymbol{C}\mathbf{y} - \mathbf{b}\|.$$

We can simplify this expression to

$$\text{minimize} \quad \|(\boldsymbol{I} - \tfrac{1}{\mathbf{a}^T\mathbf{a}}\mathbf{a}^T\mathbf{a})(\boldsymbol{C}\mathbf{y} - \mathbf{b})\|.$$

This new problem has one fewer variable. If we recurse on this idea, we have the following algorithm.

```
function least_squares_eliminate(A,b)
  a = A[:,1]
  na = norm(a)
  q = a/na
  if size(A,2) == 1
    return [a'*b]/na
  end
```

```
    y = least_squares_eliminate(A[:,2:end]-q*q'*A[:,2:end], b - q*q'*b)
    γ = q'*(b - A[:,2:end]*y)/na
    return pushfirst!(y,γ)
  end
```

15.2 A MATRIX VERSION

> **TODO** See if we can get something better here...

The matrix structure in this problem is already slightly apparent. Let $\boldsymbol{T} = (\boldsymbol{I} - \frac{1}{\mathbf{a}^T\mathbf{a}}\mathbf{a}^T\mathbf{a}^T)$. Then we have

$$\text{least-squares}(\boldsymbol{A}, \mathbf{b}) \to \text{least-squares}(\boldsymbol{TAS}, \boldsymbol{T}\mathbf{b}).$$

Here $\boldsymbol{S}$ is a matrix that selects the last $n-1$ columns of a matrix.

Now, it turns out there is an issue here. The matrix $\boldsymbol{T}$ is a special type of matrix called a *projection*. A projection matrix is any matrix where $\boldsymbol{T}^2 = \boldsymbol{T}$. It represents a projection onto a subspace, so $\boldsymbol{T}^2 = \boldsymbol{T}$ because the projection of a projection is the same projection. For this matrix $\boldsymbol{T}$ it's just a few lines of algebra to verify that $\boldsymbol{T}^2 = \boldsymbol{T}$.

> **TODO** Add these lines

This is a small issue, though, because $\boldsymbol{T}\mathbf{a} = 0$ and so $\boldsymbol{TA} = \begin{bmatrix} 0 & \boldsymbol{TC} \end{bmatrix}$. Thus, we lose all the information associated with $\mathbf{a}$ after the first transformation. However, suppose we just *memorize this* and store $\mathbf{a}$ – after normalization – at each iteration into a matrix $\boldsymbol{Q}$.

To be entirely precise, let $\boldsymbol{A} = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix}$. Let $\boldsymbol{T}_1, \ldots \boldsymbol{T}_n$ be the matrix $\boldsymbol{I} - \mathbf{q}_i\mathbf{q}_i^T$ formed in the least squares elimination algorithm at the ith call. Then we have:

$$\mathbf{q}_i = \frac{1}{\|\boldsymbol{T}_{i-1}\cdots\boldsymbol{T}_1\mathbf{a}_i\|}\boldsymbol{T}_{i-1}\cdots\boldsymbol{T}_1\mathbf{a}_i.$$

In a bit of remarkable luck, the matrix $\boldsymbol{Q}$ turns out to be orthogonal. In fact, it's the result of the Gram-Schmidt process we discuss in the next lecture.

Add failure case with singular matrix

LEAST SQUARES VIA QR FACTORIZATION & ORTHOGONALIZATION 16

Learning objectives
1. Target pieces of a matrix for an operation with pieces of the identity matrix. 2. Recognize the QR factorization.

There is another approach to solving the least squares problems

$$\text{minimize} \quad \|\mathbf{b} - \boldsymbol{A}\mathbf{x}\|$$

besides the variable elimination procedure we saw in previous classes. I don't yet have a natural derivation of this particular idea, but I believe it originates around the following set of ideas.

- The geometry of the least squares problems involves working with the span of $\boldsymbol{A}$'s columns, or the range of $\boldsymbol{A}$. In particular, we want to find a point in the range that is as close as possible to $\mathbf{b}$.
- Since this involves working with the range of $\boldsymbol{A}$, it is "natural" to seek an orthogonal basis for it.

And this is what the QR factorization of a matrix encodes: an orthogonal basis for the columns of $\boldsymbol{A}$.

More formally, the QR factorization of a tall $m \times n$ matrix $\boldsymbol{A}$ (with $m \geq n$) is a pair of matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ such that:

- $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$
- $\boldsymbol{Q}$ is square $m \times m$ and orthogonal
- $\boldsymbol{R}$ will also be upper-triangular and $m \times n$, but let's see where that comes from!

The upper-triangular structure appears to arise from early work by Schmidt on orthogonalizing a set of vectors. This is often called the "Gram-Schmidt process" and functions by successive orthogonalization.[1]

16.1 REVIEW OF GRAM-SCHMIDT

This review is meant to remind you of stuff you hopefully learned in previous linear algebra classes.

That is, if we are given a set of three vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}$ then the Gram-Schmidt process builds an orthonormal basis for their span, which is equivalent to building an orthogonal matrix $\boldsymbol{Q}$ such that

$$\begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{bmatrix} = \boldsymbol{Q}\boldsymbol{C}$$

for some non-singular, square matrix $\boldsymbol{C}$. The Gram-Schmidt process begins with the first vector $\mathbf{x}$ and sets the first column of $\boldsymbol{Q}$ to be $\mathbf{x}/\|\mathbf{x}\|$. Then we *project-out* any component of $\mathbf{x}$ on the other vectors. The matrix $\boldsymbol{P}(\mathbf{x}) = \boldsymbol{I} - \frac{\mathbf{x}\mathbf{x}^T}{\mathbf{x}^T\mathbf{x}}$ is a projector[2] to the space orthogonal to the vector $\mathbf{x}$. That is, $\mathbf{x}^T\boldsymbol{P}(\mathbf{x})\mathbf{y} = \mathbf{x}^T\mathbf{y} - \frac{\mathbf{x}^T\mathbf{x}}{\mathbf{x}^T\mathbf{x}}\mathbf{x}^T\mathbf{y} = 0$. Hence, we compute $\mathbf{y}_1 = \boldsymbol{P}(\mathbf{x})\mathbf{y}$, $\mathbf{z}_1 = \boldsymbol{P}(\mathbf{x})\mathbf{z}$. The next vector $\mathbf{q}_2 = \mathbf{y}_1/\|\mathbf{y}_1\|$. and we project $\mathbf{z}_2$ via $\boldsymbol{P}(\mathbf{y}_1)$. This gives us three vectors:

$$\boldsymbol{Q} = \begin{bmatrix} \mathbf{x}/\|\mathbf{x}\| & \mathbf{y}_1/\|\mathbf{y}_1\| & \mathbf{z}_2/\|\mathbf{z}_2\| \end{bmatrix}$$

[1] I am looking into ways of re-deriving these ideas where the upper-triangular structure is one of a few possible natural choices depending on the ideas involved, but so far I haven't hit on anything easy.

[2] A projector matrix *projects* vectors to a subspace S. Because the output from a projector is a new vector in a subspace S, it must be the case that projecting to S again will leave the result unchanged. Hence, $\boldsymbol{P}^2 = \boldsymbol{P}$ for any projector matrix!

where $\mathbf{y}_1 = \boldsymbol{P}(\mathbf{x})\mathbf{y}$ and $\mathbf{z}_2 = \boldsymbol{P}(\mathbf{y}_1)\boldsymbol{P}(\mathbf{x})\mathbf{z}$. We can write this as a matrix equation as follows:

$$\boldsymbol{A} = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{bmatrix} = \begin{bmatrix} \mathbf{x}/\|\mathbf{x}\| & \mathbf{y}_1/\|\mathbf{y}_1\| & \mathbf{z}_2/\|\mathbf{z}_2\| \end{bmatrix} \begin{bmatrix} \|\mathbf{x}\| & C_{1,2} & C_{1,3} \\ 0 & \|\mathbf{y}_1\| & C_{2,3} \\ 0 & 0 & \|\mathbf{z}_2\| \end{bmatrix}$$

where $C_{i,j}$ arises from the projection operations. Consider $C_{1,2}$, which we get from $\mathbf{y}_1 = \boldsymbol{P}(\mathbf{x})\mathbf{y} = \mathbf{y} - \frac{\mathbf{x}^T\mathbf{y}}{\mathbf{x}^T\mathbf{x}}\mathbf{x}$, we can write this to get $C_{i,j}$ for each.

Notice the similarity between this procedure and the successive elimination procedure we had in the previous class. I think this can be turned into a fairly natural derivation, but it requires a little more work.

The point of these derivations is that the Gram-Schmidt process produces an orthogonal basis for the columns of $\boldsymbol{A}$ via successive orthogonalization, which can be written:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$$

for an $m \times n$ matrix $\boldsymbol{Q}$ and a square upper-triangular matrix $n \times n$ matrix $\boldsymbol{R}$. This is often called a “thin” QR factorization because the matrix $\boldsymbol{Q}$ isn’t square but is *tall* instead.

16.2 GENERALIZING TO QR

The idea with the full QR factorization is that we can extend a "thin" QR factorization to a square matrix $\boldsymbol{Q}$ because there are n orthogonal vectors in an n-dimensional space. Given any set of m orthogonal vector (say via Gram-Schmidt), then there exist another $m - n$ vectors that are mutually orthgonal as well. Of course, because these are orthogonal, we don’t need to use them to write the matrix $\boldsymbol{A}$, so the “tail” of $\boldsymbol{R}$ becomes zero.

16.3 USING QR TO SOLVE LEAST SQUARES

Now, let’s show that we can use *any* QR factorization to compute a solution to the least squares problem. Note that $\|\mathbf{x}\| = \|\boldsymbol{Q}\mathbf{x}\| = \|\boldsymbol{Q}^T\mathbf{x}\|$ for any square orthogonal matrix $\boldsymbol{Q}$.

Hence, let $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$ be any full QR factorization with a square matrix $\boldsymbol{Q}$, then

$$\|\mathbf{b} - \boldsymbol{A}\mathbf{x}\| = \|\boldsymbol{Q}^T\mathbf{b} - \boldsymbol{Q}^T\boldsymbol{A}\mathbf{x}\| = \|\hat{\mathbf{b}} - \boldsymbol{R}\mathbf{x}\| = \|\begin{bmatrix} \hat{\mathbf{b}}_1 \\ \hat{\mathbf{b}}_2 \end{bmatrix} - \begin{bmatrix} \boldsymbol{R}_1 \\ 0 \end{bmatrix}\mathbf{x}\|.$$

Here, we used $\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_1 \\ 0 \end{bmatrix}$ where $\boldsymbol{R}_1$ is the first set of n rows of $\boldsymbol{R}$. Because $\boldsymbol{R}$ is upper-triangular, the other elements are always zero.

Note that this form helps us greatly! Note that no matter how we change $\mathbf{x}$, we cannot eliminate $\hat{\mathbf{b}}_2$ from the difference between $\mathbf{b}$ and $\boldsymbol{A}\mathbf{x}$.

Hence, the best we can do to minimize the expression is to set $\mathbf{x}$ so that $\hat{\mathbf{b}}_1 = \boldsymbol{R}_1\mathbf{x}$.

Consequently, we can use any method to produce a QR factorization to solve a least squares problem via the following algorithm:

```
Compute a full or thin QR factorization.
Compute $\hat{\mathbf{b}}_1 =$ first n rows of $\boldsymbol{Q}\mathbf{b}$ when $\boldsymbol{Q}$ is full,
  or $\hat{\mathbf{b}}_1 = \boldsymbol{Q}^T\mathbf{b}$ when $\boldsymbol{Q}$ is $m \times n$.
Solve $\boldsymbol{R}_1\mathbf{x} = \hat{\mathbf{b}}_1$.
Return $\mathbf{x}$
```

16.4 A Givens rotations and QR for a small vector.

Consider the problem of computing a QR factorization for a 2 × 1 vector $\mathbf{v}$. Recall that an orthogonal matrix is a generalization of a rotation, so we can write it as:

$$\boldsymbol{Q} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}.$$

Let's see how to pick $\boldsymbol{Q}$ for $\mathbf{v}$.

An obvious way is to try and compute θ in the above expression such that

$$\boldsymbol{Q}(\theta)\mathbf{v} = \gamma\mathbf{e}_1$$

for some γ.

However, there is a better way to do this! Note that $\boldsymbol{Q}(\theta)$ only has two unknowns, $c = \cos\theta$ and $s = \sin\theta$. To compute $\boldsymbol{Q}$, we just need these two values! Let's write out the equations:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \gamma \\ 0 \end{bmatrix}$$

This gives two equations and two unknowns.

$$v_1 c + v_2 s = \gamma \text{ and } v_2 c - v_1 s = 0.$$

We can solve these to get[3]

$$c = v_1/\gamma \text{ and } s = v_2/\gamma.$$

Because the matrix is orthogonal, we must have $\gamma = \sqrt{v_1^2 + v_2^2}$ or $\gamma = -\sqrt{v_1^2 + v_2^2}$ so that the length of $\mathbf{v}$ doesn't change.

This 2 × 2 matrix $\boldsymbol{Q}(\theta)$ is called a Givens rotation.

16.5 The QR factorization for a 3x1 vector.

Suppose $\mathbf{v}$ is 3 × 1. Then we *could* seek to build a 2d rotation matrix and solve for the coefficients. However, there is an alternative mechanism where we can use matrix structure. Let $\mathbf{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}^T$. Let

[3] The solution here is not unique. Note that we can negate these values as well as they are also a solution. See more discussion in `https://netlib.org/lapack/lawnspdf/lawn148.pdf` Some discussion of how this impacts numerical software is discussed in XXX

16.6 Givens rotations in Julia

TODO

16.7 Computing QR for a column

Consider computing a QR factorization for a $n \times 1$ vector $\mathbf{v}$ now. By the definition, we have:

$$\boldsymbol{Q}\mathbf{v} = \gamma \mathbf{e}_1.$$

where $\gamma = \pm\|\mathbf{v}\|$. This is a good exercise to work out for yourself!

What happens in Julia with 0-columns?

EIGENVALUE AND SINGULAR VALUE COMPUTATIONS 16

Learning objectives

1. Look at deflation as a technique to find the eigenvalue and singular value decomposition of a matrix in a finite number of steps.
2. Recognize a key subroutine that provides a single eigenvalue or singular value pair and how to use that to get others.

In this chapter, we treat the eigenvalue and singular value problems as finitely terminating algorithms rather than iterative algorithms as most textbooks do. This is because we assume that there exist programs to compute the following properties.

$$\text{Program 1:} \quad \begin{array}{ll} \underset{\mathbf{x}}{\text{maximize}} & \|\boldsymbol{A}\mathbf{x}\|_2 \\ \text{subject to} & \|\mathbf{x}\|_2 = 1 \end{array} \qquad \text{Program 2:} \quad \begin{array}{ll} \underset{\mathbf{x}}{\text{maximize}} & \mathbf{x}^T\boldsymbol{A}\mathbf{x} \\ \text{subject to} & \|\mathbf{x}\|_2 = 1. \end{array}$$

For program 2, we also assume that $\boldsymbol{A}$ is symmetric. Note that both of these programs can be solved via the iterative power method discussed in in the previous section. Moreover, even more specialized solvers for them can be built using tools developed elsewhere in this book.

16.8 THE EIGENVALUE DECOMPOSITION

Let $\boldsymbol{A}$ be a symmetric matrix.

16.9 THE SINGULAR VALUE DECOMPOSITION

We will following a similar deflation procedure.

16.10 SOLVING THE SUBPROBLEMS

16.11 INVARIANT SUBSPACES

EXERCISES

1. In class, we discussed deflation techniques for computing more than one eigenvalue and eigenvector with the power method. Briefly, assume that we setup the power method to compute the largest eigenvalue and eigenvector. As a convergence tolerance, use

   $$\|\boldsymbol{A}\mathbf{x} - \lambda\mathbf{x}\| \leq \|\boldsymbol{A}\|_F \varepsilon$$

   where ε is the machine precision for double precision numbers, 'eps(1.0)' in Julia.
   Recall also that deflation is the following procedure: given a matrix $\boldsymbol{A}$ and an eigenvector $\mathbf{x}$ and eigenvalue λ, we compute $\boldsymbol{B} = \boldsymbol{A} - \lambda\mathbf{x}\mathbf{x}^T$.
   This removes the largest eigenvalue and eigenvector.
   Compute a few steps of deflation and

ANALYSIS IV

We have now seen a few different types of algorithms to solve the fundamental matrix computations.

Elimination with and without pivoting for linear systems, Richardson, Steepest Descent, Gauss Seidel, Jacobi, etc.

How should we pick which one to use to solve our problem? There are various ways of thinking about this question. We'll see a few different types of analysis of these algorithms including work and operation count. But we will also consider stability.

TIME & MEMORY REQUIREMENTS 17

The simplest way to analyze the algorithms is in terms of how much time and memory they require. It turns out memory is usually a more pressing constraint than time. So let's start with that one!

17.1 MEMORY REQUIREMENTS

To run the Richardson method or steepest descent method (lecture 8), we require:

1. a way to multiple $\boldsymbol{A}$ by a vector $\mathbf{v}$
2. memory to store two or three vectors of length n. (You saw this on the homework.)

To run the elimination matrix that gave us the LU factorization of a matrix, we require:

1. $O(n^2)$ memory for a general matrix problem because we *change* the matrix after the first step
2. $O(n)$ memory for the changes to the vector $\mathbf{b}$ [1]

For a general dense matrix, there is no difference between the memory requirements. However, for a matrix with any type of structure, or especially sparse structure, then it is easier to think about how to exploit that structure to make the Richardson, steepest descent method run with less memory.

Consider a matrix that is Toeplitz. For Richardson, we only need to store $O(n)$ memory to be able to do the matrix-vector products. Whereas for the LU factorization, we will need to build the matrix at $O(n^2)$ memory in order to run the algorithm.

Of course, there is the problem of how long the Richardson method takes, and whether or not it will even converge. (Remember, we only showed that we could guarantee convergence for a symmetric positive definite matrix.)

17.2 THE FLOP COUNT

One common way of evaluating the work of an algorithm is in terms of the number of floating point operations it does. This measure is classical, but still important, because

- floating point operations used to be much more expensive than other processor operations
- most systems now have special hardware dedicated to floating point computation such as vector processing units (called AVX on Intel

[1] These can often be done in place as well.

processors) or GPUs.

Hence, having a fairly exact count of the number of FLOPS allows us to understand how fast a particular operation may go on a computer.

Aside. There is a lot more to high performance computations, such as cache efficient, blocking, vectorization, than just purely understanding the FLOPS, however. The use of FLOPS counts are largely just to suggest an upper-bound on performance.

Aside Again. What is a FLOP and FLOPS? A floating point operation (FLOP). Or a floating point operation per second (FLOPS)? Suffice it to say that these acronyms are used inconsistently and potentially mixed. The context determines if we are counting floating point operations (FLOPs) or measuring floating point operations per second (FLOPS). This is extremely confusing when there is a prefix. What is one gigaflop? One billion floating point operations or one billion floating point operations per second. Again, context will determine if we are counting or timing.

17.3 A WARM UP: THE FLOP COUNT OF MATRIX-MULTIPLY.

Consider the following algorithm for multiplying two matrices.

```
function matmul(A::Matrix, B::Matrix)
  m,k = size(A)
  k,n = size(B)
  C = similar(A, m, n)
  fill!(C, 0)
  for j=1:n
    for i=1:m
      for r=1:k
        C[i,j] += A[i,r]*B[r,j]
      end
    end
  end
  return C
```

We can count the number of FLOPs by looking just at the inner-loop, which is an inner-product. Let's consider a simple 4-element vector

$$C[i,j] = A[i,1]*B[1,j]+A[i,2]*B[1,2]+A[i,3]*B[3,j]+A[i,4]*B[4,j].$$

There are 4 multiplications and 3 additions, for a total of 7 flops. The way our code above works, however, is to use 8 flops because we always add to the existing value:

$$C[i,j] \leftarrow C[i,j]+A[i,1]*B[1,j]+A[i,2]*B[1,2]+A[i,3]*B[3,j]+A[i,4]*B[4,j].$$

Here, we have the convention that $\leftarrow$ is the "assign" operation, which is commonly expressed as = inside of programming languages, but has a distinct meaning from the mathematical = operation. For instance, `x = 2*x` is perfectly common computer code, but if used mathematically essentially implies that $x = 0$.

Consequently, there are $2r$ FLOPs in the innermost loop. This is executed mn times, so there are $2rmn$ FLOPs in this computation of a matrix-matrix product.

17.4 THE FLOP COUNT OF LU

Let's consider something more interesting, the pivoted LU decomposition of a matrix. Our code is

```
function myreduce_all_pivot(A::Matrix)
  A = copy(A) # save a copy
  n = size(A,1)
```

```
 4    L = Matrix(1.0I,n,n)
 5    U = Matrix(1.0I,n,n)
 6    d = zeros(n)
 7    p = collect(1:n)
 8    for i=1:n-1
 9      maxval = abs(A[i,i])
10      newrow = i
11      for j=i+1:n
12        if abs(A[j,i]) > maxval
13          newrow = j
14          maxval = abs(A[j,i])
15        end
16      end
17      if maxval < eps(1.0)
18        error("the system is singular")
19      end
20
21      j = newrow
22      # swap the ith row/column
23      tmp = A[i,:]
24      A[i,:] .= A[j,:]
25      A[j,:] .= tmp
26
27      p[i],p[j] = p[j], p[i]
28      L[i,1:i-1], L[j,1:i-1] = L[j,1:i-1], L[i,1:i-1]
29
30      α = A[i,i]
31      d[i] = α
32      U[i,i+1:end] = A[i,i+1:end]/α
33      L[i+1:end,i] = A[i+1:end,i]/α
34      A[i+1:end,i+1:end] -= A[i+1:end,i]*A[i,i+1:end]'/α
35    end
36    d[n] = A[n,n]
37    return L,U,d,p
38  end
```

In the first block of code (lines 2-7) we do no FLOPs because it's just allocating memory. There are $n - 1$ executions of the loop on line 8, and each execution consists of

LINE 12 : $(n - i)$ floating point comparisons (these are counted because they may require something like a subtraction), whereas absolute values are not because they can be done with an extremely simple operation.

LINES 21-28 : a swap, that does not involve FLOPs

LINES 32-33 : $2(n - i)$ divisions

LINE 34 : $3(n - i)^2$ multiplications, additions, and divisions (one of each for each of $(n - 1)^2$ elements.

Next, we have to sum this over all i. So the total FLOP count of this implementation is

$$\sum_{i=1}^{n-1} (n - i) + 2(n - i) + 3(n - i)^2.$$

There are various rules to compute these sums called a "finite calculus" that work like a crank equivalent to standard calculus. However, I find it easier just to use Wolfram Alpha, which yields

$$\sum_{i=1}^{n-1}(n-i)+2(n-i)+3(n-i)^2=n^3-n.$$

Note, however, that we can reduce Line 34 to $2(n-i)^2$ FLOPs if we use the elements of either L or U by avoiding the division. Also, note that if we actually computed the standard LU decomposition where $U[i,i+1:end]=A[i,i+1:end]$ then we could further avoid $n-i$ divisions, yielding a more efficient count of [2]

$$\sum_{i=1}^{n-1}(n-i)+(n-i)+2(n-i)^2=\frac{2}{3}(n^3-n).$$

[2] These final sums must be integer values as they are counts of integers!

18 SENSITIVITY & CONDITIONING

One of the most interesting matrices and matrix problems arose around the turn of the century when David Hilbert wanted to look at approximating functions by polynomials. This leads to a way to quantify how difficult a linear systems is to solve on the computer, called the condition number. A derivation of this matrix is given in lecture 34, but the matrix is simple to state:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & \cdots & \\ 1/2 & 1/3 & 1/4 & 1/5 & \cdots & \\ 1/3 & 1/4 & 1/5 & 1/6 & \cdots & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ & & & & & A_{ij} = \frac{1}{i+j-1} \end{bmatrix}$$

Linear systems with this matrix are incredibly sensitive to changes in the right hand side. Here is an example where we *build* a solution to the linear system of equations.

```
function hilbert(n)
  return [1/(i+j-1) for i=1:n, j=1:n]
end
n = 15
A = hilbert(n)
x = ones(n)
b = A*x
bt = b  + eps(1.0)*rand(StableRNG(1), n)
xt = A\bt
```

```
julia> xt = A\bt
15-element Vector{Float64}:
   1.000000013796333
   0.9999984758819328
   1.000027709924497
   1.0002316088164172
   0.989170724718275
   1.1282615824076014
   0.18208208600167303
   4.274200531997128
  -7.762766220634516
  17.113970204553503
 -19.459355915473054
  18.642614568867764
  -8.871061666536182
```

```
   4.233468847532954
   0.5291574408371379
```

This shows that we get the answer completely wrong even though we made an incredibly small changes to the right hand side!

```
julia> b - bt
15-element Vector{Float64}:
  0.0
  0.0
 -2.220446049250313e-16
  0.0
 -2.220446049250313e-16
  0.0
 -2.220446049250313e-16
 -2.220446049250313e-16
  0.0
  0.0
 -2.220446049250313e-16
 -1.1102230246251565e-16
 -2.220446049250313e-16
  0.0
  0.0
```

18.1 AN INFORMAL ANALYSIS

To understand why, consider what happens if we make a tiny error in evaluating $\mathbf{b}$. The answer we want is the solution $\mathbf{x}$ in $A\mathbf{x} = \mathbf{b}$ but we actually see $\mathbf{b}' = \mathbf{b} + \mathbf{d}$ where $\|\mathbf{d}\|$ is very small. Then, we would compute

$$A\mathbf{y} = \mathbf{b}' \qquad \mathbf{y} = \mathbf{x} + A^{-1}\mathbf{d}.$$

Consequently, as a rough measure of sensitivity, we'd consider $\|A^{-1}\mathbf{d}\|$.

As you might expect, the Hilbert matrix is sufficiently structured that we can just write down the inverse after some tedious calculation. (Or we can just look it up on—say—Wikipedia.)

$$A^{-1}_{ij} = (-1)^{i+j}\binom{n+i-1}{n-j}\binom{n+j-1}{n-i}\binom{i+j-2}{i-1}^2$$

This gets large extremely quickly.[1]

Hence, we have $\mathbf{y} = \mathbf{x} + \text{big} \cdot \text{small}$, so in general, we'll expect large changes to $\mathbf{y}$ if we slightly change our right hand side.

18.2 A MORE REFINED ANALYSIS

The origin of the Hilbert problem is to approximate a function $f(x)$ by a polynomial. That problem itself actually is *well conditioned*.[2] Small

[1] It seems like a good exercise in combinatorics to work out what the one-norm of this matrix is!

Aside. For those with additional background on numerical analysis, we suggest repeating this section with more sophisticated and accurate quadrature methods.

[2] The matrix method is just one method of solving the original problem, that happens to result in a problem with an ill-conditioned linear system of equations.

changes to $f(x)$ give small changes to the polynomial.

The real problem here is that we have chosen to represent the polynomial in a monomial basis. This is know to result in problems that occur because monomials are an ill-conditioned basis for polynomials. Small changes to the monomial coefficients cause big changes to the polynomial functions they represent. In this case, an issue that arises for this problem is that *a small perturbation to* **b** actually gives a large perturbation to $f(x)$. Hence, we *should* see large changes to **x** even for small changes in **b**.

Let's briefly study this perspective. Recall that

$$b_j = \int_0^1 f(x)x^{j-1}\,dx.$$

Suppose we use a crude approximation of the integral via a set of equally spaced points $x_1 = 0, x_2 = 1/N, x_3 = 2/N, \ldots, x_N = (N-1)/N, x_{N+1} = 1$, then

$$b_j \approx \sum_{k=0}^{N} f(x_{k+1})x^{j-1} 1/N.$$

Let **f** be a length N vector of these function values $f_i = f(x_i)$. Then

$$\mathbf{b} \approx \frac{1}{N} \underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_N^{n-1} \end{bmatrix}}_{=\boldsymbol{M}} \underbrace{\begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_N) \end{bmatrix}}_{\mathbf{f}} \quad \text{or more compactly} \quad \mathbf{b} = \boldsymbol{M}\mathbf{f}.$$

The matrix $\boldsymbol{M}$ is non-singular as long as the points $x_1, \ldots, x_N$ are distinct, which they are for equally spaced points. So given a change to **b**, then $\boldsymbol{M}^{-1}\mathbf{b}$ gives the change to the function values that **b** represents.

18.3 A FORMAL ANALYSIS: THE CONDITION NUMBER

This type of analysis has been formalized by studying the condition number of a problem. Let

$$\mathbf{m}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^d$$

be a map that represents the mathematical relationship between the input to a computer method **m** to its output. For instance

- *addition* then $\mathbf{m}(\mathbf{x}) = x_1 + x_2$
- *subtraction* then $\mathbf{m}(\mathbf{x}) = x_1 - x_2$
- *variance* then $\mathbf{m}(\mathbf{x}) = \sum_i (x_i - 1/N \sum_j x_j))^2$
- *linear system* then $\mathbf{m}(\mathbf{b}) = \boldsymbol{A}^{-1}\mathbf{x}$

Note that, crucially, $\mathbf{m}(\mathbf{x})$ represents the mathematical function we are trying to compute and this has no aspect of the computer implementation.

Quiz What is the map **m** for least squares?

Once we have these functions, then we can study their relative sensitivity.

Definition 18.1
The relative condition number of a map **m** relates the relative change of the output with respect to the relative change of the input. For a specific change, **d**, the value is

$$\kappa(\mathbf{x}, \mathbf{d}; \mathbf{m}) = \frac{\|\mathbf{m}(\mathbf{x}+\mathbf{d}) - \mathbf{m}(\mathbf{x})\| / \|\mathbf{m}(\mathbf{x})\|}{\|\mathbf{d}\| / \|\mathbf{x}\|}.$$

For the worst case on a differentiable function as $\|\mathbf{d}\| \to 0$, we have[3]

$$\kappa(\mathbf{x}; \mathbf{m}) = \frac{\|(\nabla \mathbf{m})(\mathbf{x})\|}{\|\mathbf{m}(\mathbf{x})\|} \|\mathbf{x}\|.$$

We need to be slightly careful with the choice of norm when we do these analyses in terms of the dimension of the Jacobian matrix.

— TODO – Expand on this with Gautschi's notes too.

EXAMPLE 18.2 What is the condition number of the simple act of multiplying two numbers? The map $m(x, y) = xy$. ◆

18.4 THE CONDITION NUMBER OF A LINEAR SYSTEM

For the map $\mathbf{m}(\mathbf{b}) = \boldsymbol{A}^{-1}\mathbf{b}$ we have:

$$\nabla \mathbf{m} = \boldsymbol{A}^{-1} \quad \text{and} \quad \boldsymbol{A}\mathbf{x} = \mathbf{b}$$

in which case we have

$$\kappa(\mathbf{b}; \mathbf{m}) = \|\boldsymbol{A}^{-1}\| \frac{\|\mathbf{b}\|}{\|\boldsymbol{A}^{-1}\mathbf{b}\|}$$

but in terms of the solution **x** we have:

$$\kappa(\mathbf{b}; \mathbf{m}) = \|\boldsymbol{A}^{-1}\| \frac{\|\boldsymbol{A}\mathbf{x}\|}{\|\mathbf{x}\|}.$$

Note that for any vector **x** and an operator induced norm, we have:

$$\frac{\|\boldsymbol{A}\mathbf{x}\|}{\|\mathbf{x}\|} \le \|\boldsymbol{A}\|$$

by the definition of an operator-induced norm. Thus

$$\kappa(\mathbf{b}; \mathbf{m}) \le \|\boldsymbol{A}\| \|\boldsymbol{A}^{-1}\|.$$

The condition number of solving a linear system arises so often that we give it a special name.

[3] Note that the Jacobian is $\lim \mathbf{d} \to 0 \|\mathbf{m}(\mathbf{x}+\mathbf{d}) - \mathbf{m}(\mathbf{x})\| \, \|\mathbf{d}\|$.

Definition 18.3 (condition number of a matrix)
The condition number of a matrix is an upper bound on the condition number of solving a linear system of equations for a general solution **x** and a general right hand side **b**. We write

$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\| \, \|\boldsymbol{A}^{-1}\|$$

which gives the condition number for any operator induced matrix norm. The choice of norm is typically the 2-norm unless otherwise specified.

19 CONDITIONING OF LEAST SQUARES & THE PSEUDOINVERSE

Consider a least squares problem with a full rank $\boldsymbol{A}$

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\| \quad .$$

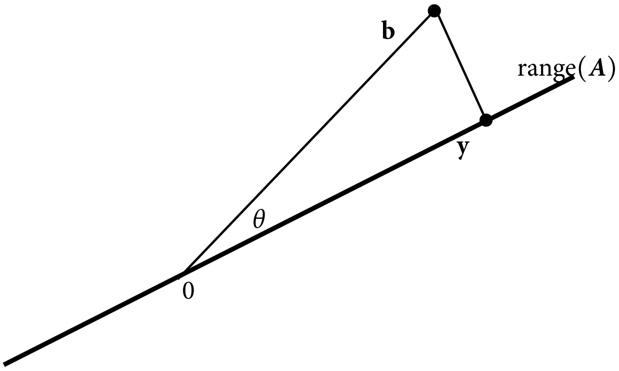
There are two natural maps to consider:

$$\text{a map from } \mathbf{b} \text{ to } \mathbf{x} \quad \mathbf{x} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\mathbf{b}$$

$$\text{a map from } \mathbf{b} \text{ to } \boldsymbol{A}\mathbf{x} \quad \mathbf{y} = \boldsymbol{A}(\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\mathbf{b}.$$

The vector $\mathbf{y}$ is the vector of least squares predictions. Even if $\mathbf{x}$ might be sensitive, it's possible that $\mathbf{y}$ may be substantially less sensitive.

There is a nice geometric discussion of how to analyze conditioning for this problem in Trefethen and Bau.[1].



A least squares problem involves the interaction of $\mathbf{b}$ with range($\boldsymbol{A}$). The angle between the two is θ. If θ is zero, then $\mathbf{b}$ is in the range of $\boldsymbol{A}$ and the residual of the least squares problem is 0.

We can easily compute $\cos\theta$ as the ratio $\|\mathbf{y}\|/\|\mathbf{b}\|$. This quantity will play a role in the conditioning. If $\theta = 0$, then $\cos\theta = 1$, whereas if $\theta = \pm\pi/2$, then $\cos\theta = 0$.

19.1 THE PSEUDOINVERSE.

It's easiest to analyze this problem by establishing a single quantity for $(\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T$. This is the pseudoinverse.

$$\boldsymbol{A}^+ = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T.$$

[1] Trefethen and Bau, Lecture 18, Theorem 18.1

However, we will write this in terms of the SVD of $\boldsymbol{A}$. Let $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$. Then

$$(\boldsymbol{A}^T\boldsymbol{A})^{-1} = (\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}\boldsymbol{V}^T)^{-1} = \boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{V}^T$$

$$\boldsymbol{A}^+ = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T = \boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{V}^T\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T = \boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{\Sigma}^T\boldsymbol{U}^T.$$

The term $(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{\Sigma}^T$ simplifies greatly. First, note that all matrices involved are diagonal. Since we are looking at full rank least squares problems, then $m \geq n$ and all $\sigma_i \neq 0$. Thus:

$$(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1} \text{ is } n \times n \text{, diagonal } = \begin{cases} \frac{1}{\sigma_i^2} & \text{on diagonal} \\ 0 & \text{otherwise.} \end{cases}$$

Hence, the entire matrix is:

$$(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{\Sigma}^T = \; n \times m \text{, diagonal } = \begin{cases} \frac{1}{\sigma_i} & \text{on diagonal} \\ 0 & \text{otherwise.} \end{cases}$$

This defines the pseudo-inverse of a diagonal matrix, which we often write:

$$\boldsymbol{\Sigma}^+ = (\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{\Sigma}^T.$$

To recap: the pseudoinverse of $\boldsymbol{A}$ is

$$\boldsymbol{A}^+ = \boldsymbol{V}\boldsymbol{\Sigma}^+\boldsymbol{U}^T.$$

19.2 THE CONDITION NUMBER OF LEAST SQUARES IN TERMS OF THE PSEUDOINVERSE.

Let $\mathbf{x} = \boldsymbol{A}^+\mathbf{b}$. Then the condition number of the least squares vector $\mathbf{x}$ in terms of $\mathbf{b}$ is:

$$\kappa(\mathbf{b}) = \frac{\|\boldsymbol{A}^+\|\,\|\mathbf{b}\|}{\|\mathbf{x}\|} = \|\boldsymbol{A}^+\|\frac{\|\mathbf{b}\|}{\|\mathbf{y}\|}\frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \frac{\|\boldsymbol{A}^+\|}{\cos\theta}\frac{\|\mathbf{y}\|}{\|\mathbf{x}\|}.$$

This second term $\frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \frac{\|\boldsymbol{A}\mathbf{x}\|}{\|\mathbf{x}\|}$ measures how large $\mathbf{y}$ is compared with $\mathbf{x}$. Clearly, this is less than $\|\boldsymbol{A}\|$. So we have

$$\kappa(\mathbf{b}) \leq \|\boldsymbol{A}^+\|\,\|\boldsymbol{A}\|\frac{1}{\cos\theta}.$$

However, if we let $\frac{1}{\eta} = \frac{\|\boldsymbol{A}\mathbf{x}\|}{\|\boldsymbol{A}\|\,\|\mathbf{x}\|}$, then $\frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \|\boldsymbol{A}\|\frac{1}{\eta}$. Consequently, we have

$$\kappa(\mathbf{b}) = \|\boldsymbol{A}^+\|\,\|\boldsymbol{A}\|\frac{1}{\eta\cos\theta}.$$

Because the term $\|\boldsymbol{A}^+\|\,\|\boldsymbol{A}\|$ arises just as frequently as $\|\boldsymbol{A}^{-1}\|\,\|\boldsymbol{A}\|$, we extend the definition of $\kappa(\boldsymbol{A})$ with a rectangular matrix $\boldsymbol{A}$.

19.3 THE CONDITION NUMBER OF THE LEAST SQUARES PREDICTION.

Let $\mathbf{y} = \boldsymbol{A}\boldsymbol{A}^{+}\mathbf{b}$. Then

$$\mathbf{y} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{T}\boldsymbol{V}\boldsymbol{\Sigma}^{+}\boldsymbol{U}^{T}\mathbf{b} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{+}\boldsymbol{U}^{T}\mathbf{b}.$$

The matrix $\boldsymbol{A}\boldsymbol{A}^{+}$ is an orthogonal projection onto range$(\boldsymbol{A})$. What this means for us is that $\|\boldsymbol{A}\boldsymbol{A}^{+}\| = 1$ for the operator induced 2-norm. Consequently

$$\kappa(\mathbf{b}) = \|\boldsymbol{A}\boldsymbol{A}^{+}\| \frac{\|\mathbf{b}\|}{\|\mathbf{y}\|} = \frac{1}{\cos\theta}.$$

The condition number of an eigenvalue ———————

Relationship with Freschet derivative ——————————————–

For the full-rank least squares system, the mathematical map from

One-pass algorithms for variance ————————————–

See online codes for this.

BACKWARDS STABILITY 20

There are two aspects to numerical accuracy.[1] The first aspect is the conditioning of a problem taht we addressed above. The second aspect is the stability of the algorithm to compute it. This lecture is about algorithmic stability, not about the conditioning of a problem. Take the most trivial function $f(x) = x$. The following algorithm:

```
for i=1 to 60
  x = $\sqrt{x}$

for i=1 to 60
  x = $x^2$
```

computes the this identity function for $x \geq 0$. Yet, if you run this algorithm on a computer, you will compute the function:

$$\tilde{f}(x) = \begin{cases} 0 & 0 \leq x < 1 \\ 1 & x \geq 1. \end{cases}$$

What this shows is that this *rather silly* algorithm is not a good idea.

Let's make this idea precise and develop a definition that would allow us to make a more precise statement. Let $\tilde{y} \approx f(x)$ and let $y = f(x)$. Ideally, we'd like the

$$\text{absolute error} = |\tilde{y} - y|$$

to be small. But if y is large, this isn't reasonble. So a better goal would be ensure the

$$\text{relative error} = |\tilde{y} - y|/|y|$$

is small. The problem with this type of error analysis is that we immediately encounter the conditioning of the underyling problem. That is, if the problem is ill-conditioned, we will not be able to show that the relative error is always small. This is independent of any algorithm that we have. What has proven to be useful instead is the idea of *backwards error analysis*. That is, we ask the question:

$$\text{is } \tilde{y} \text{ the } \textit{exact} \text{ computation of some } x + \delta?$$

Formally, can we show that an algorithm will have a small δ such that

$$\tilde{y} = f(x + \delta)?$$

If so, then we use this property to establish a relative error bound via the condition number argument:

$$|\tilde{y} - y|/|y| \text{ is something like } \kappa(y)|\delta|.$$

[1] Most of these notes and ideas are taken from Trefethen and Bau, Numerical Linear Algebra, Lecture 14 and Higham, Accuracy and Stability of Numerical Algorithms, Chapter 1.

Let's put this slightly more formally now.

Definition 20.1
An algorithm is *backwards stable* for computing $y = f(x)$ if it computes $\tilde{y} = f(x + \delta)$ with $|\delta|/|x| \leq Cu$ where C is a constant and u is the machine precision.[2]

For a matrix problem $f(\mathbf{x})$, we apply this as:

$$\tilde{\mathbf{y}} = f(\mathbf{x} + \mathbf{d}) \text{ where } \|\mathbf{d}\| / \|\mathbf{x}\| \leq C_n u.$$

In this case, C_n may depend on the dimension n of the matrix.

One question immediately presents itself: Are floating point operations backwards stable? We defined

$$\mathrm{fl}(x + y) = (x + y)(1 + \delta) \text{ for } |\delta| \leq u.$$

But we can just move the $(1 + \delta)$ inside and we have:

$$\mathrm{fl}(x + y) = x(1 + \delta) + y(1 + \delta).$$

Thus, we are computing the exact addition for a problem whose input is perturbed by $(1 + \delta)$.

Now let's see how this analysis plays out in other cases as well.

[2] Throughout this note, u is the machine precision.

21 BACKWARDS STABILITY OF LU DECOMPOSITION

These notes were copied from Gene Golub's CME 302 Matrix Computations class while David was a student at Stanford.

Warning We will adopt slightly non-standard notation. The quantity $\boldsymbol{\Delta}$ is a matrix, as are any symbols prefixed with δ, such as $\delta \boldsymbol{L}$ or $\delta \boldsymbol{U}$. Upper tildes, like $\tilde{\boldsymbol{L}}$ will denote quantities represented on the computer.

Our goal is to show that Gaussian Elimination is a backwards stable algorithm. If we show that Gaussian elimination is backwards stable, then we will show that we can compute $\tilde{\mathbf{x}}$ such that

$$(\boldsymbol{A} + \boldsymbol{\Delta})\tilde{\mathbf{x}} = \mathbf{b}.$$

In words, we compute the solution to a perturbed system where the perturbation is the matrix $\boldsymbol{\Delta}$. If so, then, if $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, we have:[1]

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \frac{\rho\kappa(\boldsymbol{A})}{1 - \rho\kappa(\boldsymbol{A})}$$

where $\rho = \|\boldsymbol{\Delta}\| / \|\boldsymbol{A}\|$.

Thus, if we can show that $\|\boldsymbol{\Delta}\|$ is small, we will have a nice bound on the error of the solution.

FRAMEWORK

This is a complicated operation. When we solve $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ with Gaussian elimination, we have three steps:

1. Factoring $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$. We will show that we find: $\tilde{\boldsymbol{L}}\tilde{\boldsymbol{U}} = \boldsymbol{A} + \boldsymbol{E}$. We will assume that partial pivoting is used, although, we will assume the permutation is known up front.
2. Next we have to solve $\boldsymbol{L}\mathbf{y} = \mathbf{b}$, or on the computer,

   $$(\tilde{\boldsymbol{L}} + \delta \boldsymbol{L})\underbrace{(\mathbf{y} + \delta \mathbf{y})}_{=\tilde{\mathbf{y}}} = \mathbf{b}.$$

   In other words, we again solve a perturbed system exactly.
3. Finally, we have to solve $\boldsymbol{U}\mathbf{x} = \mathbf{y}$. But on the computer, this is now:

   $$(\tilde{\boldsymbol{U}} + \delta \boldsymbol{U})(\mathbf{x} + \delta \mathbf{x}) = \tilde{\mathbf{y}} = \mathbf{y} + \delta \mathbf{y}.$$

Together, these results show that

$$\mathbf{b} = (\tilde{\boldsymbol{L}} + \delta \boldsymbol{L})(\tilde{\boldsymbol{U}} + \delta \boldsymbol{U})(\mathbf{x} + \delta \mathbf{x}).$$

But, we have:

$$(\tilde{\boldsymbol{L}}+\delta \boldsymbol{L})(\tilde{\boldsymbol{U}}+\delta \boldsymbol{U}) = \tilde{\boldsymbol{L}}\tilde{\boldsymbol{U}}+\delta \boldsymbol{L}\tilde{\boldsymbol{U}}+\tilde{\boldsymbol{L}}\delta \boldsymbol{U}+\delta \boldsymbol{L}\delta \boldsymbol{U} = \boldsymbol{A}+\boldsymbol{E}+\delta \boldsymbol{L}\tilde{\boldsymbol{U}}+\tilde{\boldsymbol{L}}\delta \boldsymbol{U}+\delta \boldsymbol{L}\delta \boldsymbol{U}.$$

[1] It's a good exercise to stitch this bound together from $\tilde{\mathbf{x}} + \boldsymbol{A}^{-1}\boldsymbol{\Delta}\tilde{\mathbf{x}} = \mathbf{x}$.

Thus,

$$(\boldsymbol{A} + \boldsymbol{\Delta})(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b}$$

with $\boldsymbol{\Delta} = \boldsymbol{E} + \delta\boldsymbol{L}\tilde{\boldsymbol{U}} + \tilde{\boldsymbol{L}}\delta\boldsymbol{U} + \delta\boldsymbol{L}\delta\boldsymbol{U}$.

Now we go through and find bounds on all of these terms.

ERRORS IN LU

Our goal now is to show that what we compute on the computer is: $\tilde{\boldsymbol{L}}\tilde{\boldsymbol{U}} = \boldsymbol{A} + \boldsymbol{E}$ for some $\boldsymbol{E}$.

We'll show this in two steps. In the first step, we'll just introduce additive errors into each of our operations. In the second step, we'll use the properties of floating point arithmetic to bound those errors.

Gaussian Elimination with Errors

Suppose we are computing the LU factorization of $\boldsymbol{A}$. We'll represent this as a sequence of changes to the matrix

$$\boldsymbol{A} = \boldsymbol{A}^{(1)} \quad \xrightarrow{\text{zero 1st column}} \quad \boldsymbol{A}^{(2)} \quad \xrightarrow{\text{zero 2st column}} \quad \boldsymbol{A}^{(3)} \quad \to \quad \cdots \quad \to \quad \boldsymbol{A}^{(n-1)}.$$

Thus, $\boldsymbol{A}^{(k)}$ is the matrix after $k-1$ columns have been zeroed. To move to the k+1st step, we compute:

$$A_{ij}^{(k+1)} = A_{ij}^{(k)} - L_{ik}A_{kj}^{(k)}, \quad L_{ik} = \frac{A_{ik}^{(k)}}{A_{kk}^{(k)}}.$$

Let $\boldsymbol{B}^{(k)}$ be the matrix after $k-1$ columns have been zeroed in floating point arithmetic. We have:

$$B_{ij}^{(k+1)} = B_{ij}^{(k)} - \tilde{L}_{ik}B_{kj}^{(k)} + \mu_{i,j}^{(k+1)}, \quad \tilde{L}_{ik} = \frac{B_{ik}^{(k)}}{B_{kk}^{(k)}}(1 + \eta_{ik}).$$

In these expressions, η is a standard floating point guarantee, but $\mu_{i,j}^{(k+1)}$ represents the simple floating point error in computing $B_{ij}^{(k+1)}$ from the intermediate terms. Thus, μ is a pure difference that we will quantify in terms of floating point operations later. Also, for this expression, note that μ does not need to include the effect from η because we are analyzing this expression with $\tilde{L}$ – the computed quantity, not the exact quantity.

For each element B_{ij} there is a maximum k such that we will stop looking at that element in the future.[2] Thus, when we stop looking at an element B_{ij} there are two reasons: 1) it's in the upper triangle and $i \le k$, or 2) it's zero in the lower-triangle with $j < k$.

So we'll divide our analysis into two cases that correspond to these two

[2] This is a straightforward observation if you look at LU in exact arithmetic:

$$\boldsymbol{A}^{(k)} = \begin{bmatrix} \boldsymbol{U} & \boldsymbol{C} \\ 0 & \boldsymbol{D} \end{bmatrix},$$

where $\boldsymbol{U}$ is $(k-1)$-by-$(k-1)$.

outcomes. First, suppose we are in the upper-triangle, so $j \geq i$. Then,

$$\begin{aligned} B_{ij}^{(2)} &= B_{ij}^{(1)} - \tilde{L}_{i,1} B_{1,j}^{(1)} + \mu_{ij}^{(2)} \\ B_{ij}^{(3)} &= B_{ij}^{(2)} - \tilde{L}_{i,2} B_{2,j}^{(2)} + \mu_{ij}^{(3)} \\ &\dots \\ B_{ij}^{(i)} &= B_{ij}^{(i-1)} - \tilde{L}_{i,i-1} B_{i,j}^{(i-1)} + \mu_{ij}^{(i)}. \end{aligned}$$

The goal here is a relationship between $B_{ij}^{(1)}$ and $B_{ij}^{(i)}$. If we sum up all of these expressions, we have:

$$\sum_{k=2}^{i} B_{ij}^{(k)} = \sum_{k=1}^{i-1} B_{ij}^{(k)} - \sum_{k=1}^{i-1} \tilde{L}_{i,k} B_{k,j}^{(k)} + \sum_{k=2}^{i} \mu_{ij}^{(k)}.$$

Note that this sum telescopes! In other words, we get massive cancellation of the $\boldsymbol{B}_{ij}^{(k)}$ terms. After all of them are removed, we have:

$$B_{ij}^{(i)} = B_{ij}^{(1)} - \sum_{k=1}^{i-1} \tilde{L}_{i,k} B_{k,j}^{(k)} + \sum_{k=2}^{i} \mu_{ij}^{(k)}.$$

We can rearrange this to show that:

$$B_{ij}^{(1)} + E_{ij} = B_{ij}^{(i)} + \sum_{k=1}^{i-1} \tilde{L}_{i,k} B_{k,j}^{(k)}$$

where $E_{ij} = \sum_{k=2}^{i} \mu_{ij}^{(k)}$.

We are half done with showing the error in the LU factorization. At this point, we've shown that the upper-triangular piece of our factorization is correct for a matrix $\boldsymbol{A} + \boldsymbol{E}$, with a precise accounting of where the errors occur. Now, we just have to show that same thing holds in the lower-triangular region.

If $i > j$, then

$$\begin{aligned} B_{ij}^{(2)} &= B_{ij}^{(1)} - \tilde{L}_{i,1} B_{1,j}^{(1)} + \mu_{ij}^{(2)} \\ B_{ij}^{(3)} &= B_{ij}^{(2)} - \tilde{L}_{i,2} B_{2,j}^{(2)} + \mu_{ij}^{(3)} \\ &\dots \\ B_{ij}^{(j)} &= B_{ij}^{(j-1)} - \tilde{L}_{i,j-1} B_{j-1,j}^{(j-1)} + \mu_{ij}^{(j)}. \end{aligned}$$

This is, of course, the same.[3] But we also have:

$$0 = B_{ij}^{(j)} - \tilde{L}_{i,j} B_{jj}^{(j)} + \mu_{ij}^{(j+1)}$$

because B_{ij} becomes 0 in the $(j+1)$st step. After a similar cancellation of terms, we get:

$$B_{ij}^{(1)} = 0 + \sum_{k=1}^{j} \tilde{L}_{i,k} B_{k,j}^{(k)} + E_{ij}$$

where $E_{ij} = \sum_{k=2}^{j+1} \mu_{ij}^{(k)}$.

[3] I think there might be an index mistake in here, be wary.

Thus, what we compute on the computer is:

$$\tilde{\boldsymbol{L}}\tilde{\boldsymbol{U}} = \begin{bmatrix} 1 & & & & \\ \tilde{L}_{2,1} & 1 & & & \\ \tilde{L}_{3,1} & \tilde{L}_{3,2} & 1 & & \\ \vdots & \vdots & & \ddots & \\ \tilde{L}_{n,1} & \tilde{L}_{n,2} & \cdots & \cdots & 1 \end{bmatrix} \begin{bmatrix} B_{1,1}^{(1)} & B_{1,2}^{(1)} & B_{1,3}^{(1)} & \cdots & B_{1,n}^{(1)} \\ & B_{2,2}^{(2)} & B_{2,3}^{(2)} & \cdots & B_{2,n}^{(2)} \\ & & B_{3,3}^{(3)} & & \vdots \\ & & & \ddots & \vdots \\ & & & & B_{n,n}^{(n)} \end{bmatrix}.$$

Using our equations that we derived, we can show:

$$\tilde{\boldsymbol{L}}\tilde{\boldsymbol{U}} = \boldsymbol{B}^{(1)} + \boldsymbol{E} = \boldsymbol{A} + \boldsymbol{E}.$$

Bounding the errors

Now, we need to bound each element in $\boldsymbol{E}$. We have:

$$\tilde{L}_{i,k} = \mathrm{fl}\left(B_{ik}^{(k)}/B_{kk}^{(k)}\right) = (B_{ik}^{(k)}/B_{kk}^{(k)})(1 + \eta_{ik})$$

and

$$\mathrm{fl}\left(\tilde{L}_{i,k}B_{k,j}^{(k)}\right) = (\tilde{L}_{i,k}B_{k,j}^{(k)})(1 + \theta_{ij}^{(k)}),$$

so that:

$$B_{ij}^{(k+1)} = \mathrm{fl}\left(B_{ij}^{(k)} - (\tilde{L}_{i,k}B_{k,j}^{(k)})(1 + \theta_{ij}^{(k)})\right) = \left(B_{ij}^{(k)} - (\tilde{L}_{i,k}B_{k,j}^{(k)})(1 + \theta_{ij}^{(k)})\right)(1+\phi_{ij}^{(k)}).$$

The quantities η, θ, and ϕ all obey $|\cdot| \leq u$, the machine round-off error. By reworking this bound for a while, we get:

$$\mu_{ij}^{(k+1)} = B_{ij}^{(k+1)}\left(\frac{\phi_{ij}^{(k)}}{1 + \phi_{ij}^{(k)}}\right) - \tilde{L}_{i,k}B_{k,j}^{(k)}\theta_{ij}^{(k)}.$$

Using the bound $|\tilde{L}_{ij}| \leq 1$ from using partial pivoting, we find:

$$|\mu_{ij}^{(k+1)}| \leq |B_{ij}^{(k+1)}|\frac{u}{1-u} + |B_{kj}^{(k)}|u.$$

We are getting close to a bound. We now need to understand how big elements in $\boldsymbol{B}$ can get! Here, we'll use exact computation again. Let $|A_{ij}| \leq a$ for all i, j. Using

$$A_{ij}^{(k+1)} = A_{ij}^{(k)} - L_{ik}A_{kj}^{(k)}, \quad L_{ik} = \frac{A_{ik}^{(k)}}{A_{kk}^{(k)}}$$

we find

$$\begin{aligned} |A_{ij}^{(2)}| &\leq |A_{ij}^{(1)}| + |A_{kj}^{(1)}| \leq 2a \\ |A_{ij}^{(3)}| &\leq |A_{ij}^{(2)}| + |A_{kj}^{(2)}| \leq 4a \\ &\vdots \\ |A_{ij}^{(n)}| &\leq 2^{n-1}a. \end{aligned}$$

We'll return to this bound in a second. It's pretty absurd.

[4] Recall that $\frac{1}{1-u} = 1 + u + u^2 + \ldots$.

Instead, let's bound $|B_{ij}^{(k)}| \leq Ga$ where G is called the *growth factor*. The result above suggests that the growth factor is 2^{n-1}. But let's just go ahead and use G. In that case, we get: [4]

$$|\mu_{ij}^{(k+1)}| \leq Ga\frac{u}{1-u}+Gau = Ga(2u-u^2)(1+u+u^2+\ldots) \approx 2uGa+O(u^2).$$

This is, finally, some progress. We now have a bound on all the terms μ_{ij} in the summation formulas that define the matrix E. Recall that

$$E_{ij} = \sum_{k=2}^{i} \mu_{ij}^{(k)}$$

if $j \geq i$. Thus,

$$|E_{ij}| \leq (i-1)2uGa$$

for any element in the upper triangular region. For elements in the lower-triangular region

$$|E_{ij}| = \left|\sum_{k=2}^{j+1} \mu_{ij}^{(k)}\right| \leq j2uGa.$$

We can summarize this analysis via a matrix equation:

$$|\boldsymbol{E}| \leq 2uGa \begin{bmatrix} 0 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & 1 & 1 & \cdots & \cdots & 1 \\ 1 & 2 & 2 & \cdots & \cdots & 2 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & n-2 & n-2 \\ 1 & 2 & 3 & \cdots & n-1 & n-1 \end{bmatrix} + O(u^2).$$

Thus, $\boldsymbol{A} + \boldsymbol{E} = \tilde{\boldsymbol{L}}\tilde{\boldsymbol{U}}$.

And we're done with part 1.

Growth factors

What we showed is that $|E_{ij}| \leq 2uGa$ where $G \leq 2^{n-1}$. That is not exactly small. And it can occur! The matrix

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}$$

has an LU factorization with $U_{n,n} = 2^{n-1}$. This is entirely general.

While this exponential explosion in the growth factor *can* occur. It never seems to occur naturally. It only arises in a few examples that are designed to elicit it. This has provoked much study of *why* this occurs.

Sankar, Spielman, and Teng recently took up this issue. Their paper "Smoothed Analysis of the Condition Numbers and Growth Factors of Matrices" (SIMAX 2006) shows some remarkable new results about the growth factor of a random perturbation of a matrix. Here's the abstract.

> Let $\hat{\boldsymbol{A}}$ be an arbitrary matrix and let $\boldsymbol{A}$ be a slight random perturbation of $\hat{\boldsymbol{A}}$. We prove that it is unlikely that $\boldsymbol{A}$ has a large condition number. Using this result, we prove that it is unlikely that $\boldsymbol{A}$ has large growth factor under Gaussian elimination without pivoting. By combining these results, we show that the smoothed precision necessary to solve $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, for any $\mathbf{b}$, using Gaussian elimination without pivoting is logarithmic. Moreover, when $\hat{\boldsymbol{A}}$ is an all-zero square matrix, our results significantly improve the average-case analysis of Gaussian elimination without pivoting performed by Yeung and Chan (SIAM J. Matrix Anal. Appl., 18 (1997), pp. 499-517).

Errors in forward-and-back-substitution

Here, we'll consider the problem: [5]

$$\boldsymbol{T}\mathbf{v} = \mathbf{h}$$

where $\boldsymbol{T}$ is lower-triangular, $n \times n$, non-singular. Element-wise, we find:

$$\begin{bmatrix} T_{11} & & \\ \vdots & \ddots & \\ T_{n1} & \cdots & T_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} h_1 \\ \vdots \\ h_n \end{bmatrix}.$$

Thus, we get the forward substitution procedure:

$$\begin{aligned} v_1 &= h_1 / T_{11} \\ &\vdots \\ v_k &= \frac{h_k - T_{k,1}v_1 - T_{k,2}v_2 - \cdots - T_{k,k-1}v_{k-1}}{T_{kk}}. \end{aligned}$$

Let $\tilde{v}_k$ be the computed value in floating point. In floating point, this gives us:

$$\begin{aligned} \tilde{v}_k &= \left(\frac{\left(h_k - \sum_{i=1}^{k-1} T_{k,i}\tilde{v}_i(1+\omega_{k,i})\right)(1+\alpha_k)}{T_{k,k}} \right)(1+\tau_k) \\ &= \frac{h_k - \sum_{i=1}^{k-1} T_{k,i}\tilde{v}_i(1+\omega_{k,i})}{T_{k,k}/(1+\alpha_k)(1+\tau_k)}. \end{aligned}$$

With some more *coffee-shop manipulations*, we arrive at:

$$\sum_{i=1}^{k} T_{k,i}\tilde{v}_i(1+\lambda_{k,i}) = h_k$$

or

$$\boldsymbol{T}\tilde{\mathbf{v}} + \begin{bmatrix} \lambda_{1,1}T_{1,1} & & \\ \lambda_{2,1}T_{2,1} & \lambda_{2,1}T_{2,1} & \\ \vdots & \vdots & \ddots \end{bmatrix} \tilde{\mathbf{v}} = \mathbf{h}.$$

Equivalently,[6]

$$(\boldsymbol{T} + \delta\boldsymbol{T})\mathbf{v} = \mathbf{h}$$

[5] I'm slightly less confident in the notes for this section, reader beware; Trefethen, Lecture 17 and Golub and van Loan Section 3.1 have this analysis.

[6] This matrix can take a few different forms depending on how the summation is evaluated.

where

$$|\delta \boldsymbol{T}| \leq u \begin{bmatrix} |T_{1,1}| & & & & \\ |T_{2,1}| & 2|T_{2,2}| & & & \\ 2|T_{3,1}| & |T_{3,2}| & 3|T_{3,3}| & & \\ \vdots & \vdots & & \ddots & \\ (n-1)|T_{n,1}| & \cdots & \cdots & |T_{n,n-1}| & n|T_{n,n}| \end{bmatrix} + O(u^2)$$

Thus, for solving a triangular system, we have errors:

$$(\boldsymbol{T} + \delta \boldsymbol{T})\mathbf{v} = \mathbf{h} \text{ where } |\delta T_{ij}| \leq nut + O(u^2) \text{ and } |T_{ij}| \leq t.$$

Overall errors

In our framework, the solution $\tilde{\mathbf{x}} = \mathbf{x} + \delta x$ satisfies:

$$(\boldsymbol{A} + \boldsymbol{\Delta})\mathbf{x} = \mathbf{b}$$

where $\boldsymbol{\Delta} = \boldsymbol{E} + \delta \boldsymbol{L}\tilde{\boldsymbol{U}} + \tilde{\boldsymbol{L}}\delta \boldsymbol{U} + \delta \boldsymbol{L}\delta \boldsymbol{U}$. Let's look at the maximum errors in each of these terms:

$$\max_{ij} |\delta \tilde{L}_{ij}| \leq nu + O(u^2)$$

$$\max_{ij} |\delta \tilde{U}_{ij}| \leq nuGa + O(u^2).$$

This tells us:

$$\max_{ij} |Delta_{ij}| \leq \max_{ij} |E_{ij}| + \max_{ij} |(\delta \boldsymbol{L}\tilde{\boldsymbol{U}})_{ij}| + \max_{ij} |(\tilde{\boldsymbol{L}}\delta \boldsymbol{U})_{ij}| + \max_{ij} |(\delta \boldsymbol{L}\delta \boldsymbol{U})_{ij}|$$

$$\leq 2uGan + n^2Gau + n^2Gau + O(u^2).$$

Thus,

$$\|\boldsymbol{\Delta}\|_\infty \leq 2n^2(n+1)uGa.$$

This merits a theorem.

Theorem 21.1
Gaussian Elimination is Backwards Stable!

Now, we can go even further, and bound the error in the solution as well. Let $\rho = \frac{\|\Delta\|}{\|A\|}$. Then

$$\rho \geq 2n^2(n+1)G$$

because $\|\boldsymbol{A}\|_\infty \geq a$. Going back to the beginning, we now have:

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\rho\kappa(\boldsymbol{A})}{1 - \rho\kappa(\boldsymbol{A})} = \frac{2n^2(n+1)G\kappa(\boldsymbol{A})}{1 - 2n^2(n+1)G\kappa(\boldsymbol{A})}$$

EXERCISES

1. Here is a matrix that achieves the worst-case growth factor for LU

$$\boldsymbol{A} = \begin{bmatrix} -1 & 0 & 0 & \cdots & -1 \\ 1 & -1 & 0 & \cdots & -1 \\ 1 & 1 & -1 & \cdots & -1 \\ \vdots & \vdots & \cdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 & -1 \end{bmatrix} = \boldsymbol{T} - 2\boldsymbol{I} - \mathbf{v}\mathbf{e}_n^T$$

   where $\boldsymbol{T}$ is a lower-triangular matrix of all 1s and $\mathbf{v}$ is a vector with ones in all but the last element.

   (a) Develop an expression for the inverse of this matrix.

   (b) Develop an expression for $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{A}^{-T}\boldsymbol{A}^{-1}$.

   (c) Develop an expression or upper bound for $\kappa(A)$.

2.

UNSTABLE LU FACTORIZATIONS 21

Learning objectives

1. Recognize how to analyze a matrix where we see the worst case behavior for the LU factorization.
2. Explain how we can use the tools we have developed to analyze this matrix.

In Lecture 21, we showed that sequential variable elimination or LU factorizations are backwards stable methods. Except the worst case bound is bad as we saw with matrices of the form

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ -1 & 1 & 0 & \cdots & 0 & 1 \\ -1 & -1 & 1 & \cdots & 0 & 1 \\ -1 & -1 & -1 & \ddots & \ddots & 1 \\ -1 & -1 & -1 & \cdots & -1 & 1 \end{bmatrix}.$$

The goal of this lecture is to illustrate that we can analyze this matrix to make sure it isn't any of the other potential problems. It's also an example of how we can mix numerical computation with theoretical validation to do so. Our goal for the lecture is to compute an analytical form for the inverse and condition number.

21.0.1 *The inverse*

For a small matrix, the inverse has a suggestive pattern:

SUBSPACE METHODS V

The simple methods for solving linear systems and eigenvalue problems are all based on the idea of looking at matrix powers, i.e. methods that would have guarantees like

$$\mathbf{x}^{(k)} - \mathbf{x} \approx \boldsymbol{M}^k(\mathbf{x}^{(0)} - \mathbf{x})$$

for some matrix $\boldsymbol{M}^k \to 0$. We saw these for linear systems and eigenvalues.

The one exception was steepest descent (and Nesterov accelerated steepest descent), where we saw that

$$\mathbf{x}^{(k)} - \mathbf{x} = [\prod_{j=1}^{k}(\boldsymbol{I} - \alpha_k \boldsymbol{A})](\mathbf{x}^{(0)} - \mathbf{x})$$

where the coefficients α_k is adaptively chosen.

This expression $\prod_{j=1}^{k}(\boldsymbol{I} - \alpha_k \boldsymbol{M})$ gives a polynomial in a matrix $p_k(\boldsymbol{A})$. Right now, this isn't saying much. But the idea with subspace methods is to change these deterministic choices into completely adaptive choices.

For instance, rather than looking at *only* the steepest descent polynomial $p_k(\boldsymbol{A})$, what if we looked any *any* polynomial of up to powers of $\boldsymbol{A}^k$? Likewise, rather than looking at

THE MATRIX POWERS SUBSPACE, AKA THE KRYLOV SUBSPACE 22

22.1 MOTIVATION

Learning objectives

1. Recognize that the Neumann method for solving $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ can be explained in terms of subspaces and polynomials.
2. Understand that the Krylov subspace is a subspace of matrix powers: $\operatorname{span}(\mathbf{b}, \boldsymbol{A}\mathbf{b}, \boldsymbol{A}^2\mathbf{b}, \ldots)$
3. Recognize that this view suggests a more powerful approach to approximately solve a linear system of equations by searching the entire matrix power subspace

Recall the first method we saw to solve a linear system of equations:

$$\boldsymbol{A}\mathbf{x} = \mathbf{b}$$

where we conceptually multiplied by the inverse

$$(\boldsymbol{A})^{-1} \approx \boldsymbol{I} + (\boldsymbol{I} - \boldsymbol{A}) + (\boldsymbol{I} - \boldsymbol{A})^2 + \ldots$$

to get the algorithm:

$$\mathbf{x}^{(k)} = \sum_{j=0}^{k} (\boldsymbol{I} - \boldsymbol{A})^j \mathbf{b}.$$

Let's call this the Neumann-series algorithm for linear systems.

This converged as long as $\rho(\boldsymbol{I} - \boldsymbol{A}) < 1$. We could modify it so that it would work for any symmetric positive definite problem by incorporating a scaling that gave us the Richardson method.

The inspiration for our next set of methods arises from a set of subtle insights about this original method. This will yield a set of new perspectives that we will use to generate a family of solvers for linear systems called *Krylov methods*. In keeping with the idea of introducing names that refer to ideas instead of people, we also call this the power subspace methods.[1]

The following derivations are largely procedural. Essentially, we are seeking to find *generalizations* of some easy ideas that permit us to find new perspectives. We will then be able to use these new perspectives to identify particular methods. To study the methods, then, we'll take advantage of the perspective we used to derive it! This type of analysis can be subtle. So please do ask questions if you have trouble understanding why we are looking at something.

First, note that:

$$\mathbf{x}^{(k)} = \begin{bmatrix} \mathbf{b} & (\boldsymbol{I} - \boldsymbol{A})\mathbf{b} & \ldots & (\boldsymbol{I} - \boldsymbol{A})^k \mathbf{b} \end{bmatrix} \mathbf{e}.$$

That is, we can represent the kth iteration as a (simple!) linear combination of the basis vectors

$$(\mathbf{b}, (\boldsymbol{I} - \boldsymbol{A})\mathbf{b}, \ldots, (\boldsymbol{I} - \boldsymbol{A})^k \mathbf{b}.$$

This means that, for some vector $\mathbf{c}$, we can write:

$$\mathbf{x}^{(k)} = \begin{bmatrix} \mathbf{b} & \boldsymbol{A}\mathbf{b} & \boldsymbol{A}^2\mathbf{b} & \ldots \boldsymbol{A}^k \mathbf{b} \end{bmatrix} \mathbf{c}.$$

Let's work this out, which will give us a lead on our next perspective.

Lemma 22.1
Consider the kth iteration from a Neumann-series based approach, where $\mathbf{x}^{(k)} = \sum_{j=0}^{k} (\boldsymbol{I} - \boldsymbol{A})^j \mathbf{b}$. Then we can write $\mathbf{x}^{(k)} = \sum_{j=0}^{k} c_j \boldsymbol{A}^j \mathbf{b}$ for some coefficients $c_0, \ldots, c_k$.

[1] The ideas behind these methods were independently described around the same time by both Krylov and Lanczos.

PROOF The proof follows from the binomial expansion:

$$(\boldsymbol{I}-\boldsymbol{A})^k \mathbf{b} = \sum_{j=0}^{k} \binom{k}{j} (-\boldsymbol{A})^j \mathbf{b}.$$

But a more useful realization is as follows:

$$(\boldsymbol{I}-\boldsymbol{A})^k \mathbf{b} = \text{polynomial}(\boldsymbol{A})\mathbf{b}.$$

In which case, the theorem is just giving a change of basis between polynomials in powers of $(1-x)$ and x.[2] ■

Just to be clear, let's state the other result as well.

Corollary 22.2
Consider the kth iteration from a Neumann-series based approch, where $\mathbf{x}^{(k)} = \sum_{j=0}^{k} (\boldsymbol{I}-\boldsymbol{A})^j \mathbf{b}$, then $\mathbf{x}^{(k)} = p(\boldsymbol{A})\mathbf{b}$ for some polynomial $p(x) = \sum_{j=0}^{k} c_j x^j$.

22.1.1 *The basis for a polynomial*

What is a polynomial?[3] In our setting, we are only concerned with univariate polynomials. Consequently, a polynomial is any function of the form[4]

$$p(x) : \mathbb{S} \to \mathbb{S} \text{ where } p(x) = c_0 + c_1 x + c_2 x^2 + \cdots c_k x^k.$$

For the moment, think of $\mathbb{S} = \mathbb{R}$, the reals, which is where much of our intuition will come from. The degree of the polynomial is the highest power. So $p(x) = 5 + 2x + 3x^2$ is a degree 2 polynomial. The basis for a polynomial has to do with how we represent $p(x)$ as a sum of *functions of* x. For instance, we can introduce

$$f_0(x) = 1, f_1(x) = (1-x), f_2(x) = (1-x)^2 \text{ then}$$

$$p(x) = 3f_2(x) - 8f_1(x) + 0f_0(x).$$

The set of functions we use to write a polynomial is called the polynomial basis. Note that the actual function $p(x)$ is *independent* of the basis in which we write the functions.

Hence, what the previous lemma shows is simply that

$$p(x) = \sum_{j=0}^{k} \underbrace{f_j(x)}_{=(1-x)^j} = \sum_{j=0}^{k} s_j \underbrace{g_j(x)}_{=(x^j)}.$$

In this case, we need to produce coefficients s_j that correspond with the power, or monomial basis, $g_j(x) = x^j$.

[2] See the discussion section 22.1.1.

[3] Much more on polynomials will be discussed in a future chapter on Orthogonal Polynomials, lecture 25. Read more there now if you wish.

[4] We define this in terms of a general set $\mathbb{S}$, which could be a mathematical ring as we need only addition and multiplication to define a polynomial.

22.1.2 *Subspaces and Polynomials*

Consider $\mathbf{x}^k$ from the Neumann series

Subspaces

The subspace view is that

$$\mathbf{x}^{(k)} = \begin{bmatrix} \mathbf{b} & \boldsymbol{A}\mathbf{b} & \boldsymbol{A}^2\mathbf{b} & \cdots & \boldsymbol{A}^{k-1}\mathbf{b} \end{bmatrix} \mathbf{c}$$

to indicate that $\mathbf{x}^{(k)}$ is a specific linear combination of the basis vectors from the matrix powers subspace

$$\begin{bmatrix} \mathbf{b} & \boldsymbol{A}\mathbf{b} & \boldsymbol{A}^2\mathbf{b} & \cdots & \boldsymbol{A}^{k-1}\mathbf{b} \end{bmatrix}.$$

Polynomials

The polynomial view is that

$$\begin{aligned} \mathbf{x}^{(k)} &= \mathbf{b} + (\boldsymbol{I} - \boldsymbol{A})\mathbf{b} + \\ &\quad (\boldsymbol{I} - \boldsymbol{A})^2\mathbf{b} + \cdots + \\ &\quad (\boldsymbol{I} - \boldsymbol{A})^{k-1}\mathbf{b} \\ &= \text{poly}(\boldsymbol{A})\mathbf{b} \end{aligned}$$

where $\text{poly}(\boldsymbol{A}) \approx \boldsymbol{A}^{-1}$.

The key thing in both perspectives is that we can choose $\mathbf{c}$ to find *a different element of the matrix power subspace* or *a different polynomial* to find a better approximation of $\boldsymbol{A}^{-1}$. And also that these are *the same idea*!

The goal of our next set of methods, the

Krylov subspace methods

is to seek better vectors in these subspaces than the choice of the Neumann series. Equivalently, we can think of these as finding a better polynomial to represent $\boldsymbol{A}^{-1}$.

22.2 THE MATRIX POWERS SUBSPACE

The matrix powers subspace is the set of vectors

$$\mathbb{K}_k(\boldsymbol{A}, \mathbf{b}) = \text{span}(\mathbf{b}, \boldsymbol{A}\mathbf{b}, \boldsymbol{A}^2\mathbf{b}, \ldots \boldsymbol{A}^k\mathbf{b}).$$

This is typically called the *Krylov subspace*. Hence, the Neumann method just uses a specific element of $\mathbb{K}_k(\boldsymbol{A}, \mathbf{b})$ to approximation the solution of the linear system.

There is nothing "magic" about the Krylov subspace. Although, it does arise surprisingly often and in a number of forms.

Let's start with a simple theorem (with a slightly magic proof).

Theorem 22.3
Let $\boldsymbol{A}$ be full rank. Suppose that $\boldsymbol{A}^k\mathbf{b} \in \mathbb{K}_{k-1}(\boldsymbol{A}, \mathbf{b})$. Then the solution of $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ is contained within $\mathbb{K}_{k-1}(\boldsymbol{A}, \mathbf{b})$ as well.

PROOF Let $\boldsymbol{X}$ be any basis for $\mathbb{K}_{k-1}(\boldsymbol{A}, \mathbf{b})$. Then we have that $\boldsymbol{A}^k\mathbf{b} = \boldsymbol{X}\mathbf{y}$ for some vector $\mathbf{y}_k$. Consequently, we also have that $\boldsymbol{A}^{k+1}\mathbf{b} = \boldsymbol{X}\mathbf{y}_{k+1}$. Hence, for any set of powers beyond k, they exist in the basis $\boldsymbol{X}$. The simplest way to prove this is to appeal to a slightly fancy result involving the Cayley-Hamilton theorem. [5] Note that, by the Cayley-Hamilton theorem,

[5] The Cayley-Hamilton Theorem states that there is a degree n polynomial $q(x)$ such that $q(\boldsymbol{A}) = 0$. (And also that $q(x) = \prod_{i=1}^{n}(x - \lambda_i)$ where λ_i are the eigenvalues, but that isn't relevant.) Consider that $q(\boldsymbol{A})A^{-1} = 0$ too, but $q(\boldsymbol{A}) = c_n A^n + \ldots c_0 I$ so $q(\boldsymbol{A})A^{-1} = c_n \boldsymbol{A}^{n-1} + c_0 A^{-1} = 0$, which we can solve for A^{-1} to get a degree $n-1$ polynomial for the inverse.

there is a degree n polynomial $p(\boldsymbol{A})$ such that $p(\boldsymbol{A}) = \boldsymbol{A}^{-1}$. (See section below for an explicit derivation.) Hence, we by the assumptions of the theorem, we have that $p(\boldsymbol{A})\mathbf{b}$ is in the subspace too. ∎

The reason this theorem is nice is because is says we never need to be concerned about singular $\boldsymbol{X}$. If $\boldsymbol{X}$ is singular, then we have solved our linear system!

22.2.1 *A polynomial expression for the matrix inverse*

22.2.2 *The problem with the Krylov subspace*

When we want to work with the Krylov subspace, we need a basis for it. The simple choice is

$$\boldsymbol{X} = \begin{bmatrix} \mathbf{b} & \boldsymbol{A}\mathbf{b} & \boldsymbol{A}^2\mathbf{b} & \ldots \boldsymbol{A}^k\mathbf{b} \end{bmatrix}$$

as that is how the subspace is defined. The problem with this basis, however, is that $\boldsymbol{X}$ becomes very ill-conditioned as k gets large.

Let's see this for a *diagonal* linear system! Suppose that

$$\boldsymbol{A}_n = \begin{bmatrix} 1 & & & & \\ & 1/2 & & & \\ & & 1/4 & & \\ & & & 1/8 & \\ & & & & \ddots \end{bmatrix}$$

where $\boldsymbol{A}_n$ is n-by-n.

Then suppose that $\mathbf{b} = \mathbf{e}$, so we get the vector of all ones. We have that

$$\boldsymbol{X} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & 1/2 & 1/4 & \ldots & 1/(2^k) \\ 1 & 1/4 & 1/16 & \ldots & 1/(4^k) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1/(2^{n-1}) & 1/(2^{n-1})^2 & \ldots & 1/(2^{n-1})^k \end{bmatrix}.$$

For k large (think $k \geq 52$ for floating point), then

$$\boldsymbol{A}^k\mathbf{b} = \begin{bmatrix} 1 \\ \leq \varepsilon \\ \leq \varepsilon^2 \\ \leq \varepsilon^4 \\ \vdots \end{bmatrix} \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix}$$

Further note that for subsequent k , $\boldsymbol{A}^{k+1}\mathbf{b} \approx \boldsymbol{A}^k\mathbf{b}$ and so the matrix $\boldsymbol{X}$ is nearly singular because it has columns that are the same.

A good way to characterize this is via the ill-conditioning of the matrix. We plot the condition number of $\boldsymbol{X}$ as a function of k.

This example is hardly unique.

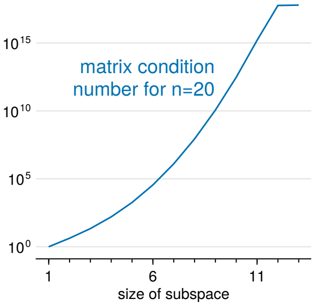


FIGURE 22.1 – The matrix $\boldsymbol{X}$ quickly becomes ill-conditioned with only a few entries from the matrix power basis.

Up next: A better basis for the subspace

What we'd ideally like is an orthogonal basis for $\mathbb{K}_k(\boldsymbol{A}, \mathbf{b})$. We can get this via the Arnoldi process. When the Arnoldi process is used on a symmetric matrix, it simplifies and becomes the Lanczos process.

23 ORTHOGONAL BASES FOR THE MATRIX POWERS SUBSPACE, AKA THE ARNOLDI AND LANCZOS PROCESSES

Learning objectives

1. Identify the Arnoldi process outcome as an orthogonal basis for the Krylov / matrix power subspace
2. Recognize how the Arnoldi process simplifies on a symmetric matrix and we can
3. Recognize that the residual of $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ simplifies when expressed in the Arnoldi or Lanczos basis for the Krylov subspace.

The Krylov or matrix power subspace is

$$\boldsymbol{K}_k = \operatorname{span}(\mathbf{b}, \boldsymbol{A}\mathbf{b}, \boldsymbol{A}^2\mathbf{b}, \ldots, \boldsymbol{A}^k\mathbf{b}).$$

As shown in the previous lecture, this subspace is useful to represent the iterates from the Neumann and Richardson methods.

The problem with this subspace is that the directly creating it from the monomial basis results in the ill-conditioning. Specifically, $\boldsymbol{X}_k = \begin{bmatrix}\mathbf{b} & \boldsymbol{A}\mathbf{b} & \boldsymbol{A}^2\mathbf{b} & \ldots & \boldsymbol{A}^k\mathbf{b}\end{bmatrix}$ is ill-conditioned. Note that there is already structure in the matrix $\boldsymbol{X}$. We have $\boldsymbol{X}_{k+1} = \begin{bmatrix}\mathbf{b} & \boldsymbol{A}\boldsymbol{X}_k\end{bmatrix}$. We also have[1]

$$\boldsymbol{A}\boldsymbol{X}_{k+1} = \boldsymbol{X}_k \begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix}.$$

An orthogonal matrix always has condition number 1. We now seek to understand how to build a well-conditioned basis for this subspace.

23.1 ORTHOGONALIZING AT EACH STEP

The first idea we consider is building the orthogonalization iteratively at each step.[2]

Let $\boldsymbol{X}_k = \begin{bmatrix}\mathbf{b} & \boldsymbol{A}\mathbf{b} & \boldsymbol{A}^2\mathbf{b} & \ldots & \boldsymbol{A}^k\mathbf{b}\end{bmatrix}$. Then suppose we have $\boldsymbol{X}_k = \boldsymbol{Q}_k\boldsymbol{R}_k$ as the thin-QR factorization of this matrix where $\boldsymbol{R}_k$ is square. We can straightforwardly update from

$$\boldsymbol{X}_k, \boldsymbol{Q}_k\boldsymbol{R}_k \quad \text{to} \quad \boldsymbol{X}_{k+1}, \boldsymbol{Q}_{k+1}, \boldsymbol{R}_{k+1}.$$

Let $\boldsymbol{X}_{k+1} = \begin{bmatrix}\boldsymbol{Q}_k\boldsymbol{R}_k & \mathbf{y}\end{bmatrix}$ be a block partitioning of this matrix. Then $\boldsymbol{Q}_k^T\boldsymbol{X}_{k+1} = \begin{bmatrix}\boldsymbol{R}_k & \boldsymbol{Q}_k^T\mathbf{y}\end{bmatrix}$. Consider the difference pieces of the last column of this matrix:

$$\boldsymbol{Q}_k^T\mathbf{y} = \begin{bmatrix}\mathbf{z}_1 \\ \mathbf{z}_2\end{bmatrix}.$$

If we build a Householder matrix $\boldsymbol{V}_{k+1}$ where

$$\boldsymbol{V}_{k+1}\mathbf{z} = \begin{bmatrix}\mathbf{z}_1 \\ \pm\|\mathbf{z}_2\|\mathbf{e}_1\end{bmatrix}$$

then we have

$$\boldsymbol{X}_{k+1} = \underbrace{\boldsymbol{Q}_k\boldsymbol{V}_{k+1}}_{=\boldsymbol{Q}_{k+1}} \underbrace{\begin{bmatrix}\boldsymbol{R}_k & \begin{bmatrix}\mathbf{z}_1 \\ \pm\|\mathbf{z}_2\|\mathbf{e}_1\end{bmatrix}\end{bmatrix}}_{=\boldsymbol{R}_{k+1}}.$$

[1] The matrix $\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix}$ has size $k+1$ by k and is a "upward shift"

$$\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix} = \begin{bmatrix}\mathbf{e}_2 & \mathbf{e}_3 & \cdots & \mathbf{e}_{k+1}\end{bmatrix}$$

or

$$\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ 0 & \cdots & \ddots & \ldots \\ 0 & \cdots & 0 & 1 \end{bmatrix}$$

[2] This ideas has been asked by students.

Note that the matrix $\boldsymbol{R}$ is not involved in building the orthogonal basis $\boldsymbol{Q}$ we are interested in. So we need not worry about maintaining $\boldsymbol{R}$.

This gives us an easy algorithm to build $\boldsymbol{Q}_k$ as a product of orthogonal reflectors matrices.

```
function successive_orthogonalization(A,b,k)
  # determine the element type of reflector
  FloatType = unify_type(eltype(A),eltype(b))
  Qs = OrthogonalReflector{FloatType}[]
  push!(Qs, reflector(FloatType, b, index=1))
  y = copy(b)
  for i in 1:k
    y = A*y
    z = reflector_transpose_product(Qs, y) # compute Qk'*y
    push!(Qs, reflector(FloatType, y, index=i+1))
  end
  return Qs
end
```

This algorithm, however, can be easily simplified. Note that we don't actually need all of the vector $\mathbf{z}$, we only need $\mathbf{z}_2$.

We also do not directly need the *full* orthogonal matrix $\boldsymbol{Q}$.

However, there is an even bigger simplification possible. Recall also that $\boldsymbol{A}\boldsymbol{X}_k = \boldsymbol{X}_{k+1}\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix}$. Using our QR decomposition:

$$\boldsymbol{A}\boldsymbol{X}_k = \boldsymbol{A}\boldsymbol{Q}_k\boldsymbol{R}_k = \boldsymbol{Q}_{k+1}\boldsymbol{R}_{k+1}\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix}.$$

The matrix $\boldsymbol{R}_k$ is non-singular because otherwise, the Krylov subspace would be complete. This means that

$$\boldsymbol{A}\boldsymbol{Q}_k = \boldsymbol{Q}_{k+1}\boldsymbol{R}_{k+1}\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix}\boldsymbol{R}_k^{-1}.$$

Finally,

the matrix $\boldsymbol{R}_{k+1}\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix}\boldsymbol{R}_k^{-1}$ is an upper Hessenberg matrix,

$$\boldsymbol{H}_{k+1}\boldsymbol{R}_{k+1}\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix}\boldsymbol{R}_k^{-1} = \text{TODO Add picture of upper Hessenberg.}$$

Using this idea gives us the Arnoldi process where we attempt to directly compute the matrix $\boldsymbol{H}_{k+1}$.

23.2 THE ARNOLDI PROCESS

The idea with the Arnoldi process is that we can build an orthogonal basis for

the Krylov subspace $\boldsymbol{K}_k$ directly from $\boldsymbol{A}, \mathbf{b}$ without QR.

That is, we can identify the matrix $\boldsymbol{Q}_k$ that is an orthogonal directly from the expression:

$$\boldsymbol{A}\boldsymbol{Q}_k = \boldsymbol{Q}_{k+1}\boldsymbol{H}_{k+1}, \quad \mathbf{q}_1 = \mathbf{b}/\|\mathbf{b}\|, \quad \boldsymbol{Q}_k^T\boldsymbol{Q}_k = \boldsymbol{I}, \quad \boldsymbol{Q} = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_k \end{bmatrix}.$$

This, perhaps surprisingly, can be done rather straightforwardly and constructively just by enumerating constraints and conditions.

Consider the case for $k = 1$

$$\boldsymbol{A}\mathbf{q}_1 = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 \end{bmatrix} \begin{bmatrix} H_{1,1} \\ H_{2,1} \end{bmatrix}.$$

We are given $\mathbf{q}_1$ by the construction of starting with $\mathbf{b}$. Hence, we can compute $\mathbf{y} = \boldsymbol{A}\mathbf{q}_1$. We want to find $\mathbf{q}_2$ such that

$$\mathbf{y} = H_{1,1}\mathbf{q}_1 + H_{2,1}\mathbf{q}_2$$

where $\mathbf{q}_2$ is orthogonal to $\mathbf{q}_1$. If we compute $\sigma = \mathbf{q}_1^T\mathbf{y}$, then we find

$$\sigma = \mathbf{q}_1^T\mathbf{y} = H_{1,1}\underbrace{\mathbf{q}_1^T\mathbf{q}_1}_{=1} + H_{2,1}\underbrace{\mathbf{q}_1^T\mathbf{q}_2^T}_{=0} = H_{1,1}.$$

Using the value of $H_{1,1}$ that we have constructed, we can compute

$$\mathbf{z} = \mathbf{y} - H_{1,1}\mathbf{q}_1.$$

By construction

$$\text{the vector } \mathbf{z} = H_{2,1}\mathbf{q}$$

and $H_{2,1} = \|\mathbf{z}\|$ (by convention, we take the positive norm in $H_{2,1}$ although we could also choose the negative norm) and the vector $\mathbf{q}_2 = \mathbf{y}/H_{2,1}$.

Now consider the case for $k = 2$ when we know $\mathbf{q}_1, \mathbf{q}_2$.

$$\boldsymbol{A}\begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 \end{bmatrix} \begin{bmatrix} H_{1,1} & H_{1,2} \\ H_{2,1} & H_{2,2} \\ 0 & H_{3,2} \end{bmatrix}.$$

We begin by computing $\mathbf{y} = \boldsymbol{A}\mathbf{q}_2$. In this case

$$\mathbf{y} = H_{1,2}\mathbf{q}_1 + H_{2,2}\mathbf{q}_2 + H_{3,2}\mathbf{q}_3.$$

Let $\sigma_1 = \mathbf{q}_1\mathbf{y}$ and we have

$$\sigma_1 = H_{1,2}\underbrace{\mathbf{q}_1\mathbf{q}_1}_{=1} + H_{2,2}\underbrace{\mathbf{q}_1^T\mathbf{q}_2}_{=0} + H_{3,2}\underbrace{\mathbf{q}_1^T\mathbf{q}_3}_{=0} = H_{1,2}.$$

Likewise, we can compute

$$\sigma_2 = \mathbf{q}_2\mathbf{y} = H_{2,2}.$$

Just as before,

$$\text{compute } \mathbf{z} = \mathbf{y} - H_{2,1}\mathbf{q}_1 - H_{2,2}\mathbf{q}_2 \text{ and}$$

$$\text{set } H_{3,2} = \|\mathbf{z}\| \quad \mathbf{q}_3 = \mathbf{z}/H_{3,2}.$$

We can easily code this algorithm. There is only one change we make. Because of the orthogonality of the vectors $\mathbf{q}_i$, after we compute σ_i

```
function arnoldi(A,b,k)
  Q = zeros(n,k+1)
  H = zeros(k+1,k)
  Q[:,1] .= b
  normalize!(@view(Q[:,1]))
  for j=1:k
    y = A*Q[:,j]
    for i=1:j
      H[i,j] = Q[:,i]'*y # σ_i
      y -= H[i,j]*Q[:,i]
    end
    H[j+1,j] = norm(y)   # y = z at this point
    Q[:,j+1] = y / H[j+1,j]
  end
  return Q, H
end
```

A cautious check

TODO - We want to verify that nothing went wrong! And directly show that the Arnoldi process spans the Krylov subspace.

23.3 THE LANCZOS PROCESS

23.4 ALTERNATIVE POLYNOMIAL BASES

I'm not sure what I was thinking here... maybe Newton polynomials as a basis for the Krylov subspace? Chebyshev methods?

EXERCISES

1. Fix $\boldsymbol{A}$ and $\mathbf{b}$. What is the condition number of problem of minimizing the residual in the kth Krylov subspace?
2. Show that the product $\boldsymbol{R}_{k+1}\begin{bmatrix} 0 \\ \boldsymbol{I}_k \end{bmatrix}\boldsymbol{R}_k^{-1}$ is upper triangular.
3. **Move this question after we introduce CG / MINRES, but we don't need the optimized versions.**
   Let $\boldsymbol{A}$ be a tridiagonal matrix with 2 on the diagonal and -1 on the off diagonal. Consider solving $\boldsymbol{A}\mathbf{x} = \mathbf{e}$.
   (a) Show that MINRES method has a simple form for the residual after k steps when starting from $\mathbf{x}_0 = 0$, so $\mathbf{r}_0 = \mathbf{e}$.
   (b) Show that the CG method has a simple form the residual after k steps when starting from $\mathbf{x}_0 = 0$, so $\mathbf{r}_0 = \mathbf{e}$.
   (c) Show that MINRES and CG will terminate in exactly $n/2$ steps.
4. Let

CONJUGATE GRADIENT 24

The conjugate gradients (CG) method is one of the most celebrated algorithms for solving $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ when $\boldsymbol{A}$ is large, sparse, and *symmetric positive definite*. It is also a sly method in the sense that there are *three derivations* of the CG method. Each starts from a different point, but gives rise to the same sequence of iterates. They are:

1. the Lanczos process – e.g. via matrix approximation
2. the steepest descent method – i.e. via optimization
3. the three-term recurrents – i.e. via orthogonal polynomials (This is done in lecture 25.)

The derivation for the Lanzcos process is, perhaps, the best as it provides a straightforward path to solve symmetric indefinite systems as well.

Throughout these notes, let $\boldsymbol{A}$ be $n \times n$, symmetric positive definite.

24.1 CONJUGATE GRADIENTS VIA THE LANCZOS PROCESS

Because $\boldsymbol{A}$ is symmetric, we can run the Lanczos process to iteratively compute a tridiagonal matrix $\boldsymbol{T}$ that approximates the matrix $\boldsymbol{A}$. *For linear systems, we also begin the Lanczos process with the vector* $\mathbf{b}/\|\mathbf{b}\|$. After k steps, we have:

$$\underbrace{\boldsymbol{A}}_{n\times n}\underbrace{\boldsymbol{V}}_{n\times k} = \underbrace{\boldsymbol{V}_{k+1}}_{n\times k+1}\underbrace{\boldsymbol{T}_{k+1}}_{k+1\times k}$$

where $\boldsymbol{V} = \begin{bmatrix}\mathbf{v}_1, \dots, \mathbf{v}_k\end{bmatrix}$ holds the first k vectors in the Lanczos process and $\boldsymbol{V}_{k+1}$ holds the first k vectors *and the* $k + 1$*st vector*. The matrix

$$\boldsymbol{T}_{k+1} = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_k \\ & & \beta_k & \alpha_k \\ & & & \beta_{k+1} \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{T}} \\ \beta_{k+1}\mathbf{e}_k^T \end{bmatrix}$$

where $\bar{\boldsymbol{T}}$ is the $k \times k$ tridiagonal matrix from the first k rows. When it's important, we'll write:

$$\boldsymbol{A}\boldsymbol{V}_k = \boldsymbol{V}_{k+1}\boldsymbol{T}_{k+1}$$

to denote the full sequence of matrices. Likewise, the Lanczos process gives rise to a sequence of tridiagonal matrices:

$$\bar{\boldsymbol{T}}_1, \bar{\boldsymbol{T}}_2, \dots, \bar{\boldsymbol{T}}_k, \bar{\boldsymbol{T}}_{k+1}, \dots$$

However, we'll often drop the index k when it applies to *any index*. For instance, in exact arithmetic, $\boldsymbol{V}^T\boldsymbol{A}\boldsymbol{V} = \bar{\boldsymbol{T}}$.

NOTE The vector $\mathbf{v}_1 = \mathbf{b}/\|\mathbf{b}\|$, and also $\boldsymbol{V}\mathbf{e}_1 = \mathbf{v}_1$ for all k.

QUIZ Show that $\bar{\boldsymbol{T}}$ is positive definite if $\boldsymbol{A}$ is positive definite.

24.1.1 *Lanczos and Linear Systems*

Let $\mathbf{y} = \boldsymbol{V}\mathbf{z}$ be a vector in the span of the Lanzcos vectors after k steps. Recall that we showed this means that $\mathbf{y}$ is a member of the kth Krylov subspace

$$\mathbf{y} \in \mathcal{K}_k(\boldsymbol{A}, \mathbf{b}).$$

For any vector $\mathbf{y} = \boldsymbol{V}\mathbf{z}$:

$$\|\mathbf{b} - \boldsymbol{A}\mathbf{y}\| = \|\mathbf{b} - \boldsymbol{A}\boldsymbol{V}\mathbf{z}\| = \|\boldsymbol{V}_{k+1}\left(\|\mathbf{b}\|\mathbf{e}_1 - \boldsymbol{T}_{k+1}\mathbf{z}\right)\| = \|\|\mathbf{b}\|\mathbf{e}_1 - \boldsymbol{T}_{k+1}\mathbf{z}\|.$$

Thus, we want to pick $\mathbf{z}$ such that $\|\mathbf{b}\|\mathbf{e}_1 - \boldsymbol{T}_{k+1}\mathbf{z}$ is small at each step.

In the conjugate gradients method, we choose $\mathbf{z}$ such that

$$\bar{\boldsymbol{T}}\mathbf{z} = \|\mathbf{b}\|\mathbf{e}_1$$

so that

$$\|\|\mathbf{b}\|\mathbf{e}_1 - \boldsymbol{T}_{k+1}\mathbf{z}\| = |\beta_{k+1}z_k|.$$

In contrast, in the MINRES method, we choose $\mathbf{z}$ to minimize $\|\|\mathbf{b}\|\mathbf{e}_1 - \boldsymbol{T}_{k+1}\mathbf{z}\|$ at each step; and in the SYMMLQ method, we choose $\mathbf{y} = \boldsymbol{V}_{k+1}\mathbf{z}$ and $\mathbf{z}$ is the minimum norm solution of $\boldsymbol{T}_{k+1}^T\mathbf{z}$. We won't spend too much time studying these methods.

24.1.2 *The simple CG method*

Consequently, and conceptually, the CG method is rather simple:

```
for k=1, 2, ...
  Compute $V_k, T_k$ from $k$-steps of the Lanczos process
  Solve $\bar{T}_k \mathbf{z}_k = \|\mathbf{b}\|\mathbf{e}_1$
  Compute $\mathbf{x}_k = V_k \mathbf{z}_k$
  If $|\beta_{k+1}\mathbf{z}_k| <$ tol, stop.
```

The essence of the method is that we replace solving $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ with solving $\bar{\boldsymbol{T}}_k\mathbf{z} = \|\mathbf{b}\|\mathbf{e}_1$. Put another way, the idea is that the matrix $\bar{\boldsymbol{T}}_k$ "approximates" $\boldsymbol{A}$.

THE DIFFICULTY WITH THE SIMPLE METHOD However, at each step, there is still quite a bit of work in this method. We can efficiently compute the kth step of the Lanczos vector sequence from the $k-1$st step, so computing $\boldsymbol{V}_k$ and $\boldsymbol{T}_k$ isn't a problem using one matrix vector and a few inner-products, so it's $O(\texttt{mat-vec} + n)$ work where $O(\texttt{mat-vec})$ is the work involved in the matrix-vector product. To solve the system with $\bar{\boldsymbol{T}}$ is $O(k)$ work because it's a tridiagonal system. However, the problem is that computing $\mathbf{x}_k = \boldsymbol{V}_k\mathbf{z}$ is $O(nk)$ work. If n is large and k is small, then this operation is expensive. Another problem is that we need to keep k Lanczos vectors. This gets *very* expensive in terms of memory for large k.

24.1.3 *Making CG efficient*

We'll now see how to do the CG method with $O(\texttt{mat-vec} + n)$ work per iterations. To do so, we need to determine how to compute $\mathbf{x}_k$ directly from $\mathbf{x}_{k-1}$ and avoid storing $\boldsymbol{V}_k$. We'll have to keep the last two iterates though, so we can continue the Lanczos process.

In the following discussion, we'll work through how to make this happen. There is one leap in this derivation that we'll get to soon.

USING AND UPDATING CHOLESKY FOR THE SUBSYSTEM We begin with a straightforward computation. Think about how to compute $\mathbf{z}_k$ efficiently. Recall that $\boldsymbol{A}$ is symmetric positive definite. Based on the quiz above, this means that $\bar{\boldsymbol{T}}$ is also symmetric, positive definite. So it has a Cholesky factorization

$$\bar{\boldsymbol{T}}_k = \boldsymbol{F}_k\boldsymbol{F}_k^T.$$

On the last homework, we worked out that:

$$\boldsymbol{F} = \begin{bmatrix} \eta_1 & & & \\ \mu_2 & \eta_2 & & \\ & \ddots & \ddots & \\ & & \mu_k & \eta_k \end{bmatrix}$$

Moreover, we can compute μ_{k+1} and η_{k+1} from

$$\bar{\boldsymbol{T}}_{k+1} = \begin{bmatrix} \bar{\boldsymbol{T}}_k & \beta_{k+1}\mathbf{e}_k \\ \beta_{k+1}\mathbf{e}_k^T & \alpha_{k+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{F}_k & \\ \mu_{k+1}\mathbf{e}_k^T & \eta_{k+1} \end{bmatrix} \begin{bmatrix} \boldsymbol{F}_k^T & \mu_{k+1}\mathbf{e}_k \\ & \eta_{k+1} \end{bmatrix}.$$

By equating terms, we find that

$$\beta_{k+1}\mathbf{e}_k = \mu_{k+1}\boldsymbol{F}_k\mathbf{e}_k = \mu_{k+1}\eta_k\mathbf{e}_k$$

and

$$\alpha_{k+1} = \eta_{k+1}^2 + \mu_{k+1}^2.$$

We can solve both of these to find:

$$\mu_{k+1} = \beta_{k+1}/\eta_k \text{ and } \eta_{k+1} = \sqrt{\alpha_{k+1} - \mu_{k+1}^2}.$$

But, there is no way to compute $\mathbf{z}_{k+1}$ from $\mathbf{z}_k$ because *all the elements* change. To go beyond this, we need to look at the problem more closely.

THE LEAP What we actually want is

$$\mathbf{x}_k = \|\mathbf{b}\|\,\boldsymbol{V}_k\bar{\boldsymbol{T}}^{-1}\mathbf{e}_1.$$

Let's substitute the Cholesky fatorization in here:

$$\mathbf{x}_k = \|\mathbf{b}\| \, \boldsymbol{V}_k \boldsymbol{F}^{-T} \boldsymbol{F}^{-1} \mathbf{e}_1.$$

The "leap" is that we need to look at:

$$\mathbf{x}_k = \|\mathbf{b}\| \cdot \underbrace{\boldsymbol{C}_k}_{=\boldsymbol{V}_k \boldsymbol{F}^{-T}} \cdot \underbrace{\mathbf{p}_k}_{=\boldsymbol{F}^{-1}\mathbf{e}_1} .$$

As we study these expressions, we'll find that they can be updated efficiently.

First, let's tackle $\mathbf{p}_k$. Given $\mathbf{p}_k = \begin{bmatrix} p_1, \dots, p_k \end{bmatrix}^T$, note that:

$$\boldsymbol{F}_{k+1}\mathbf{p}_{k+1} = \begin{bmatrix} \boldsymbol{F} & \\ \mu_{k+1}\mathbf{e}_k^T & \eta_{k+1} \end{bmatrix} \begin{bmatrix} \mathbf{p}_k \\ p_{k+1} \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1 \\ 0 \end{bmatrix}.$$

Thus, $p_{k+1} = -\mu_{k+1} p_k / \eta_{k+1}$. This is great news because $\mathbf{p}_{k+1}$ only differs from $\mathbf{p}_k$ by the last element.

Let's see how to find $\boldsymbol{C}_{k+1}$ from $\boldsymbol{C}_k$. We'll write:

$$\boldsymbol{C}_{k+1}\boldsymbol{F}_{k+1}^T = \boldsymbol{V}_{k+1}$$

or

$$\begin{bmatrix} \boldsymbol{C}_k & \mathbf{c}_{k+1} \end{bmatrix} \begin{bmatrix} \boldsymbol{F}_k^T & \mu_{k+1}\mathbf{e}_k \\ & \eta_{k+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{V}_k & \mathbf{v}_{k+1} \end{bmatrix}.$$

Hence,

$$\mu_{k+1}\boldsymbol{C}_k\mathbf{e}_k + \mathbf{c}_{k+1}\eta_{k+1} = \mu_{k+1}\mathbf{c}_k + \eta_{k+1}\mathbf{c}_{k+1} = \mathbf{v}_{k+1}.$$

Thus, we can compute $\mathbf{c}_{k+1}$ just from $\mathbf{c}_k$.

The last step is to show that we can combine these and compute $\mathbf{x}_{k+1}$ from $\mathbf{x}_k$. Again, we expand:

$$\mathbf{x}_{k+1} = \boldsymbol{C}_{k+1}\mathbf{p}_{k+1} = \underbrace{\boldsymbol{C}_k\mathbf{p}_k}_{=\mathbf{x}_k} + \mathbf{c}_{k+1}p_{k+1}.$$

And we have found an efficient expression for $\mathbf{x}_k$ in the CG method.

All together now, we have:

```
$\beta_1 = \|\mathbf{b}\|$
$\mathbf{v}_0 = 0, \ \mathbf{x}_0 = 0$
$\mathbf{v}_1 = \mathbf{b}/\beta_1$
for i=1, 2, ...
    $\mathbf{w} = \boldsymbol{A}\mathbf{v}_i - \beta_i \mathbf{v}_{i-1}$
    $\alpha_i = \mathbf{v}_i^T \mathbf{w}$
    $\mathbf{w} \leftarrow \mathbf{w} - \alpha_i \mathbf{v}_i$
    $\beta_{i+1} = \|\mathbf{w}\|$
    $\mathbf{v}_{i+1} = \mathbf{w}/\beta_{i+1}$
    if $i = 1$
        $\mu_i = 0, \ \eta_i = \sqrt{\alpha_i}, \ p_i = \beta_1/\eta_i, \ \mathbf{c}_i = \mathbf{v}_1/\eta_1.$
    else
        $\mu_i = \beta_i/\eta_{i-1}, \ \eta_i = \sqrt{\alpha_i - \mu_i^2}, \ p_i = -\mu_i p_{i-1}/\eta_i, \ \mathbf{c}_i = (\mathbf{v}_i - \mu_i \mathbf{c}_{i-1})/\eta_i$
    $\mathbf{x}_i = \mathbf{x}_{i-1} + p_i \mathbf{c}_i$
```

In this iteration, you only need to keep the vector $\mathbf{v}_i$ and $\mathbf{c}_i$ to complete the iteration.

24.2 CONJUGATE GRADIENTS VIA OPTIMIZATION

This section is incomplete

In the second derivation of the CG method, we study the problem:

$$\min \phi(x) \text{ where } \phi(x) = \frac{1}{2}\mathbf{x}^T \boldsymbol{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$$

where $\boldsymbol{A}$ is $n \times n$, symmetric positive definite. In this case, the solution is $\mathbf{x} = \boldsymbol{A}^{-1}\mathbf{b}$, which we can derive by setting the gradient of $\phi(x)$ to zero,[1] that is,

$$\partial\phi/\partial\mathbf{x} = \boldsymbol{A}\mathbf{x} - \mathbf{b} = 0.$$

Thus, in the second derivation of the CG method we work from the premise of finding a sequence of vectors $\mathbf{x}_k$ that make $\phi(\mathbf{x}_k)$ smaller at each step.

ASIDE ON STEEPEST DESCENT One of the classic ways to minimize a function is called gradient descent, and it computes

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha_k \mathbf{g}_k$$

where $\alpha_k > 0$ and $\mathbf{g}_k$ is the gradient $\partial\phi/\partial\mathbf{x}$ evaluated at $\mathbf{x}^{(k)}$. For this function ϕ, $\mathbf{g}_k = \boldsymbol{A}\mathbf{x}_k - \mathbf{b}$. The constant α_k is chosen to make:

$$\phi(\mathbf{x}_k - \alpha\mathbf{g}_k)$$

as *small as possible*. It's a bit of a tangent to derive this value, but we can work out the solution, which is:[2]

$$\alpha_k = \frac{\mathbf{g}_k^T\mathbf{g}_k}{\mathbf{g}_k^T\boldsymbol{A}\mathbf{g}_k}.$$

This simple method:

```
$\mathbf{x}^{(1)} = 0$
for k=1, ...
  $\mathbf{g}^{(k)} = \boldsymbol{A}\mathbf{x}^{(k)} - \mathbf{b}$
  if $\|\mathbf{g}^{(k)}\|$ < tol
    stop and return $\mathbf{x}_k$
  $\alpha^{(k)} = \mathbf{g}^{(k)T}\mathbf{g}^{(k)}/(\mathbf{g}^{(k)T}\boldsymbol{A}\mathbf{g}^{(k)})$
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{g}^{(k)}$
```

will always converge to the solution of a positive definite system using only matrix-vector products. The convergence rate is proportional to the condition number of the matrix.

Now, suppose we consider a sequence of directions $\mathbf{p}^{(k)}$ such that $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)}$, we can set:

$$\alpha_k = \mathbf{p}^{(k)T}\mathbf{r}^{(k)}/(\mathbf{p}^{(k)T}\boldsymbol{A}\mathbf{p}^{(k)})$$

as long as $\mathbf{p}^{(k)}\mathbf{r}^{(k)} \neq 0$ where $\mathbf{r}^{(k)}$ is the kth residual $\mathbf{b} - \boldsymbol{A}\mathbf{x}^{(k)}$.

To get to conjugate gradients, we want the set of search directions $\mathbf{p}^{(k)}$ to be linearly independent, and in fact, conjugate. We call a sequence of vectors conjugate if $\mathbf{p}^{(i)T}\boldsymbol{A}\mathbf{p}^{(j)} = 0$ if $i \neq j$. In this case, $\mathbf{x}^{(k)} = \sum_{i=1}^{n} \alpha_i\mathbf{p}^{(i)}$ or $\mathbf{x}^{(k)} \in \operatorname{span}\{\mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(k)}\}$.

[1] this condition suffices because the problem is strongly convex

[2] check the sign on this value

24.3 CG HISTORY

1. Initially proposed by Hestenes and Stiefel as a direct method (1952). Both Hestenes and Stiefel came up with the method independently, and then wrote a joint paper about it.
2. First suggested as a large sparse solver by Reid (1971).
3. Finally widely accepted for matrices once preconditioning was invented.

Information from Diane O'Leary, `https://www.siam.org/meetings/la09/talks/oleary.pdf` and Numerical Analysis: Historical Developments in the 20th Century; By C. Brezinski, L. Wuytack; Gene H. Golub and Dianne P. O'Leary, "Some history of the conjugate gradient and Lanczos algorithms: 1948-1976," SIAM Review 31 (1989) 50-102. `http://www.cs.umd.edu/~oleary/reprints/j28.pdf`

25 ORTHOGONAL POLYNOMIALS & MATRIX COMPUTATIONS

One of the most wonderful and surprising connections in the field of matrix computations is the elegant interplay between matrices and orthogonal polynomials. These relationships lead to magnificently simple insights into complicated methods. In this lecture we shall unshroud some of these connections. However, the field is so deep. Probably the best textbook on the topic was written by Purdue's own Walter Gautschi. It is tersely titled: "Orthogonal Polynomials."

The notes here arise from Saad, Iterative Methods, 2nd edition §6.72; Gutknecht, A General Framework for Recursions for Krylov Space Solvers, ETH SAM report 2005-09, `ftp://ftp.sam.math.ethz.ch/pub/sam-reports/reports/reports2005/2005-09.pdf`; and my own notes from working with Gene Golub.

We will use $\mathbf{x}_k$ as the kth iterate in this section instead of $\mathbf{x}^{(k)}$. This is because we are frequently taking transposes of the iterate and $\mathbf{x}^{(k)\,T}$ becomes awkward.

25.1 WHAT ARE ORTHOGONAL POLYNOMIALS?

For the purposes of this lecture, a polynomial is a univariate function:[1]

$$p(t) = \sum_{i=0}^{n} p_i t^i.$$

For example:

$$p(t) = \frac{1}{2}t^2 - \frac{1}{2}$$

$$q(t) = \frac{5}{2}t^3 - \frac{3}{2}t.$$

The degree of a polynomial is the power of the largest term. In the generic polynomial definition, the degree is n. For the examples of p and q, the degrees are 2 and 3 respectively. A polynomial of degree 0 is a constant.

We call two polynomials orthogonal if:

$$\int_{-1}^{1} p(t)q(t)\,dt = 0.$$

This is a type of continuous analog of two vectors:

$$\mathbf{v}^T\mathbf{u} = \sum_{i=1}^{n} v_i u_i = 0.$$

[1] Multivariate generalizations exist, but we won't need them.

EXAMPLE 25.1 The two polynomials p and q given above are orthogonal.

$$\begin{aligned}\int_{-1}^{1} p(t)q(t)\,dt &= \int_{-1}^{1} \frac{1}{2}(t^2-1)\frac{1}{2}(5t^3-3t)\,dt\\ &= \frac{1}{4}\int_{-1}^{1} 5t^5-5t^3-3t^3+3t\,dt\\ &= \frac{1}{4}\int_{-1}^{1} 5t^5-8t^3+3t\,dt.\end{aligned}$$

All of these terms are odd functions, and they are integrated over a symmetric region, hence the result is 0. ◆

More generally, we can consider polynomials that are orthogonal with respect to

an arbitrary interval	$\int_a^b p(t)q(t)\,dt$
a weighted integral	$\int_a^b p(t)q(t)\,dw(t)$
a discrete weight	$\int_a^b p(t)q(t)\,dw(t) = \sum_{i=1}^{n} p(\lambda_i)q(\lambda_i)w_i$

If two polynomials are orthogonal with respect to an integral $\int_a^b\, dw(t)$, then we'll often call this the *measure* that they are orthogonal with respect to.

Sequences and families of orthogonal polynomials

We are often concerned with a sequence or family of orthogonal polynomials. We will index these by the degree of a polynomial, so that we have the sequence in order of increasing degree. Let $p_k(t)$ be the polynomial of degree k. By convention, we take $p_{-1}(t) = 0$ and $p_0(t) = c$ for some constant.

Thus, we have the following sequence of orthogonal polynomials that are orthogonal with respect to $\int_{-1}^{1}\, dt$:

$$\begin{aligned}p_{-1}(t) &= 0\\ p_0(t) &= 1\\ p_1(t) &= t\\ p_2(t) &= \tfrac{1}{2}(3t^2-1)\\ p_3(t) &= \tfrac{1}{2}(5t^3-t)\\ p_4(t) &= \tfrac{1}{8}(35t^4-30t^2+3)\end{aligned}$$

This sequence is called the Legendre polynomials. Other popular families are:

Family	Measure
Legendre	$\int_{-1}^{1} dt$
Laguarre	$\int_{0}^{\infty} e^{-t}\, dt$
Hermite	$\int_{-\infty}^{\infty} e^{-t^2}\, dt$
Chebyshev (1st kind)	$\int_{-1}^{1} \frac{1}{\sqrt{1-t^2}}\, dt$
Chebyshev (2nd kind)	$\int_{-1}^{1} \sqrt{1-t^2}\, dt$
Jacobi	$\int_{-1}^{1} (1-t)^{\alpha}(1+t)^{\beta}\, dt$
Gegenbauer	$\int_{-1}^{1} (1-t^2)^{\alpha-1/2}\, dt$

Note that these can be scaled and shifted to arbitrary intervals $[a, b]$ too.

Orthonormal polynomials

Yes, there are orthonormal polynomials too! Checkout Gautschi's book for more on the relationship. If $\int_a^b dw(t)$ is the measure, the idea is that we need $\int_a^b p(t)^2\, dw(t) = 1$ in order to get orthonormal polynomials.

25.2 THE THREE TERM RECURRENCE & TRIDIAGONAL MATRICES

The following fact, which we will not prove, is profound:

> ***Theorem 25.2***
> Any sequence of orthogonal polynomials of increasing degree satisfies a three-term recurrence and any three-term recurrence defines a sequence of orthogonal polynomials.

EXAMPLE 25.3 Consider the Legendre polynomials (described above). They satisfy:

$$p_{k+1}(t) = \frac{2k+1}{k+1} t p_k(t) - \frac{k}{k+1} p_{k-1}(t).$$

For instance,

$$p_2(t) = \frac{2+1}{1+1} t \underbrace{t}_{p_1(t)} - \frac{1}{1+1} \underbrace{1}_{p_0(t)}.$$

◆

The general form of the three-term recurrence is:[2]

$$p_{k+1}(t) = \mu_k\big(p_k(t) - \gamma_k t p_k(t)\big) + \eta_k p_{k-1}(t),$$

where the constants μ_k, γ_k, η_k may depend on k. What this recurrence means is that if we have any sequences of numbers μ_k, γ_k, η_k, then they give rise to a set of polynomials that is orthogonal with respect to *some measure*.

[2] In the notes I'm writing this from, μ_k is ρ_k, so beware of future typos.

From a three-term recurrence to a tridiagonal matrix

It's this three term recurrence that brings us back to matrix computations, and specificall tridiagonal matrices. Consider the Legendre family. Another way to write the recurrence is:

$$p_{-1}(t) = 0 \quad p_0(t) = 1 \quad p_1(t) = t \quad (k{+}1)p_{k+1}(t) = (2k{+}1)tp_k(t){-}kp_{k-1}(t).$$

This recurrence forms a matrix:

$$\underbrace{\begin{bmatrix} 1 & & & & & \\ -t & 1 & & & & \\ 1 & -3t & 2 & & & \\ 0 & \ddots & \ddots & \ddots & & \\ & & k & -(2k+1)t & (k+1) & \\ & & & \ddots & \ddots & \ddots \end{bmatrix}}_{\text{tridiagonal}T} \underbrace{\begin{bmatrix} p_0(t) \\ p_1(t) \\ p_2(t) \\ \vdots \\ p_{k+1}(t) \\ \vdots \end{bmatrix}}_{\mathbf{p}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \end{bmatrix}$$

Thus, for instance, we can *evaluate* a sequence of orthogonal polynomials at a point t by constructing $\boldsymbol{T}(t)$ and solving:

$$\boldsymbol{T}(t)\mathbf{p} = \mathbf{e}_1.$$

Consequently, *any* tridiagonal matrix corresponds to a set of orthogonal polynomials.

There's a Matlab demo of this in the `orthopolys.m` function!

There are additional relationships with a matrix called the Jacobi matrix:

$$\boldsymbol{J} = \begin{bmatrix} \alpha_1 & 1 & & \\ \beta_1 & \alpha_2 & 1 & \\ 0 & \ddots & \ddots & \ddots 0 \end{bmatrix}$$

but you'll need to read more about that in Gautschi's book.

25.3 POLYNOMIALS AND MATRICES

Note that if we have a univariate polynomial $p(t) = \sum_{i=0}^{n} p_i t^i$, then we can evaluate that polynomial with a *square* matrix argument:

$$p(\boldsymbol{A}) = \sum_{i=0}^{n} p_i \boldsymbol{A}^i.$$

25.4 POLYNOMIALS AND ITERATIVE METHODS

First, let us introduce the idea of using polynomials and iterative methods. This has been a homework or exam problem in the past, so it's worth understanding the details! Consider a Krylov subspace method. The kth iterate $\mathbf{x}_k$ is in $\mathbb{K}_k(\boldsymbol{A}, \mathbf{b}) = \text{span}\{\mathbf{b}, \boldsymbol{A}\mathbf{b}, \ldots, \boldsymbol{A}^{k-1}\mathbf{b}\}$. This means that there is some polynomial such that:[3]

[3] This is the detail that you should work out!

$$\mathbf{x}_k = s_{k-1}(\boldsymbol{A})\mathbf{b}.$$

Now, if $\mathbf{x}_k$ is determined by a polynomial $s_{k-1}(t)$ of degree $k-1$, this means the residual at the kth step is determined by a polynomial of degree k:

$$\mathbf{r}_k = \mathbf{b} - \boldsymbol{A}\mathbf{x}_k = \mathbf{b} - \boldsymbol{A}s_{k-1}(\boldsymbol{A})\mathbf{b} = \big(\boldsymbol{I} - \boldsymbol{A}s_{k-1}(\boldsymbol{A})\big)\mathbf{b}. \quad \text{Thus} \quad \mathbf{r}_k = p_k(\boldsymbol{A})\mathbf{b}$$

where the polynomial $p_k(t) = 1 - ts_{k-1}(t)$ has degree k.

We call these two polynomials:

$$\begin{array}{rl} s_k(t) & \text{the solution polynomial} \\ p_k(t) & \text{the residual polynomial.} \end{array}$$

It turns out that the residual polynomial already must have some special structure! Note that $p_k(t)$ is defined to be equal to $1 - ts_{k-1}(t)$. Thus, $p_k(0) = 1$. So any residual polynomial must evaluate to 1 at the value $t = 0$. As a matrix statement, this means: $p_k(0)\mathbf{b} = \mathbf{b}$.

25.5 ORTHOGONAL POLYNOMIALS AND ITERATIVE METHODS

Thus far, we haven't run into orthogonal polynomials yet. But let's design an iterative method with a fairly natural property using orthogonal polynomials.

DESIGN GOAL We want the kth residual from the iterative method to be *orthogonal* to all previous residuals. Or more formally, $\mathbf{r}_k^T\mathbf{r}_j = 0$ for $j < k$. This goal is equivalent to the idea that our residual should always include *new* information at each step and should never include information we could have factored out.

Orthogonal polynomials will help us achieve this goal!

Let's state what we have:

$$\mathbf{r}_k = \mathbf{b} - \boldsymbol{A}\mathbf{x}_k = p_k(\boldsymbol{A})\mathbf{b}.$$

We want:

$$\mathbf{r}_k^T\mathbf{r}_j = \mathbf{b}^T p_k(\boldsymbol{A})^T p_j(\boldsymbol{A})\mathbf{b} = 0.$$

Thus, if we *create* an orthogonal polynomial $p_k(t)$ where $p_k(0) = 1$ and

$$\int p_k(t)p_j(t)\,dw(t) = (\mathbf{r}_k)^T\mathbf{r}_j,$$

we will implicitly create an iterative method where the residuals are orthogonal.[4]

The three term recurrence helps us do this!

[4] Take a moment to understand what is going on here, as it's a key step. We are first noting that $\mathbf{r}_k$ can be expressed as a polynomial in $\boldsymbol{A}$. We are now saying, let's control that polynomial to achieve our goal! But we'll have to obey some constraints to make it work.

Recall:

$$p_{k+1}(t) = \mu_k\big(p_k(t) - \gamma_k t p_k(t)\big) + \eta_k p_{k-1}(t).$$

So if we can determine μ_k, γ_k, η_k from our constraints, then we'll be able to figure out what the next residual polynomial is.

Let's enumerate our constraints:

$$\begin{aligned}
&(\text{i}) p_{k+1}(0) = 1\\
&(\text{ii}) \mathbf{r}_{k+1} = p_{k+1}(\boldsymbol{A})\mathbf{b} = \mu_k\big(\mathbf{r}_k - \gamma_k \boldsymbol{A}\mathbf{r}_k\big) + \eta_k \mathbf{r}_{k-1}\\
&(\text{iii}) (\mathbf{r}_{k+1})^T \mathbf{r}_j = 0, j < k.
\end{aligned}$$

Constraint (i) implies:

$$1 = p_{k+1}(0) = \mu_k + \eta_k \quad \Rightarrow \quad \eta_k = 1 - \mu_k.$$

Thus, we now have a revised constraint (ii):

$$\mathbf{r}_{k+1} = p_{k+1}(\boldsymbol{A})\mathbf{b} = \mu_k\big(\mathbf{r}_k - \gamma_k \boldsymbol{A}\mathbf{r}_k\big) + (1 - \mu_k)\mathbf{r}_{k-1}.$$

If we apply constraint (iii) with $j = k$, we have:

$$0 = (\mathbf{r}_k)^T \mathbf{r}_{k+1} = \mu_k\big((\mathbf{r}_k)^T \mathbf{r}_k - \gamma_k (\mathbf{r}_k)^T \boldsymbol{A}\mathbf{r}_k\big) + (1 - \mu_k)(\mathbf{r}_k)^T \mathbf{r}_{k-1}$$

or

$$\gamma_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_k^T \mathbf{A}\mathbf{r}_k}.$$

Finally, using constraint (iii) with $j = k - 1$, we can solve for[5]

$$\mu_k = \frac{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1} + \gamma_k \mathbf{r}_{k-1}^T \boldsymbol{A}\mathbf{r}_k.}$$

This lets us compute $\mathbf{r}_{k+1}$

Getting the solution polynomial

Thus far, we have the residual polynomial $\mathbf{r}_k = p_k(\boldsymbol{A})\mathbf{b}$. We do, however, need to recreate the solution too! To do so, we note that

$$s_k(t) = \frac{1 - p_{k+1}(t)}{t} = \frac{1 - \mu_k(p_k(t) - \gamma_k t p_k(t)) - (1 - \mu_k)p_{k-1}(t)}{t}.$$

Now we add and subtract μ_k/t in order to rewrite this as:

$$\begin{aligned}
s_k(t) &= \mu_k\left[\frac{1 - p_k(t)}{t} - \gamma_k p_k(t)\right] - (1 - \mu_k)\frac{1 - p_{k-1}(t)}{t}\\
&= \mu_k(s_{k-1}(t) - \gamma_k p_k(t)) - (1 - \mu_k)s_{k-2}(t).
\end{aligned}$$

Hence, we have:

$$\mathbf{x}_{k+1} = \mu_k(\mathbf{x}_k - \gamma_k \mathbf{r}_k) - (1 - \mu_k)\mathbf{x}_{k-1}.$$

[5] Check for an error here... and this simplifies more, see Saad

25.6 A POLYNOMIAL FORM OF CONJUGATE GRADIENT

In a small surprise, we've arrived at a new form of the conjugate gradient algorithm! The iterates generated by this method are mathematically equivalent to those generated by CG! The way to prove this is to show that the residuals constructed in CG automatically satisfy the same property and live in the same subspace, hence, they must be the same.

26 EFFICIENT GMRES

Recall the prototype-GMRES method.

```
Given $A,\mathbf{b}$ where we can only multiply by $A$.
for i=1 to maxiter
  Update the Arnoldi factorization $Q_k, H_{k+1}$.
  Solve for $\mathbf{z}_k$ by minimizing $\|H_{k+1}\mathbf{z}_k - \|\mathbf{b}\|\mathbf{e}_1\|$,
    i.e. $\mathbf{z}_k = \operatorname{argmin} \|H_{k+1}\mathbf{z}_k - \|\mathbf{b}\|\mathbf{e}_1\|$
  Let $\mathbf{x}_k = Q_k\mathbf{z}_k$.
  Check $\|A\mathbf{x}_k - \mathbf{b}\|$
```

To implement this, we need to solve a lseast squares problem at each step. This takes $O(k^2)$ work because it's a Hessenberg matrix. Then we need to construct the solution and check the residual. These take $O(nk)$ and another matrix-vector product. We can do all of these steps more efficiently!

Here is the outline for the essential idea to optimize GMRES.

we only need to check the residual at each step, and do not need to compute $\mathbf{x}_k$.

So the method we'll look at optimizing is:

```
Given $A,\mathbf{b}$ where we can only multiply by $A$.
for i=1 to maxiter
  Update the Arnoldi factorization $Q_k, H_{k+1}$.
  Compute $\|\mathbf{r}_k\|$ where
      $\mathbf{r}_k = A\mathbf{x}_k - \mathbf{b}$
      $\mathbf{x}_k = Q_k\mathbf{z}_k$
      $\mathbf{z}_k = \operatorname{argmin} \|H_{k+1}\mathbf{z}_k - \|\mathbf{b}\|\mathbf{e}_1\|$
    and stop once $\|\mathbf{r}_k\|$ is sufficiently small, i.e.
    Update $\|\mathbf{r}_k\| \to \|\mathbf{r}_{k+1}\|$ and stop if it's small enough.
Explicitly compute $\mathbf{z}_k = \operatorname{argmin} \|H_{k+1}\mathbf{z}_k - \|\mathbf{b}\|\mathbf{e}_1\|$
  and return $\mathbf{x}_k = Q_k\mathbf{z}_k$ only at the end of the iteration
```

26.0.1 *The optimization idea*

Let's study the quantity we want to compute, let $\|\mathbf{b}\| = \beta_0$, then

$$\|\mathbf{r}_k\| = \|\mathbf{b} - A\mathbf{x}_k\| = \|\mathbf{b} - AQ_k\mathbf{y}_k\| = \|H_k\mathbf{y}_k - \beta_0\mathbf{e}_1\|.$$

After a four steps, this is:

$$\left\| \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}}_{H_k} \mathbf{y}_4 - \beta_0 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right\|.$$

We solve least squares problems via QR, so suppose that

$$H_k = U_k R_k$$

is the QR factorization after k-steps. Then

$$\|\mathbf{r}_k\| = \|\boldsymbol{U}_k\boldsymbol{R}_k\mathbf{y}_k - \beta_0\mathbf{e}_1\| = \|\boldsymbol{R}_k\mathbf{y}_k - \beta_0\boldsymbol{U}_k^T\mathbf{e}_1\|.$$

Showing this after a few steps gives us the idea more clearly:

$$\|\mathbf{r}_k\| = \left\| \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\boldsymbol{R}_k} \mathbf{y}_4 - \beta_0 \underbrace{\begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_5 \end{bmatrix}}_{\boldsymbol{U}_k^T\mathbf{e}_1} \right\|. = \beta_0\gamma_5.$$

(Remember we solve for $\mathbf{y}_k$ such that this term is zero in the first four components. So we just need to figure out what γ_5 is to get $\|\mathbf{r}_k\|$.

26.0.2 *Taking it deeper*

We need to note a two things here to continue our optimization:

1. We only need Givens rotations to get $\boldsymbol{H}_k \to \boldsymbol{R}_k$.
2. We only need *one* rotation to update $\boldsymbol{R}_k \to \boldsymbol{R}_{k+1}$. (Woah!)

Let's review step 1 and see how that will help us with step 2.

Finish and improve this figure, the point is we use $J_1, \ldots, J_4$ to do the Givens rotations. These give us $\boldsymbol{U}_4^T = \boldsymbol{J}_4\boldsymbol{J}_3 \ldots \boldsymbol{J}_1$.

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{J_1} \begin{bmatrix} * & * & * & * \\ \circ & * & * & * \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \qquad \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{J_2} \begin{bmatrix} \times & \times & \times & \times \\ 0 & * & * & * \\ 0 & \circ & * & * \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{J_3} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & * & * \\ 0 & 0 & \circ & * \\ 0 & 0 & 0 & \times \end{bmatrix} \qquad \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{J_4} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & \circ \end{bmatrix}$$

Now, suppose we have $\boldsymbol{U}_4$ and $\boldsymbol{R}_4$, how do we get $\boldsymbol{U}_5, \boldsymbol{R}_5$?

$$\boldsymbol{H}_5 = \begin{bmatrix} \boldsymbol{H}_4 & \mathbf{h} \\ 0 & h_{6,5} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} & \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} & h_{6,5} \end{bmatrix}$$

If we rotate by $\boldsymbol{U}_4^T$, we get:

$$\begin{bmatrix} \boldsymbol{U}_4^T & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{H}_5 = \begin{bmatrix} \boldsymbol{U}_4^T & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{H}_5 = \begin{bmatrix} \boldsymbol{R}_4 & \boldsymbol{U}_4^T\mathbf{z}_4 \\ 0 & h_{6,5} \end{bmatrix}$$

So at this point, we just have the one Givens rotation: $\boldsymbol{J}_5$ that we need to do to fixup the element $h_{6,5}$ and so $\boldsymbol{U}_5^T = \boldsymbol{J}_5\boldsymbol{J}_4 \cdots \boldsymbol{J}_1$, which is just one update.

26.0.3 *Seeking gamma.*

Note that the elements of gamma are just the first column of $\boldsymbol{U}_k^T$. Let $\mathbf{g}_k = \boldsymbol{U}_k^T\mathbf{e}_1 = \begin{bmatrix} \gamma_1 & \gamma_2 & \ldots & \gamma_k \end{bmatrix}^T$. Then by our previous relationship:

$$\mathbf{g}_{k+1} = \boldsymbol{U}_{k+1}^T\mathbf{e}_1 = \boldsymbol{J}_{k+1}\boldsymbol{U}_k^T\mathbf{e}_1 = \boldsymbol{J}_{k+1}\mathbf{g}_k.$$

But this is weird, because $\boldsymbol{J}$ is a $k + 1 \times k + 1$ matrix and $\mathbf{g}_k$ is a length k vector. So what we really mean is

$$\boldsymbol{J}_{k+1}\begin{bmatrix} \mathbf{g}_k \\ 0 \end{bmatrix}$$

where we grew the vector by one element in order to make it work. Note that we don't need to actually update $\mathbf{g}_k$ even though it should change.

26.0.4 *The whole algorithm*

```
$\mathbf{g} = \beta_0 \mathbf{e}_1$
for k=1 to ...
  Update $\mathbf{Q}_k, \boldsymbol{H}_k$
  Let $\eta_{k+1} = H_{k+1,k}$.
  Let $\mathbf{z}_k = H_{1:k,k}$.
  Apply $\boldsymbol{J}_1 ... \boldsymbol{J}_{k-1}$ to $\mathbf{z}_k$, and update $\boldsymbol{H}$
  Create $\boldsymbol{J}_k$ to elminate $\eta_{k+1}$.
  Determine $\mathbf{g}_k$ from $\boldsymbol{J}_k \mathbf{g}_{k-1}$ growing by zeros as needed.
  If $\mathbf{g}_k(\text{end})$ is small enough, then stop iterating.
At this point, $\boldsymbol{H}$ has the factor $\boldsymbol{R}$, and (if we do keep $\mathbf{g}$ accurate), then
$\mathbf{g}$ is the right hand side, so we can just solve $\boldsymbol{R}_k \mathbf{y}_k = \mathbf{g}_k$ and then
output $\mathbf{x} = \mathbf{Q}_k \mathbf{y}$.
```

26.1 GMRES VS. FOM

See notes.

This section is incomplete.

The major point is that FOM is like CG in that it solves a linear system based on the truncated Arnoldi factorization.

The large scale study by Peter Brown (`http://dx.doi.org/10.1137/0912003`) concluded: there is little difference, but liked the minimum residual property of GMRES.

ADVANCED PROBLEMS VI

The standard problems in linear algebra are

$$\min \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\| \qquad \text{least squares}$$

$$\boldsymbol{A}\mathbf{x} = \mathbf{b} \qquad \text{linear systems}$$

$$\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x} \qquad \text{eigenvalues}$$

$$\boldsymbol{A}\mathbf{x} = \sigma\mathbf{y} \qquad \text{singular values}$$

In this part, we look at variations on these problems that we call

advanced

not because they are hard, but because they would occur in the context of a different type of application. We will also see one new problem, the matrix function!

27 MULTIPLE RIGHT HAND SIDES

For a linear system of least squares problem, an extremely common variation is that we have a set of right hand sides to solve. This occurs in two forms, one where all of the vectors are available at once, and a second where each solution gives rise to a new problem.

27.1 MULTIPLE RIGHT HAND SIDES ALL KNOWN AHEAD OF TIME

The problem here is that we need to solve $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ for many vectors $\mathbf{b}_1, \dots, \mathbf{b}_k$. In the simplest case, we will know $\mathbf{b}_1, \dots, \mathbf{b}_k$ ahead of time. In this case, we really have the matrix problem

$$\boldsymbol{A}\boldsymbol{X} = \boldsymbol{B} \qquad \boldsymbol{B} = \begin{bmatrix} \mathbf{b}_1 & \cdots & \mathbf{b}_k \end{bmatrix}$$

where $\boldsymbol{X}$ is the n by k *matrix* of all k solutions.

Such a scenario arises in a number of places. First, consider actually computing the inverse of a matrix $\boldsymbol{A}$. [1] Then we would set $\boldsymbol{B} = \boldsymbol{I}$, and there are $k = n$ vectors $\mathbf{b}$ all known.

Another, more realistic scenario, arises in block Gaussian elimination. Suppose we are solving

$$\boldsymbol{A}\mathbf{b} = \mathbf{b} \text{ where we have the partition } \begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{A}_2 \\ \boldsymbol{A}_3 & \boldsymbol{A}_4 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}.$$

Then note that if $\boldsymbol{A}_1$ is non-singular, then $\mathbf{x}_1$ must satisfy $\boldsymbol{A}_1\mathbf{x}_1 + \boldsymbol{A}_2\mathbf{x}_1 = \mathbf{b}_1$ or

$$\mathbf{x}_1 = \boldsymbol{A}_1^{-1}\mathbf{b}_1 + \boldsymbol{A}_1^{-1}\boldsymbol{A}_2\mathbf{x}_2.$$

The matrix $\boldsymbol{A}_1^{-1}\boldsymbol{A}_2$ is exactly this type of system.

The simplest way to solve these is just to call \ in Julia or Matlab. This will look at the structure of $\boldsymbol{A}_1$ and choose an appropriate method to solve for all right hand sides simultaneously. It will use multiple threads and processors as appropriate.

In practice, what this will do is compute a factorization of the matrix $\boldsymbol{A}$ and then apply this to all the vectors $\mathbf{b}_1, \dots, \mathbf{b}_k$ at the same time.

In general, for a dense system of linear equations, it takes $O(n^3)$ work to compute a factorization and then $O(n^2)$ work to solve a system with the factors. This gives an overall runtime of $O(n^2k + n^3)$, which is $O(n^3)$ if $k \le O(n)$ and *more interseting* if $k \ge O(n)$.

As an example where the latter scenario arises, consider the partiton above where $\boldsymbol{A}_1$ is 16×16 and n is 1024.

[1] Aside, you shouldn't generally do this! It's a good way to fail the class if you do this without careful consideration of the alternatives.

27.2 MULTIPLE RIGHT HAND SIDES DETERMINED SEQUENTIALLY

The second setting for multiple right hand sides is that we have

$$\boldsymbol{A}\mathbf{x}_1 = \mathbf{b}_1$$

which determines $\mathbf{b}_2$, so $\mathbf{b}_2$ is unknown until we have solved $\mathbf{x}_1$. Then we must solve

$$\boldsymbol{A}\mathbf{x}_2 = \mathbf{b}_2$$

$$\boldsymbol{A}\mathbf{x}_3 = \mathbf{b}_3 = \text{function of } \mathbf{x}_2.$$

$$\text{and so on...}$$

The key is that the matrix $\boldsymbol{A}$ is fixed, which is a scenario that arises in

- the inverse power method for eigenvalues
- backward Euler for linear ODEs.

INVERSE POWER METHOD Recall the power method for dominant eigenvalue, eigenvector pair of a matrix

$$\mathbf{x}^{(0)} = \text{ arbitrary} \qquad \mathbf{x}^{(k+1)} = \rho_k \boldsymbol{A}\mathbf{x}^{(k)} \quad \rho_k = \frac{1}{\|\boldsymbol{A}\mathbf{x}^{(k)}\|}.$$

If the largest magnitude eigenvalue of $\boldsymbol{A}$ is unique, then $\mathbf{x}^{(k)}$ will converge towards the associated eigenvector. The inverse power method simply runs this iteration on $\boldsymbol{A}^{-1}$ instead (assuming $\boldsymbol{A}$ is non-singular). For instance, if $\boldsymbol{A}$ is symmetric positive definite, then the inverse power method will converge to the smallest eigenvalue of $\boldsymbol{A}$. Here, we have exactly this type of setting where $\mathbf{b}^{(k+1)} = \rho_k \mathbf{x}^{(k)}$.[2]

BACKWARD EULER FOR A LINEAR ODE XXX-TODO-XXX

THE FACTORIZATION SOLUTION The best way to approach these problems is to factorize your linear system $\boldsymbol{A}$ via Cholesky, LU, or QR. These are be done on $O(n^3)$ work and then each solve is $O(n^2)$ time. This approach also allows us to exploit structure in the matrix $\boldsymbol{A}$ that may not exist in the inverse $\boldsymbol{A}^{-1}$ to make things go faster.

THE JULIA CODE Julia includes a number of awesome routines to work with matrix factorizations *like* the original matrix. For instance,

```
F = lufact(A) # produces a factorization object F
F \ y # solves Ax = y using the LU factorizations without recomputing it.
```

So we could implement the inverse power method as follows

[2] Again, note that the way to do this isn't to compute $\boldsymbol{A}^{-1}$ and use that instead of $\boldsymbol{A}$! That's another good way to fail the class.

```julia
function invpower(A::Matrix)
  x = normalize!(randn(size(A,1)))
  F = lufact(A)
  for iter = 1:maxiter
    x = normalize!(F\x)
  end
  return x
end
```

PRECONDITIONING 28

The notes here are my own, based on Golub and van Loan, Trefethen, and Saad's textbooks, respectively.

Preconditioning is the process of taking a given linear system:

$$\boldsymbol{A}\mathbf{x} = \mathbf{b}$$

and turning it into a new linear system (with $\boldsymbol{B}$ non-singular):

$$\boldsymbol{B}\mathbf{y} = \mathbf{c}$$

such that it's "easy" to find $\mathbf{x}$ from $\mathbf{y}$ and

$$\text{an iterative method for } \boldsymbol{B}\mathbf{y} = \mathbf{c} \text{ is } \left\{ \begin{matrix} \text{faster} \\ \text{more accurate} \\ \text{better behaved} \\ \text{convergent} \\ \text{easier}, \ldots \end{matrix} \right\}.$$

THE STANDARD PRECONDITIONER. The standard goal with preconditioning is to make an iterative method for $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ go faster. Typically this is done by taking a non-singular matrix $\boldsymbol{M}$ and looking at the linear system: [1]

$$\boldsymbol{M}\boldsymbol{A}\mathbf{x} = \boldsymbol{M}\mathbf{b}.$$

The standard idea is that $\boldsymbol{M}\boldsymbol{A} \approx \boldsymbol{I}$, and we'll see how to make this idea precise shortly. *Also*, we need a *fast* way to *create* $\boldsymbol{M}$, and to *multiply* $\boldsymbol{M}$ by a vector. While this seems like an easy task, *many* preconditioners involve solving a system, hence, $\boldsymbol{M} = \boldsymbol{P}^{-1}$ for some matrix $\boldsymbol{P}$ (which could also be called a preconditioner!). Thus, just multiplying by $\boldsymbol{M}$ can be expensive itself.

Quiz Why do we need $\boldsymbol{M}$ to be non-singular?

QUESTION 1 (THE FUNDAMENTAL QUESTION IN PRECONDITIONING) Thus, we arise at the fundamental question. Given $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, how do I pick $\boldsymbol{M}$ or $\boldsymbol{P}$ such that I actually make the iterative method faster? ◆

Some thoughts on preconditioning

THERE IS NO UNIVERSAL PRECONDITIONER. A great open problem is to find a preconditioning strategy that works for all matrices $\boldsymbol{A}$. Recently, there has been some work on how to do this for symmetric, diagonally dominant linear systems;[2]

[1] So in this case $\boldsymbol{B} = \boldsymbol{M}\boldsymbol{A}$, $\mathbf{x} = \mathbf{y}$ and $\mathbf{c} = \boldsymbol{M}\mathbf{b}$.

[2] This is the celebrated Spielman and Teng nearly-linear time solver for SDD systems. The current runtime is $O(\text{nnz}\sqrt{\log n})$ in theory, which means that it's faster to solve $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ with a SDD matrix than it is to sort a vector. It's currently unknown how to extend that work to symmetric, positive definite systems, however.

PRECONDITIONING IS MORE ART THAN SCIENCE. As you might then expect, much of preconditioning is based on well-informed heuristic procedures. These are ideas that are theoretically grounded, but often make a *leap*. Some leaps are more effective than others!

WHEN POSSIBLE, PRECONDITION THE PROBLEM, NOT THE MATRIX. Suppose that our problem $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ arises from a physics-based application or a complex engineered system. The problem that we want to solve gives rise to some matrix $\boldsymbol{A}$ and some right hand side $\mathbf{b}$. While we could *study* the matrix $\boldsymbol{A}$ and attempt to use a matrix-based preconditioner on $\boldsymbol{A}$, it is often a better strategy to attempt to decompose your problem as:

$\boldsymbol{A}$ = approximation with analytical solution given a right hand-side+correction.

In which case, we really have:

$$\boldsymbol{A} = \underbrace{\boldsymbol{S}}_{\text{simple}} + \underbrace{\boldsymbol{C}}_{\text{correction}}$$

and $\boldsymbol{M} = \boldsymbol{S}^{-1}$ is a good preconditioner because

$$\boldsymbol{S}^{-1}\boldsymbol{A} = \boldsymbol{I} + \boldsymbol{S}^{-1}\boldsymbol{C}.$$

28.1 A MORE FORMAL TREATMENT.

The following theorem justifies why $\boldsymbol{S}^{-1}$ would be a good preconditioner.

Theorem 28.1 (Golub and van Loan, 3rd edition, 10.2.5)
[3] If $\boldsymbol{A} = \boldsymbol{I} + \boldsymbol{B}$ is an n-by-n symmetric postive definite matrix and $\operatorname{rank}(\boldsymbol{B}) = r$, then Krylov methods converge in at most $r + 1$ iterations.

PROOF This is a standard proof strategy. We show that in at most $r + 1$ iterations, the Krylov space $\mathbb{K}_{r+1}(\boldsymbol{A}, \mathbf{b})$ contains the solution $\mathbf{x}$. To do so, note that:

$$\begin{aligned}\mathbb{K}_k(\boldsymbol{A}, \mathbf{b}) &= \operatorname{span}(\mathbf{b}, \boldsymbol{A}\mathbf{b}, \ldots, \boldsymbol{A}^{k-1}\mathbf{b}) \\ &= \operatorname{span}(\mathbf{b}, (\boldsymbol{I}+\boldsymbol{B})\mathbf{b}, (\boldsymbol{I}+\boldsymbol{B})^2 \ldots, (\boldsymbol{I}+\boldsymbol{B})^{k-1}\mathbf{b}) \\ &= \operatorname{span}(\mathbf{b}, \boldsymbol{B}\mathbf{b}, \boldsymbol{B}^2\mathbf{b}, \ldots, \boldsymbol{B}^{k-1}\mathbf{b}).\end{aligned}$$

Because $\boldsymbol{B}$ has rank r, we know that $\boldsymbol{B}^r$ has some polynomial expression in lower powers[4]; thus, the Krylov subspace terminates at this step and we know the space must contain the solution. Because of the optimality properties, any Krylov method will terminate in $r + 1$ steps in exact arithmetic. ■

[3] In the third edition, they split this into 11.3.1 and 11.3.2.

[4] This is a corrolary of the Cayley-Hamilton theorem, among other facts.

More generally speaking, we have the following theorem on the convergence of CG.

Theorem 28.2 (Trefethen 38.5)
Let the CG iteration be applied to a symmetric positive definite linear system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$, where $\boldsymbol{A}$ has 2-norm condition number κ. Then there is a norm $\|\mathbf{z}\|_*$ where

$$\|\mathbf{x} - \mathbf{x}^{(k)}\|_* \leq 2\|\mathbf{x} - \mathbf{x}^{(0)}\|_* \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k.$$

This gives rise to a *linear convergence theorem* that depends on the condition number of a matrix:

$$\|\mathbf{x} - \mathbf{x}^{(k)}\|_* = O(\rho^k)$$

where ρ depends on κ.

Quiz What is $\kappa(\boldsymbol{I})$?

Suppose $\kappa(\boldsymbol{A})$ is big (like one hundred million), then what happens? We get $\rho \approx 1$ (like 0.99999999).

Suppose $\kappa(\boldsymbol{A})$ is nearly 1 (like 16), then what happens? We get $\rho \approx 0$ (like $3/5$).

So given any linear system, if we take $\boldsymbol{M} = \boldsymbol{A}^{-1}$, we will converge in one step. But, computing $\boldsymbol{M}^{-1}\mathbf{x}$ is just as expensive as our original problem. So we want something cheaper.

28.2 Designing a preconditioner

The above theorems motivate three different types of preconditioners:

1. Find a matrix $\boldsymbol{P}$ where $\boldsymbol{P}^{-1}$ is a fast operator and $\boldsymbol{P}^{-1}\boldsymbol{A} \approx \boldsymbol{I}$, i.e. $\kappa(\boldsymbol{MA}) \ll \kappa(\boldsymbol{A})$.
2. Find a matrix $\boldsymbol{P}$ where $\boldsymbol{P}^{-1}$ is a fast operator and $\boldsymbol{P}^{-1}\boldsymbol{A} = \boldsymbol{I} +$ low-rank.
3. Find a matrix $\boldsymbol{P}$ where $\boldsymbol{P}^{-1}$ is a fast operator and $\boldsymbol{P}^{-1}\boldsymbol{A} =$ has few eigenvalues.

In all cases we need $\boldsymbol{P}$ to be something that is easy to find as well.

Quiz Why do we get the 3rd type of preconditioner? (This is not a simple answer, but does follow from the properties of Krylov subspaces; try showing $\dim(\mathbb{K}_k(\boldsymbol{A}, \mathbf{b})) \leq 2$ when $\boldsymbol{A}$ is diagonalizable with two distinct eigenvalues.)

Some subtleties

Suppose we want to use conjugate gradient. Then we need $\boldsymbol{A}$ to be symmetric positive and definite. Suppose we have a matrix $\boldsymbol{MA}$ where $\boldsymbol{M}$ is fast operator and easy to find. Can we always use CG? No, because

$$\boldsymbol{MA} \neq (\boldsymbol{MA})^T$$

in general.

28.3 TYPES OF PRECONDITIONERS

Thus, we consider four types of preconditioners:

$$\begin{array}{lll}
\textit{Left} & \text{solve} & \underbrace{\boldsymbol{MA}}_{\boldsymbol{B}}\,\mathbf{x} = \underbrace{\boldsymbol{M}\mathbf{b}}_{\mathbf{c}} \\
\textit{Right} & \text{solve} & \underbrace{\boldsymbol{AM}}_{\boldsymbol{B}}\,\underbrace{(\boldsymbol{M}^{-1}\mathbf{x})}_{\mathbf{y}} = \mathbf{b} \\
\textit{Left \& Right} & \text{solve} & \underbrace{\boldsymbol{M}_1\boldsymbol{A}\boldsymbol{M}_2}_{\boldsymbol{B}}\,\underbrace{(\boldsymbol{M}_2^{-1}\mathbf{x})}_{\mathbf{y}} = \underbrace{\boldsymbol{M}_1\mathbf{b}}_{\mathbf{c}} \\
\textit{Symmetric} & \text{solve} & \underbrace{\boldsymbol{MAM}^T}_{\boldsymbol{B}}\,\underbrace{(\boldsymbol{M}^{-T}\mathbf{x})}_{\mathbf{y}} = \underbrace{\boldsymbol{M}\mathbf{b}}_{\mathbf{c}}
\end{array}$$

For the CG case above, we want to use a symmetric preconditioner to preserve symmetry. Often, these are written with $\boldsymbol{C}$:

$$\underbrace{\boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{C}^{-T}}_{\boldsymbol{B}}\mathbf{y} = \boldsymbol{C}^{-1}\mathbf{b} \qquad \mathbf{x} = \boldsymbol{C}^{-1}\mathbf{y}.$$

With the hope that $\boldsymbol{B}$ has a small condition number, or clustered eigenvalues, ...

28.3.1 *Ensuring positive definiteness*

We also need $\boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{C}^{-T}$ to be positive definite when $\boldsymbol{A}$ is. We can insure this by taking $\boldsymbol{C}\boldsymbol{C}^T$ as the Cholesky factorization of any positive definite matrix $\boldsymbol{T}$.

28.3.2 *Optimizing CG*

Once we know we are solving a preconditioned linear system, it's often advantageous to know this in the linear solver. We can rewrite CG optimally to use a preconditioner like in Golub and van Loan (4th edition) 11.5.7.

28.4 EXAMPLES OF PRECONDITIONERS

28.4.1 *Diagonals*

The simplest case of preconditioning is to use the diagonal entries. Let $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{N}$ (be a splitting into the diagonal and off-diagonal terms), then:

$$\boldsymbol{M} = \boldsymbol{D}^{-1}$$

is a preconditioner that makes

$$\boldsymbol{M}\boldsymbol{A} = \boldsymbol{I} + \boldsymbol{D}^{-1}\boldsymbol{N}.$$

Quiz Is it always easy to use a diagonal precondition on a matrix?

Quiz How could you do symmetric diagonal preconditioning?

28.4.2 *Polynomials*

Recall the expansion of $\boldsymbol{A}^{-1}$ as it's Neumann series:[5]

$$(\boldsymbol{I} - \boldsymbol{A})^{-1} = \boldsymbol{I} + \boldsymbol{A} + \boldsymbol{A}^2 + \boldsymbol{A}^3 + \dots.$$

Then we can use a finite truncation as the preconditioner to $\boldsymbol{A}\mathbf{x} = \mathbf{b}$:

$$\boldsymbol{M} \approx \boldsymbol{A}^{-1} = \boldsymbol{I} + (\boldsymbol{I} - \boldsymbol{A}) + (\boldsymbol{I} - \boldsymbol{A})^2 + (\boldsymbol{I} - \boldsymbol{A})^3.$$

28.4.3 *Incomplete factorizations*

Incomplete Cholesky and Incomplete LU are both factorizations:

$$\boldsymbol{A} = \boldsymbol{C}\boldsymbol{C}^T - \boldsymbol{R} \qquad \boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}^T - \boldsymbol{R}$$

that are Cholesky-like and LU-like, but that have a new residual term. We call them incomplete if $\boldsymbol{R}$ has a zero-entry whenever $\boldsymbol{A}$ is non-zero. Thus, these ideas can be used for large sparse systems.

Any symmetric, positive definite matrix with a non-negative inverse (called a Stieltjes matrix) has an incomplete Cholesky factorization as worked out in Golub and van Loan.

28.4.4 *Sparse approximate inverses*

Suppose we want the best tridiagonal preconditioner for a matrix $\boldsymbol{A}$. To find this, we could consider the best approximation of the inverse:

$$\begin{aligned} &\text{minimize} && \|\boldsymbol{I} - \boldsymbol{A}\boldsymbol{M}\| \\ &\text{subject to} && \boldsymbol{M} \text{ is tridiagonal.} \end{aligned}$$

The sparsity structure should be given, so the more general problem is, given sparsity structure matrix $\boldsymbol{S}$:

$$\begin{aligned} &\text{minimize} && \|\boldsymbol{I} - \boldsymbol{A}\boldsymbol{M}\| \\ &\text{subject to} && \boldsymbol{M} \text{ has the same non-zeros as } \boldsymbol{S}. \end{aligned}$$

[5] This is a matrix based on the geometric series: $1+t+t^2+\dots = \frac{1}{1-t}$

Consider the tridiagonal case. We can compute $\boldsymbol{M}$ a column at a time:

$$\text{Let } \boldsymbol{M}\mathbf{e}_i = \mathbf{m}_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \alpha \\ \beta \\ \gamma \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{then} \quad \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \text{ solve } \text{ minimize } \quad \left\| \mathbf{e}_i - \begin{bmatrix} \boldsymbol{A}_{i-1} & \boldsymbol{A}_i & \boldsymbol{A}_{i+1} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \right\| .$$

28.4.5 *Multi-grid*

Recall how we thought about approximating the *problem* as a type of preconditioning. Suppose that $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ arises from a n-by-n discretization of Poisson's equation. This gives us an $n^2 \times n^2$ linear system: $\boldsymbol{A}\mathbf{x} = \mathbf{b}$. Now, what if we had solved Poisson's equation for an $n/2$-by-$n/2$ node discretization instead? This is a continuous equation, so we might hope it's reasonable to guess that simply interpolating the solution would give us a good approximation to $\boldsymbol{A}\mathbf{x} = \mathbf{b}$? But then, we could repeat the same argument and use an $n/4$-by-$n/4$ node discretization, and so on and so forth.

This idea gives rise to a preconditioner called *multi-grid* that is incredible at solving Poisson's equations. Using a multi-grid strategy allows us to solve $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ in time $O(n^2)$ where the system has size $n^2 \times n^2$. This is a linear time algorithm![6]

[6] Demmel's textbook: Applied Numerical Linear Algebra has a nice treatment of this algorithm.

EIGENVALUE ALGORITHMS

VII

EIGENVALUE THEORY 29

29.1 INVARIANT SUBSPACES

29.2 DEGENERATE EIGENVALUES

30 EIGENVALUE ALGORITHMS

In these notes, we will frequently assume that $\boldsymbol{A}$ is symmetric. Some of the methods generalize to non-symmetric matrices (e.g. the power method), but there are often complexities involved. Please consult Golub and van Loan for more details on these cases.

These notes are a collection of Trefethen and Bau (Lecture 28) and my own (later) interpolations. The result on the convergence of the power method is due to Luca Trevisan (`http://theory.stanford.edu/~trevisan/expander-online/lecture03.pdf`) which was pointed out to me by Petros Drineas.

30.1 USEFUL PROPERTIES OF EIGENVALUES.

Lemma 30.1
Let $\boldsymbol{A}$ be a symmetric matrix. If $p(\boldsymbol{A})$ is a polynomial of $\boldsymbol{A}$, then the eigenvectors of $\boldsymbol{A}$ and $p(\boldsymbol{A})$ are the same, but the eigenvalues change.

PROOF Note that $p(\boldsymbol{A}) = c_0\boldsymbol{I} + c_1\boldsymbol{A} + c_2\boldsymbol{A}^2 + \cdots + c_k\boldsymbol{A}^k$ for some coefficients $c_0, \ldots, c_k$. Then $\boldsymbol{A} = \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^T$ is the eigenvalue decomposition and $\boldsymbol{A}^j = \boldsymbol{V}\boldsymbol{D}^j\boldsymbol{V}^T$. Hence $p(\boldsymbol{A}) = \boldsymbol{V}p(\boldsymbol{D})\boldsymbol{V}^T$. ■

(The same proof holds for non-symmetric matrices with the Jordan canonical decomposition instead.)

Lemma 30.2 (Gerschgorin disks, Golub and van Loan Theorem 7.2.1)
Let $\boldsymbol{A}$ be any matrix. Suppose that $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{F}$ where $\boldsymbol{D}$ is diagonal and $\boldsymbol{F}$ has zero diagonal. (So $\boldsymbol{D}$ are the diagonal entries of $\boldsymbol{A}$ and $\boldsymbol{F}$ are all the other entries. Then the eigenvalues of $\boldsymbol{A}$ are contained within the set:

$$\lambda(\boldsymbol{A}) \subseteq \bigcup_{i=1}^{n} G_i$$

where G_i is the ith Gerschgorin disk:

$$G_i = \{z \in \mathbb{C} : |z - D_{i,i}| \leq \sum_j |F_{i,j}|.$$

PROOF Let $\lambda \in \lambda(\boldsymbol{A})$ be any eigenvalue that is not equal to $D_{i,i}$ for any i. Then $\boldsymbol{A}\mathbf{x} = \lambda\mathbf{x}$ yields:

$$(\boldsymbol{D} + \boldsymbol{F} - \lambda I)\mathbf{x} = 0 \Leftrightarrow \mathbf{x} = (\boldsymbol{D} - \lambda\boldsymbol{I})^{-1}\boldsymbol{F}\mathbf{x}.$$

From the latter fact, we have:

$$1 \leq \|(\boldsymbol{D} - \lambda\boldsymbol{I})^{-1}\boldsymbol{F}\|$$

for any sub-multiplicative norm. If this is the ∞-norm, then

$$\|(\boldsymbol{D} - \lambda \boldsymbol{I})^{-1}\boldsymbol{F}\|_\infty = \sum_j \frac{|F_{k,j}|}{|D_{k,k} - \lambda|}$$

for some value k. Or $|D_{k,k} - \lambda| \le \sum_j |F_{k,j}|$ and so λ is in G_k.

Note that if $\lambda = D_{i,i}$ then this result holds immediately. ■

EXAMPLE 30.3 For the Laplacian matrix in Poisson's equation on a 2d mesh, all the Gerschgorin disks intersect and give a bound on the largest eigenvalue of $\boldsymbol{A}$ of 8. ◆

30.2 THE POWER METHOD

Recall that one way to find the largest eigenvalue and associated eigenvector of a matrix is to use the power method

```
Given a starting vector x with norm(x) = 1 and tolerance $\tau$
Iterate
  $\mathbf{y} = \boldsymbol{A}\mathbf{x}$
  $\lambda = \mathbf{x}^T\mathbf{y}$
  $\mathbf{x} = \mathbf{y}/$ norm(y)
Until norm($\boldsymbol{A}\mathbf{x} - \lambda\mathbf{x}$) $\le \tau$ (your tolerance)
```

If the largest magnitude eigenvalue associated with $\boldsymbol{A}$ is unique, then this iteration converges to it.

— TODO – Insert picture of the eigenvalue convergence.

30.2.1 *A useful upper bound on the spectral radius of a symmetric positive definite matrix.*

In many cases, we do not actually need the largest eigenvalue of a matrix itself, but rather, a bound on the spectral radius. That is, we want to compute a value θ such that $\rho(\boldsymbol{A}) \le \theta$. This is easy to do via Gerschgorin disks. However, we also usually want θ to be close to $\rho(\boldsymbol{A})$. Gerschgorin disks can be fairly far away.

EXAMPLE 30.4 Consider the matrix $\boldsymbol{A} = \begin{bmatrix} 0 & \mathbf{e}^T \\ \mathbf{e} & 0 \end{bmatrix}$ where $\mathbf{e}$ has length n. Then the Gerschgorin bound on the largest eigenvalue if n. However, the largest eigenvalue is actually $\sqrt{n}$. ◆

For this reason, we might think of using the power method to approximate the spectral radius. For a symmetric matrix, however, we will always have $|\lambda| \le |\rho(\boldsymbol{A})|$ because the spectral radius is the most extreme point. The following theorem gives an upper bound.

Theorem 30.5 (Trevisan 9.6, Lecture Notes on Graph Partitioning, Expanders and Spectral Methods, 2006) Let $\boldsymbol{M}$ be a symmetric positive semi-definite matrix, then running the power method for k steps from a vector $\mathbf{x}^{(0)}$ with

$x_i^{(0)} = \pm 1$ at random, produces a vector $\mathbf{x}^{(k)}$ with Rayleigh quotient

$$\mathbf{x}^{(k)^T} \boldsymbol{M} \mathbf{x}^{(k)} / \mathbf{x}^{(k)^T} \mathbf{x}^{(k)} \geq \rho(\boldsymbol{A})(1 - \varepsilon) \frac{1}{1 + 4n(1 - \varepsilon)^{2k}}.$$

with probability at least 3/16.

This can be then translated into a useful upper-bound on $\rho(\boldsymbol{A})$ and you can use the tightest value of ε to make the result you want.

30.2.2 *The smallest eigenvalue of a symmetric positive semi-definite matrix.*

The above result gives us an immediate algorithm to find the smallest eigenvalue of a symmetric positive semi-definite matrix. We get an upper-bound $\theta \geq \rho(\boldsymbol{A})$ and run the power method on $\theta \boldsymbol{I} - \boldsymbol{A}$. In this case, we have to adjust the eigenvalue estimate $\lambda = \mathbf{x}^T \mathbf{y}$ to $\lambda = \theta - \mathbf{x}^T \mathbf{y}$ to adjust for the difference.

30.2.3 *The inverse power method.*

There are a variety of ways to use the power method to get other eigenvalues besides the largest. The first is the inverse power method, where we run the power method on $\boldsymbol{A}^{-1}$ itself, which corresponds to the changing one line of the iteration:

$$\mathbf{y} = \boldsymbol{A}\mathbf{x} \quad \rightarrow \quad \boldsymbol{A}\mathbf{y} = \mathbf{x}$$

so that we solve a linear system at each step. With this change, the power method will converge to the *smallest magnitude* eigenvalue of $\boldsymbol{A}$ instead. (That is, the one closest to 0.)

— TODO – Insert picture of the eigenvalue transformation $1/\lambda$

30.2.4 *Targeting a specific eigenvalue.*

If we wish to find an eigenvalue close to a value α, then we can use the iteration:

$$(\boldsymbol{A} - \alpha \boldsymbol{I})\mathbf{y} = \mathbf{x}.$$

The eigenvalues of $\boldsymbol{A}$ that are near α will be close to zero in the matrix $(\boldsymbol{A} - \alpha \boldsymbol{I})$, and so when we use inverse iteration on $(\boldsymbol{A} - \alpha \boldsymbol{I})$ then we will find those as the solutions.

— TODO – Insert picture of the eigenvalue transformation $1/(\lambda - \alpha)$

30.2.5 *Using a folded spectrum to target an eigenvalue.*

If $\boldsymbol{A}$ is large and sparse, then we may not have a good way to solve linear systems with $\boldsymbol{A}$. This is often the case for problems that arise based on data where we do not have well-known and effective preconditioner techniques. In this case, if we are interested in finding the smallest eigenvalue, we can employ an idea called the *folded spectrum*. The idea is that $\boldsymbol{A}^2$ is always a symmetric positive semi-definite matrix whose small eigenvalues are close to 0. If we have an upper-bound on the spectral radius of $\theta^2 \leq \boldsymbol{A}^2$, then we can use the same idea for the smallest eigenvalue of a symmetric positive semi-definite matrix.

This can further be adapted to target an eigenvalue near α by using $(\boldsymbol{A} - \alpha\boldsymbol{I})^2$ to make eigenvalues nearby α close to zero. This approach is called the *folded spectrum* method.

30.2.6 *Finding other eigenvalues via deflation*

We can actually use the power method itself to compute multiple eigenvectors via a procedure called deflation. Let $\mathbf{x}, \lambda$ be an eigenpair of $\boldsymbol{A}$. Then create a householder matrix $\boldsymbol{H}$ with $\mathbf{x}$ as associated vector such that $\boldsymbol{H}\mathbf{x} = \mathbf{e}_1$. The matrix $\boldsymbol{HAH}^T$ has the following structure:

$$\begin{bmatrix} \lambda & 0 \\ 0 & \boldsymbol{B} \end{bmatrix}$$

This is actually what we did way back when we created the SVD!

EXAMPLE 30.6 Here is some example code to demonstrate this.

```
A = randn(5,5)
A = A+A’
## Example of the deflation method
λs,X = eig(A)
eval = rand(1:size(A,2))
x = X[:,eval] # just an
xe1 = zeros(size(A,1))
xe1[1] = norm(x)
v = xe1-x
H = eye(size(A)...) - 2*v*v’/(v’*v)
display(H*A*H’)
display(λs[eval])
```

30.3 SUBSPACE ITERATION AND THE QR ALGORITHM

We can generalize the power method to compute the largest few eigenvalues and vectors.

```
Given a starting orthogonal matrix $X$ and tolerance $\tau$
Iterate
  $Y = AX$
  $\Lambda$ = diagonal elements of $X^T AX$ arranged as a diagonal matrix
  [$X, R$] = qr($Y$)
Until norm($AX - X\Lambda$) $\leq \tau$ (your tolerance)
```

Again, if the matrix is symmetric, and these large eigenvalues are unique, we can show that this converges. This method is also called the block power method or the orthogonal iteration.

30.3.1 *Subspace iteration*

Except that when this is usually done, we want to do it for all eigenvalues and vectors, and start with the identity matrix.

```
Given tolerance $\tau$
Set $X = I$
Iterate
  $Y = AX$
  $\Lambda =$ diagonal elements of $X^T AX$ arranged as a diagonal matrix
  $[X, R]$ = qr($Y$)
Until norm($AX - X\Lambda$) $\le \tau$ (your tolerance)
```

EXAMPLE 30.7 Here's one sample of the subspace method converging

```
A = randn(5,5)
A = A+A'

X = eye(size(A,1),size(A,2))
D = diagm(sort(diag(X'*A*X)))
for i = 1:200
  Y = A*X
  D = diagm(sort(diag(X'*Y)))
  X,R = qr(Y)
end
@show [diag(D) eigvals(A)]
```

30.3.2 *The QR iteration*

Here's another way you'll read about to compute eigenvalues and eigenvectors and it's called the QR iteration.

```
Given tolerance $\tau$
$X = I$
Iterate
  $Q, R$ = qr($A$)
  $A = RQ$.
  $\Lambda =$ diagonal elements of $A$ arranged as a diagonal matrix
  $X = XQ$
Until norm($AX - X\Lambda$) $\le \tau$ (your tolerance)
```

When I saw this algorihtm, I found it truly strange! Why would you take the product of a QR factorization in the reverse order? The following important note gives some intuition for what is going on:

Important note. $A = QR$ implies that $R = Q^T A$. So $A_{\text{next}} = Q^T AQ$. Essentially, we are multiplying by an orthogonal matrix on the left and the right.

EXAMPLE 30.8 Here's one sample of the QR iteration converging

```
X = eye(size(A,1),size(A,2))
A = copy(Ainit)
for i = 1:100
  Q,R = qr(A)
  A = R*Q
  D = diagm(sort(diag(A)))
  X = X*Q
end
@show [diag(D) eigvals(Ainit)]
```

30.3.3 *QR and Subspace Iteration are equivalent*

Here, we'll consider the following two methods and show that they are equivalent.

Subspace iteration

```
$X_0 = I$
for $i$ = 1, ...
  $X_i, R_i$ = qr($AX_{i-1}$)
```

QR iteration

```
$A_1 = A$
for $i$ = 1, ...
  $Q_i, R_i$ = qr($A_i$)
  $A_{i+1} = R_i Q_i$
```

Lemma 30.9
$\boldsymbol{X}_i = \boldsymbol{Q}_1 \boldsymbol{Q}_2 \cdots \boldsymbol{Q}_i$

PROOF We'll prove this by induction. The key insight is that $\boldsymbol{R}_i$ is actually the same matrix. (Technical note, we need to assume that the signs of the diagonal elements of $\boldsymbol{R}_i$ are taken to be positive to get a unique $\boldsymbol{Q}$ factor in the QR decomposition.)

Base case. $\boldsymbol{X}_1 = \text{qr}_Q(\boldsymbol{AX}_{i-1}) = \text{qr}(\boldsymbol{A})$ and also $\boldsymbol{Q}_1 = \text{qr}_Q(\boldsymbol{A})$ as well. Because $\boldsymbol{R}_i = \boldsymbol{Q}_1^T\boldsymbol{A} = \boldsymbol{X}_1^T\boldsymbol{A}$, we get the same $\boldsymbol{R}_i$ factor here.

Inductive hypothesis. We assume that $\boldsymbol{X}_i = \boldsymbol{Q}_1\boldsymbol{Q}_2 \cdots \boldsymbol{Q}_i$.

Important note. We saw this before, but $\boldsymbol{A}_i = \boldsymbol{Q}_i\boldsymbol{R}_i$ implies that $\boldsymbol{R}_i = \boldsymbol{Q}_i^T\boldsymbol{A}_i$. So $\boldsymbol{A}_{i+1} = \boldsymbol{Q}_i^T\boldsymbol{A}_i\boldsymbol{Q}_i$. If we iterate this, then note that

$$\boldsymbol{A}_{i+1} = \boldsymbol{Q}_i^T\boldsymbol{A}_i\boldsymbol{Q}_i = \boldsymbol{Q}_i^T \cdots \boldsymbol{Q}_1^T\boldsymbol{A}\boldsymbol{Q}_1 \cdots \boldsymbol{Q}_i. \qquad \blacksquare$$

By our induction hypotheis $\boldsymbol{A}_{i+1} = \boldsymbol{X}_i^T\boldsymbol{A}\boldsymbol{X}_i$.

Note that $\boldsymbol{X}_{i+1}\boldsymbol{R}_{i+1} = \text{qr}(\boldsymbol{AX}_i) = \text{qr}(\boldsymbol{AQ}_1 \cdots \boldsymbol{Q}_i)$.

So $\boldsymbol{X}_{i+1}^T\boldsymbol{AQ}_1 \cdots \boldsymbol{Q}_i = \boldsymbol{R}_{i+1}$.

30.3.4 *The issue with QR*

The problem with the QR method is that it is very expensive. Each iteration is $O(n^3)$.

In order to make this more efficient, we want to translate the matrix $\boldsymbol{A}$ into one that has a structure that makes the RQ-iteration efficient. The properties we need are:

1. we can translate a general symmetric matrix $\boldsymbol{A}$ to a matrix $\boldsymbol{B}$ with this property via orthogonal transformations $\boldsymbol{B} = \boldsymbol{Q}^T\boldsymbol{AQ}$ or similarity transformations $\boldsymbol{B} = \boldsymbol{X}^{-1}\boldsymbol{AX}$ (so we preserve eigenvalues)
2. computing a QR factorization of $\boldsymbol{B}$ is efficient
3. computing the product $\boldsymbol{RQ}$ is efficient
4. the matrix $\boldsymbol{RQ}$ has the same property as $\boldsymbol{B}$.

There are some modestly esoteric classes of matrices that enable these (such as hierarchical semi-seperable), but the most straightforward case is a tridiagonal matrix.

Here, we show that if $\boldsymbol{B}$ is tridiagonal, then $\boldsymbol{RQ}$ is also tridiagonal. This gives us most of the properties since we always saw how tridiagonal matrices enable a linear-time QR factorization. (In the section on GMRES.)

Consider a symmetric tridiagonal matrix

$$\boldsymbol{B} = \begin{bmatrix} \bullet & \bullet & & & \\ \bullet & \bullet & \bullet & & \\ & \bullet & \bullet & \bullet & \\ & & \bullet & \bullet & \bullet \\ & & & \bullet & \bullet \end{bmatrix}$$

Then in the first step of QR, we are going to "zero" out the 2, 1 entry via a Givens transform. This gives us:

$$\boldsymbol{Q}_1 = \begin{bmatrix} \bullet & \bullet & & & \\ \bullet & \bullet & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \qquad \boldsymbol{Q}_1^T\boldsymbol{B} = \begin{bmatrix} \times & \times & \times & & \\ 0 & \times & \times & & \\ & \bullet & \bullet & \bullet & \\ & & \bullet & \bullet & \bullet \\ & & & \bullet & \bullet \end{bmatrix}$$

$$\boldsymbol{Q}_2 = \begin{bmatrix} 1 & & & & \\ & \bullet & \bullet & & \\ & \bullet & \bullet & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \qquad \boldsymbol{Q}_2^T\boldsymbol{Q}_1^T\boldsymbol{B} = \begin{bmatrix} \bullet & \bullet & \bullet & & \\ 0 & \times & \times & \times & \\ & 0 & \times & \times & \\ & & \bullet & \bullet & \bullet \\ & & & \bullet & \bullet \end{bmatrix}$$

Hence, at the end, we'll have

$$\boldsymbol{Q}_4^T\boldsymbol{Q}_3^T\boldsymbol{Q}_2^T\boldsymbol{Q}_1^T\boldsymbol{B} = \boldsymbol{R} = \begin{bmatrix} \bullet & \bullet & \bullet & & \\ 0 & \bullet & \bullet & \bullet & \\ & 0 & \bullet & \bullet & \bullet \\ & & 0 & \bullet & \bullet \\ & & & 0 & \bullet \end{bmatrix}$$

$$\boldsymbol{Q} = \boldsymbol{Q}_1\boldsymbol{Q}_2\boldsymbol{Q}_3\boldsymbol{Q}_4 = \begin{bmatrix} \bullet & \bullet & & & \\ \bullet & \bullet & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & \\ & \bullet & \bullet & & \\ & \bullet & \bullet & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \cdots$$

So

$$\boldsymbol{RQ} = \underbrace{\begin{bmatrix} \bullet & \bullet & \bullet & & \\ 0 & \bullet & \bullet & \bullet & \\ & 0 & \bullet & \bullet & \bullet \\ & & 0 & \bullet & \bullet \\ & & & 0 & \bullet \end{bmatrix}\begin{bmatrix} \bullet & \bullet & & & \\ \bullet & \bullet & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix}}_{\text{linear combination of columns 1,2}} \boldsymbol{Q}_2\boldsymbol{Q}_3\boldsymbol{Q}_4 = \begin{bmatrix} \times & \times & \bullet & & \\ \times & \times & \bullet & \bullet & \\ & 0 & \bullet & \bullet & \bullet \\ & & 0 & \bullet & \bullet \\ & & & 0 & \bullet \end{bmatrix} \boldsymbol{Q}_2\boldsymbol{Q}_3\boldsymbol{Q}_4$$

$$= \begin{bmatrix} \bullet & \times & \times & & \\ \bullet & \times & \times & \bullet & \\ & \times & \times & \bullet & \bullet \\ & & 0 & \bullet & \bullet \\ & & & 0 & \bullet \end{bmatrix} \boldsymbol{Q}_3\boldsymbol{Q}_4$$

$$= \begin{bmatrix} \bullet & \bullet & \times & \times & \\ \bullet & \bullet & \times & \times & \\ & \bullet & \times & \times & \bullet \\ & & \times & \times & \bullet \\ & & & 0 & \bullet \end{bmatrix} \boldsymbol{Q}_4$$

$$= \begin{bmatrix} \bullet & \bullet & \bullet & \times & \times \\ \bullet & \bullet & \bullet & \times & \times \\ & \bullet & \bullet & \times & \times \\ & & \bullet & \times & \times \\ & & & \times & \times \end{bmatrix}$$

We can keep going with this and we'll find that $\boldsymbol{RQ}$ is upper-Hessenberg.

But, $\boldsymbol{R} = \boldsymbol{Q}^T\boldsymbol{A}$ so $\boldsymbol{RQ} = \boldsymbol{Q}^T\boldsymbol{QQ}$ is symmetric, upper-Hessenberg, i.e. tridiagonal!

30.4 REDUCTION TO TRIDIAGONAL FORM

EXAMPLE 30.10 A = randn(6) ◆

30.5 QR WITH SHIFTS

30.6 EIGENVALUES AND EIGENVECTORS OF SPARSE MATRICES

Run the power method! Then we'll get

30.6.1 *Deflation*

30.6.2 *Subspace iteration*

30.6.3 *Lanczos*

30.6.4 *ARPACK*

EXAMPLE 30.11 This example shows how to use the eigs function in Matlab

```
A = sprandsym(10000,50/10000);
[V,D] = eigs(A);
```

ALGORITHMS FOR THE SVD 31

The entire goal of our class was to help study matrix problems through their structure. Here we will consider matrices that have what we will call "bipartite" structure, following the conventions of a graph theory view on matrices. A more standard name for this structure is "consistently ordered" but that includes quite a few more details.

Where do bipartite matrices arise? The first place is in algorithms for the SVD. Another place is the 2d Laplacian, or any matrix derived from a bipartite graph. The point of this lecture is to look at the relationships between these persepctives.

We have seen algorithms to compute eigenvalues of matrices. Algorithms for the SVD follow from two points of view. Assume without loss of generality that $\boldsymbol{A}$ has more rows than columns.

VIEW 1 The singular values of the matrix $\boldsymbol{A}$ are the eigenvalues of $\boldsymbol{A}^T\boldsymbol{A}$

VIEW 2 . The singular values of the matrix $\boldsymbol{A}$ are the positive eigenvalues of $\boldsymbol{B} = \begin{bmatrix} 0 & \boldsymbol{A} \\ \boldsymbol{A}^T & 0 \end{bmatrix}$.

The matrix $\boldsymbol{B}$ in view 2 is a specific instance of a bipartite matrix!

More generally, the theorem underlying View 2 is

Theorem 31.1
Let $\boldsymbol{B} = \begin{bmatrix} 0 & \boldsymbol{A} \\ \boldsymbol{A}^T & 0 \end{bmatrix}$ and let $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ be the SVD of $\boldsymbol{A}$. Then the eigenvalues of $\boldsymbol{B}$ are $\pm\sigma_i$ along with $m + n - 2n$ additional zeros. Given an eigenvalue $+\sigma_i$ eigenvector $\boldsymbol{B}\mathbf{z} = \sigma\mathbf{z}$ if we partition $\mathbf{z} = \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}$. Then, $\boldsymbol{A}\mathbf{y} = \sigma\mathbf{x}$ is one of the singular vectors and value sets.

PROOF We have $\boldsymbol{B} = \begin{bmatrix} 0 & \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \\ \boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T & 0 \end{bmatrix} = \begin{bmatrix} \boldsymbol{U} & 0 \\ 0 & \boldsymbol{V} \end{bmatrix} \begin{bmatrix} 0 & \boldsymbol{\Sigma} \\ \boldsymbol{\Sigma}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{U} & 0 \\ 0 & \boldsymbol{V} \end{bmatrix}$.

— TODO – Show more of this matrix, including the number of zeros. Then note that there exists a permutation matrix $\boldsymbol{P}$ such that

$$\boldsymbol{P}\begin{bmatrix} 0 & \boldsymbol{\Sigma} \\ \boldsymbol{\Sigma}^T & 0 \end{bmatrix}\boldsymbol{P}^T = \begin{bmatrix} 0 & \sigma_1 & & & \\ \sigma_1 & 0 & & & \\ & & 0 & \sigma_2 & \\ & & \sigma_2 & 0 & \\ & & & & \ddots \end{bmatrix}$$

— TODO – Work out more of this matrix, including the number of zeros.

Further, we have

$$\begin{bmatrix} 0 & \sigma_i \\ \sigma_i & 0 \end{bmatrix} = \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} +\sigma_i & \\ & -\sigma_i \end{bmatrix} \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}.$$

At which point, we are done. ■

This enables us to work with the matrix $\boldsymbol{B}$ instead of $\boldsymbol{A}$. Of course, we do not actually form $\boldsymbol{B}$, rather we work with it implicitly.

Just like eigenvalue algorithms, the first step is to reduce the matrix size via a set of orthogonal operations. For the SVD, we can make this two-sided! This enables us to reduce $\boldsymbol{A}$ to a bidiagonal matrix $\boldsymbol{F}$.

— TODO – New nodes on doing the bidiagonal reduction via a full Householder step on the left, then a partial on the right, and then full on the left. ...

— TODO – Note that we could do lower-bidiagonal too!

We can also see this via the Lanczos perspective. Consider running the Lanczos method on $\boldsymbol{B}$.

EXAMPLE 31.2

```
A =
 -5.0  -5.0   5.0  -3.0
 -3.0  -2.0  -1.0   1.0
  4.0   3.0   0.0  -4.0
  0.0   2.0  -4.0   3.0
 -5.0   0.0  -3.0  -1.0
  5.0   2.0   2.0  -2.0


U,T,rho = lanczos(A, [ones(6); zeros(4)], 4)

U =
10 x 5 Array{Float64,2}:
 0.408248   0.0         0.558726    0.0
0.446931
 0.408248   0.0        -0.0423861   0.0
0.410261
 0.408248   0.0        -0.0192664   0.0         -0.687834
 0.408248   0.0        -0.5279      0.0
0.214123
 0.408248   0.0         0.466248    0.0         -0.332179
 0.408248   0.0        -0.435421    0.0         -0.0513024
 0.0       -0.549442    0.0        -0.540403
0.0
 0.0        0.0         0.0        -0.636057
0.0
 0.0       -0.137361    0.0         0.47354
0.0
 0.0       -0.824163    0.0         0.281345
0.0

T =
 0.0       2.97209
 2.97209  0.0       5.94127
         5.94127  0.0       7.37874
```

```
                    7.37874   0.0
```

There is a large amount of structure that emerges! Let's decode this structure!

APPLICATION DERIVATIONS

VIII

32 GRAPH CONVOLUTIONAL NETWORKS

Let G be a graph with neighborhood sets $N(i)$ for all vertices i. Then a graph convolutional network, in the simplest form, is a machine learning algorithm that seeks to identify a set of weight matrices $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k$ to enable prediction of values associated with the nodes of the network. There are many variations. To be concrete, we will describe one example before giving a more general picture. Let $d_i = |N(i)|$ be the degree of each node. Let $\mathbf{x}_i$ be a vector of data associated with each node called the initial embedding. The values of $\mathbf{x}_i$ could be determined by features or could be randomly initialized.

The approach, in words, is as follows:

As the first layer, multiply the initial embedding $\mathbf{x}_i$ by a matrix $\boldsymbol{W}_1$. Then each node sums up these modified vectors from its neighbors, in a degree-weighted fashion, and we apply a ReLU nonlinear function.

At the second layer, multiply the embeddings from the first layer by a matrix $\boldsymbol{W}_2$, and then each nodes again sums up the modified vectors from its neighbors in the same degree weighted fashion.

At the final layer, we apply a log-soft-max function to predict a single class.

Algebraically, this corresponds to the following procedure

$$\begin{aligned}\boldsymbol{\ell}_i^{(1)} &= \operatorname{ReLU}\left(\textstyle\sum_{j \in N(i)} \frac{1}{\sqrt{d_i d_j}} \boldsymbol{W}_1 \mathbf{x}_j\right)\\ \boldsymbol{\ell}_i^{(2)} &= \textstyle\sum_{j \in N(i)} \frac{1}{\sqrt{d_i d_j}} \boldsymbol{W}_2 \boldsymbol{\ell}_j^{(1)}\\ \mathbf{y}_i &= \operatorname{LogSoftMax}(\boldsymbol{\ell}_i^{(2)})\end{aligned}$$

The LogSoftMax function takes an arbitrary vector of data and converts it into log probabilities.

$$\operatorname{LogSoftMax}(\mathbf{x}) = \left[\log\left(\frac{\exp(x_1)}{\exp(\mathbf{e}^T\mathbf{x})}\right) \quad \log\left(\frac{\exp(x_2)}{\exp(\mathbf{e}^T\mathbf{x})}\right) \quad \cdots \quad \log\left(\frac{\exp(x_n)}{\exp(\mathbf{e}^T\mathbf{x})}\right)\right]^T.$$

The SoftMax function makes data more "indicator vector like", for example, it moves the vector $[1, 2, 8]$ to the vector $[0.001, 0.002, 0.997]$. The LogSoftMax function just takes the log of the result to get a value less than 0. There are many simplifications of this function (e.g. using LogSumExp routines).

The idea with graph convolutional networks is that we take this scenario and generalize it!

- We can make the initial embeddings trainable parameters.
- We can add more layers.
- We can make the layers more complicated functions.

- We can embed invariances directly into the computation.

However, for the sake of simplicity, when we generalize by simply adding layers, we just get a sequence of matrices for the parameters:

$$\boldsymbol{W}_1, \boldsymbol{W}_2, \ldots,$$

which become the variables of our optimization problem.

TODO: Cook up some simple problem with a GCN.

EXERCISES

1. Suppose that the initial embeddings or features matrix $\boldsymbol{X}$ are non-negative. Suppose also that we always produce non-negative matrices $\boldsymbol{W}_i$ in a simple two layer GCN. Simplify the resulting algorithm.
2.

33 ADAPTIVE NOISE CANCELLATION

Learning objectives

1. See where Toeplitz and Hankel matrices arise in a simplified explanation of adaptive noise cancellation.

Adaptive noise cancellation for audio arose out of the following goal: how do we ensure that pilots can hear radio communications? Airplanes are *loud*, even on the inside. One solution is just to increase the volume of a set of headphones so that the audio is louder than the ambient noise. This solution has the downside that the volumes involved often cause damage to our fragile ears.

Another solution is to attempt to adjust the audio output so that what we hear is a more faithful representation of what we should be hearing. Let $x(t)$ be the source audio signal. While in mathematical setting we can think of this as a continuous time function, in all practical scenarios, $x(t)$ is a discrete time representation of that continuous signal and uses equally spaced timestamps. This is the case for essentially all audio files on a computer or transmitted through a digital medium. Consequently, let

$$\mathbf{x} = \begin{bmatrix} x(0) & x(h) & x(2h) & x(3h) & \ldots x((N-1)h) \end{bmatrix}^T$$

be a set of discrete time samples from our signal $x(t)$. We have chosen to represent this as a column vector following our conventions. Here is an example of a real signal.

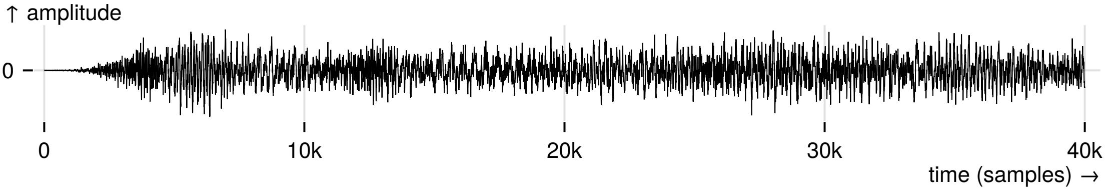


FIGURE 33.1 – A set of 40000 samples from the start of Beethoven's 5th Symphony, Da-Da-Da-Dum.

Assume we are in a noisy room and have two microphones. The first is measuring the room noise itself. The second is measuring a signal nearby the speaker in the ear. This gives rise to a number of distinct signals:[1]

x	the source audio signal we want to transmit to the ear
r	the room noise or reference signal from a microphone
e	the measurement from a microphone nearby the ear
d	the disturbance between the source and what we measure
f	the filter we are applying to correct the noise

Suppose $\mathbf{f} = 0$ (there is no correction), then we have

$$\mathbf{e} = \mathbf{x} + \mathbf{d}$$

[1] These have different symbols in many signal processing setups: **d** is the desired or source signal, **e** is the error signal, which matches our **e** signal, **x** is the reference signal, **y** is the filter. Maybe I should correct it.

and we assume that **d** is some function of **r**. This is easy to do if we assume that **d** is just a convolution of **r** with a simple filter **p**

$$d_i = \sum_{\ell=0}^{L} r_{i-\ell} p_{\ell}.$$

The vector **p** can model a number of acoustical behaviors. For instance, it's very likely that the microphone has a delay of about 1 millisecond. That's 441 samples at a standard sampling rate of 44100 samples per second (which is used in the Bluetooth standard for music). Consequently, we may need to look back over 441 samples in order to adjust for the appropriate delay.

In general, we won't know **p**. *This filter is something we'll need to find to do the noise cancellation.*

Since we can measure **r** and we can measure **d** (which is just **e** − **x**), then we can setup a problem to find **p**

$$\begin{aligned} d_L &= r_1 p_{L-1} + r_2 p_{L-2} + \cdots + r_{L-1} p_1 + r_L p_0 \\ d_{L+1} &= r_2 p_{L-1} + r_3 p_{L-2} + \cdots + r_L p_1 + r_{L+1} p_0 \\ &\vdots \end{aligned}$$

This setup gives us a set of linear equations

$$\mathbf{d}^{-} = \boldsymbol{R}\mathbf{p} \qquad \mathbf{p} = \begin{bmatrix} p_0 & \cdots & p_{L-1} \end{bmatrix}$$

where $\mathbf{d}^{-}$ just removes the first $L - 1$ values from the vector, that is, $\mathbf{d}^{-} = \begin{bmatrix} d_L & \ldots & d_N \end{bmatrix}^T$. The matrix $\boldsymbol{R}$ is

$$\boldsymbol{R} = \begin{bmatrix} r_L & r_{L-1} & \cdots & r_2 & r_1 \\ r_{L+1} & r_L & \cdots & r_3 & r_2 \\ \vdots & & & & \end{bmatrix}.$$

This is a right-shift matrix (Toeplitz) since every row is a left-shift of the previous row.[2]

Since all this setup is approximate, we do not expect this equation to hold with equality. Consequently, we seek a least squares measurement

$$\underset{\mathbf{p}}{\text{minimize}} \quad \|\boldsymbol{R}\mathbf{p} - \mathbf{d}^{-}\| \ .$$

This uses the data from the room or reference microphone to build **r** and then $\boldsymbol{R}$. It uses the difference of **x** and **e** to get **d** and then $\mathbf{d}^{-}$.

Now, the idea is to use our knowledge of **p** to create **f** such that if the speaker plays **x** + **f** then we'll hear **e** = **x** at the speaker. It seems tempting to just try and set **f** = −**d** but remember we don't know **d** until *after* we've played **x**.[3]

The idea then is to use **p** to estimate **d** based on **r** and then use that to create **f**. Note that we can continue to estimate **d** because we know **x**, **e**, **f**. TBD – show an example of this working.

[2] If you arrange **p** from p_{L-1} to p_0 then you will get a left-shift matrix (Hankel) instead.

[3] More generally, there were will slight delays throughout this system.

ADDITIONAL READING

- `https://arxiv.org/pdf/1302.6945`
- `https://maths-people.anu.edu.au/~brent/pd/rpb092i.pdf`
- `https://maths-people.anu.edu.au/brent/pd/rpb143-final.pdf`
- `https://dsp-group.mit.edu/wp-content/uploads/2024/11/Baran-2013-LevinsonLectNotes.pdf` Describes the symmetric Levinson equation from an algebraic point of view.

EXERCISES

1. The matrix $\boldsymbol{R}^T\boldsymbol{R}$ is approximately Toeplitz too. Quantify this approximation and develop an algorithm to find the best Toeplitz approximation.

DERIVATION OF THE HILBERT MATRIX 34

One of the most interesting matrices and matrix problems arose around the turn of the century when David Hilbert wanted to look at approximating functions by polynomials. This leads to a way to quantify how difficult a linear systems is to solve on the computer, called the condition number.

Given a function $f(x)$ we want to create a polynomial approximation $p(x)$ that minimizes the squared error over the interval $[0,1]$. Let $p(x) = c_1 + c_2x + c_3x^2 + \cdots + c_kx^{k-1}$. Then we want to pick $\mathbf{c}$ such that:

$$E(\mathbf{c}) = \int_0^1 (f(x) - p(x))^2 \, dx$$

is as small as possible. If we simply expand the objective function, then we have:

$$E(\mathbf{c}) = \int_0^1 (f(x) - p(x))^2 \, dx = \int_0^1 (f(x) - \textstyle\sum_{j=1}^k c_j x^{j-1})^2 \, dx$$

$$E(\mathbf{c}) = \sum_{j=1}^k \sum_{i=1}^k c_i c_j \int_0^1 x^{j-1}x^{i-1} \, dx - 2\sum_{j=1}^k c_j \int_0^1 f(x)x^{j-1} \, dx + \int_0^1 f(x)^2 \, dx.$$

Hence, we have

$$E(\mathbf{c}) = \mathbf{c}^T \boldsymbol{A}\mathbf{c} - 2\mathbf{c}^T\mathbf{b}$$

where

$$A_{ij} = \int_0^1 x^{j-1}x^{i-1} \, dx = 1/(i + j - 1) \qquad b_j = \int_0^1 f(x)x^{j-1} \, dx.$$

This minimizer of this is just the solution $\boldsymbol{A}\mathbf{c} = \mathbf{b}$.

But the matrix $\boldsymbol{A}$ is surprising. It is called the Hilbert matrix (after David Hilbert) and is one of the most highly sensitive linear systems of equations (see lecture 18).

35 CHESS RANKING

Discuss Vigna's early reference along with Richard Brent's paper.

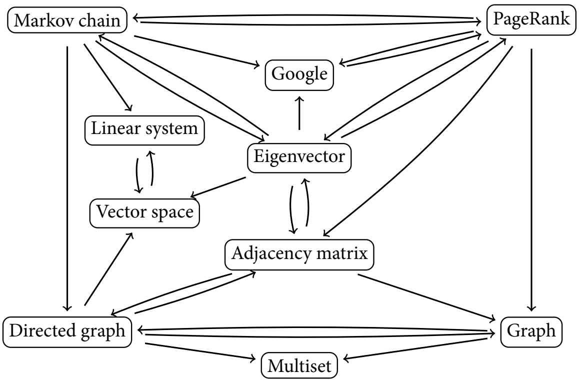
PAGERANK 36

Learning objectives

1. See the derivation of the PageRank linear system of equations.
2. See the relationship between PageRank and an eigenvector problem.

The goal of PageRank is to estimate *the most important nodes in a directed graph*. Methods to address the importance of nodes in a graph are an instance of *centrality computations* in network analysis. PageRank was initially conceived of for this question when the directed graph represents a set of connected web pages on the internet. Consequently, you may often hear a vertex or node called a *page* and a directed edge a *link* in this context.

Let $\boldsymbol{A}$ be the adjacency matrix of a directed graph. An entry of $A_{ij} = 1$ when there is a directed edge from node i to node j and an entry has a value of 0 otherwise. Here is an example where we have extracted a subset of articles from Wikipedia and the directed links among them.[1]



$$\boldsymbol{A} = \begin{array}{c} \\ \left[\begin{array}{cccccccccc} 0&1&1&1&1&0&1&0&0&0\\ 1&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&1&1&1&0&0&0\\ 1&1&0&0&1&1&0&1&0&0\\ 1&1&1&1&0&0&0&0&1&0\\ 0&0&1&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&0&0&0&1\\ 0&0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&0&0 \end{array}\right] \end{array} \begin{array}{l} \text{PageRank}\\ \text{Google}\\ \text{Adjacency matrix}\\ \text{Markov chain}\\ \text{Eigenvector}\\ \text{Directed graph}\\ \text{Graph}\\ \text{Linear system}\\ \text{Vector space}\\ \text{Multiset} \end{array}$$

Columns: PageRank, Google, Adjacency matrix, Markov chain, Eigenvector, Directed graph, Graph, Linear system, Vector space, Multiset

The way that importance works in PageRank is that we model a process[2] that behaves as follows. At node i

- with probability α, we follow an out-edge to another node. (If there are no out-links, then we jump to a page chosen uniformly at random.)
- with probability $(1 - \alpha)$, we jump to a page chosen uniformly at random (a reset or restart behavior).

This process is related to a so called *random surfer model* because it models someone randomly clicking around web pages. (This activity used to be called surfing the web.) At each page, we click a random link. Periodically, we restart the session entirely – like when someone would get up. This idea was a better model for how people engaged with the web in 1998 than it is for the modern social network dominated web and its algorithmic feeds, but let us continue to analyze it anyway.

[1] This graph was originally created around 2013, so the the data may have changed slightly.

[2] Formally, this is a memoryless stochastic process, also called a Markov chain.

Assume that each node has an associated index from 1 to n, the total number of nodes.[3] Let X_t be the identity of the node the process is on at step t. These values are an infinite sequence that represents the behavior. An example is

$$\begin{array}{ccccccccccccccc} X_1 & & X_2 & & X_3 & & X_4 & & X_5 & & X_6 & & X_7 & & \cdots \\ 1 & \rightarrow & 3 & \rightarrow & 5 & \rightarrow & 9 & \textit{reset} & 4 & \rightarrow & 5 & \rightarrow & 4 & \rightarrow & \end{array}$$

We see a reset after the third step. The PageRank vector of the graph is defined as the amount of time that this process spends in each node as it runs forever

$$x_i = \lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} \begin{cases} 1 & X_t = i \\ 0 & \text{otherwise} \end{cases}.$$

Let's explain the sum. For shorthand, note that we can replace the two cases with an indicator function and $Ind[X_t = i]$ is exactly the function with value 1 if $X_t = i$ and 0 otherwise. Let $f_i(T) = \frac{1}{T} \sum_{t=0}^{T} Ind[X_t = i]$. Note that $f_i(T)$ is always a probability distribution over all nodes i. This property occurs because there are exactly T places where $Ind[X_t = i]$ is 1 as we look over the choices of i. Consequently, we have a limit of averages over where the process spends its time.[4] Let $\mathbf{x}$ be the vector formed by the x_is. Then the vector $\mathbf{x}$ must be a probability distribution itself. These distributions are also called *stochastic vectors*.

Definition 36.1 (stochastic vector)
A stochastic vector $\mathbf{v}$ has non-negative entries that sum to 1, that is, $v_i \geq 0$ and $\mathbf{e}^T \mathbf{v} = 0$.

In turns out that $\mathbf{x}$ always exists for any sample of a memoryless stochastic process, but it need not be unique. We won't prove this as it's a lengthy diversion. For the PageRank process it is unique because of the reset steps, although we can show this a slightly different way.

Add a reference for where this can be found.

Since we know that x_i exists, we can attempt to state conditions on what must be true about it. Our strategy here is the same as we used in the hitting time analysis in theorem 3.3. Since we have let the process run for *infinite* time, then the probability of getting to x_i must be able to be deduced from the other probabilities. There are three ways to get to node i

1. from an incoming link,
2. from jumping from a node with no outlinks
3. from a reset jump.

Let d_j be the number of nodes that node j links to (so this is the out-degree of node j). Then

$$x_i = \underbrace{\alpha \sum_{j \text{ links to } i} x_j / d_j}_{\text{incoming links}} + \underbrace{\alpha \sum_{j \text{ with } d_j = 0} x_j / n}_{\text{nodes with no outlinks}} + \underbrace{(1 - \alpha) \sum_{j=1}^{n} 1/n}_{\text{reset jumps}} .$$

[3] For our case, we have the indices

1	PageRank
2	Google
3	Adjacency matrix
4	Markov chain
5	Eigenvector
6	Directed graph
7	Graph
8	Linear system
9	Vector space
10	Multiset

[4] This limit of averages is an instance of what is also called a Cesáro sum.

Note that this is a *linear system* of equations, we now seek to introduce notation to understand it better.

Let $\boldsymbol{P}$ be the matrix where

$$\bar{P}_{i,j} = \begin{cases} 1/d_j & \text{node } j \text{ links to node } i,\, d_j > 0 \\ 0 & \text{otherwise.} \end{cases}$$

If $\boldsymbol{A}$ is the adjacency matrix described above then

$$\bar{\boldsymbol{P}} = \boldsymbol{A}^T \boldsymbol{D}^{-1} \text{ where } D^{-1}_{ij} = \begin{cases} 1/d_i & i = j, d_j > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The matrix $\boldsymbol{D}^{-1}$ is really the inverse of the diagonal matrix of node degrees, where we simply skip over any node with degree 0.[5] Let $\mathbf{c}$ be the indicator vector where $c_i = 1$ for all nodes with $d_i = 0$ and $c_i = 0$ for all other nodes, that is $c_i = Ind[d_i = 0]$. Then we can rewrite the elementwise system as

$$\mathbf{x} = \alpha \bar{\boldsymbol{P}} \mathbf{x} + \alpha \mathbf{e}(\mathbf{c}^T \mathbf{x})/n + (1 - \alpha)\mathbf{e}/n.$$

If the graph underlying the problem is *connected* or all pages have outlinks, then we will have $\mathbf{c} = 0$.[6]

Let us write an example of these equations for the introductory example of Wikipedia

dothis

do this!

36.1 THE PAGERANK LINEAR SYSTEM & SOME NOTATION.

It turns out that we can simplify this setting even more. Note that the way we have defined $\bar{\boldsymbol{P}}$ and $\mathbf{c}$ are coupled. Columns of $\bar{\boldsymbol{P}}$ correspond the outlinks of nodes. For each node without outlines, the corresponding column of $\bar{\boldsymbol{P}}$ is empty. This means that $\mathbf{c}$ really is an indicator vector over the columns of $\bar{\boldsymbol{P}}$ that are empty. Let $\mathbf{v}$ be *any* stochastic vector, which includes the choice above $\mathbf{v} = \mathbf{e}/n$. We can combine these elements into into a single matrix

$$\boldsymbol{P} = \bar{\boldsymbol{P}} + \mathbf{v}\mathbf{c}^T.$$

Each column of this matrix $\boldsymbol{P}$ is now a stochastic vector, which is something we call a *column stochastic matrix.*

Definition 36.2 (column stochastic matrix)
A column stochastic matrix $\boldsymbol{P}$ has the property each column is a stochastic vector, equivalently we have $P_{i,j} \geq 0$ and $\mathbf{e}^T \boldsymbol{P} = \mathbf{e}^T$.

EXAMPLE 36.3 When $\boldsymbol{P} = \bar{\boldsymbol{P}} + \mathbf{v}\mathbf{c}^T$, let's check that it is column stochastic. Since all the quantities are non-negative, we have $P_{i,j} \geq 0$. Now for

[5] This "0-ignoring" inverse is often called the pseudoinverse.

[6] Note that we have $\mathbf{c} = 0$ for the example with the pages from Wikipedia.

$\mathbf{e}^T \boldsymbol{P} = \mathbf{e}^T$, we have

$$\begin{aligned} \mathbf{e}^T(\boldsymbol{A}^T \boldsymbol{D}^{-1} + \mathbf{v}\mathbf{c}^T) &= \underbrace{(\boldsymbol{A}\mathbf{e})^T}_{\text{outdegrees } d_i \text{ or } 0} \boldsymbol{D}^{-1} + (\mathbf{e}^T\mathbf{v})\mathbf{c}^T \\ &= Ind[d_i > 0]^T + Ind[d_i = 0]^T = \mathbf{e}^T. \end{aligned}$$ ◆

Consequently, the PageRank equation above is

$$\mathbf{x} = \alpha \boldsymbol{P}\mathbf{x} + (1 - \alpha)\mathbf{v}$$

where we have substituted $\mathbf{v}$ for the stochastic vector $\mathbf{e}/n$. This is currently written as a fixed point. Usually we write linear systems with a single variable. This results in the following definition of

Definition 36.4 (PageRank)
The PageRank vector $\mathbf{x}$ is defined as the solution of the linear system

$$\mathbf{x} = \alpha \boldsymbol{P}\mathbf{x} + (1 - \alpha)\mathbf{v} \quad \Leftrightarrow \quad (\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x} = (1 - \alpha)\mathbf{v}$$

where $\boldsymbol{P}$ is a column stochastic matrix, $0 < \alpha < 1$, and $\mathbf{v}$ is a stochastic vector.

EXAMPLE 36.5 Let us show from this definition that $\mathbf{x}$ is a stochastic vector. , the tricky part is non-negativitiy. ◆

TODO

36.2 THE PAGERANK EIGENVALUE PROBLEM

Recall the PageRank vector is the solution of the linear system

$$(\boldsymbol{I} - \alpha \boldsymbol{P})\mathbf{x} = (1 - \alpha)\mathbf{v}$$

where $\mathbf{x}$ is a *stochastic vector* where $\mathbf{e}^T\mathbf{x} = 1$. Then we can multiply by just the right value of 1 to write

$$\mathbf{x} = (\alpha \boldsymbol{P} + (1 - \alpha)\mathbf{v}\mathbf{e}^T)\mathbf{x}.$$

This new formulation is an eigenvalue problem instead of a linear system.

36.3 ALGORITHMS FOR PAGERANK

All of the algorithms from this book can be specialized for PageRank. We highlight a few key ideas in the problems below.

EXERCISES

1. Let $\boldsymbol{A}$ be a symmetric matrix for the adjacency matrix of an undirected graph. Show that we can multiply the PageRank vector $\mathbf{x}$

by a diagonal matrix $\boldsymbol{B}$ to find $\mathbf{y} = \boldsymbol{B}\mathbf{x}$ where there is a symmetric positive definite linear system that we can solve to find $\mathbf{y}$.

2. Develop an algorithm for coordinate descent on the PageRank linear system.
3. Compare sequential and parallel variable updates on a binomial random graph. These binomial random graphs can be generated easily using `sprand` in Julia and Matlab.

```julia
using SparseArrays
"""
    binomial_graph(n,d)

Generate a binomial random graph, also called an Erdos Renyi graph,
with n vertices and average degree d. The return value is a sparse
adjacency matrix. If d > O(log(n)), for a small constant,
then the graph is connected with high probability. Often
d > 2 log(n) is sufficient.
"""
function binomial_graph(n::Integer,d::Real)
  A = sprand(n, d/n)
  map!(x->x > 0 ? 1.0 : 0, A.nzval)
  return A
end
```

4.
5.

TENSOR FACTORIZATION & MATRIX MULTIPLICATION 37

Learning objectives

1. Recognize fast matrix multiplication algorithms as a tensor factorization.

Consider the product of two matrices

$$C = AB \text{ where } C_{ij} = \sum_k A_{ik}B_{kj}.$$

For simplicity in this lecture, we will work with $n \times n$ matrices throughout. Matrix multiplication as expressed takes n^3 multiplication operations. In 1969, we made a discovery that matrix multiplication can be done in fewer than n^3 multiplications [Strassen, 1969]. To see how, let's consider the 2×2 case. This may seem a strange way to start, but the simple 2×2 case will enable us to generalize to a block 2×2 case and a recursive algorithm.

For this 2×2 case, let

$$\boldsymbol{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \quad \boldsymbol{C} = \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$
$$w = ae + bg \quad x = af + bh \quad y = ce + dg \quad z = cf + dh.$$

We clearly see the 8 multiplications $ae, bg, af, bh, ce, dg, cf, dh$. Strassen [1969] showed that we can compute seven intermediate products with only seven multiplications

$$\begin{aligned} m_1 &= (a+d)(e+h) \\ m_2 &= (c+d)e \\ m_3 &= a(f-h) \\ m_4 &= d(-e+h) \\ m_5 &= (a+b)h \\ m_6 &= (-a+c)(e+f) \\ m_7 &= (b-d)(g+h) \end{aligned}$$

and then

$$\begin{aligned} w &= m_1 + m_4 - m_5 + m_7 \\ x &= m_3 + m_5 \\ y &= m_2 + m_4 \\ z &= m_1 - m_2 + m_3 + m_6. \end{aligned}$$

This method involves more additions, but fewer multiplications! When we generalize to a block algorithm, then reducing multiplications is crucial because matrix multiplication is more expensive than matrix addition. We first wish to explain *where this algorithm comes from.*[1]

In one of the most beautiful connections between algorithms and mathematics, it turns out that these multiplications can be encoded into a

[1] This was not in Strassen's original paper, the connection to tensor rank came later via

$4 \times 4 \times 4$ tensor $\underline{\boldsymbol{T}}$. The choice of modes of the tensor is arbitrary and our choice mirrors linear algebra where the first mode is the *output* or *result* and later modes are inputs. Hence, we associate mode 1 with the result $\boldsymbol{C}$, mode 2 with the first input $\boldsymbol{A}$, and mode 3 with the second input $\boldsymbol{B}$.

$$\underset{w}{\underline{\boldsymbol{T}}(1,:,:)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \quad \underset{x}{\underline{\boldsymbol{T}}(2,:,:)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \quad \underset{y}{\underline{\boldsymbol{T}}(3,:,:)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \quad \underset{z}{\underline{\boldsymbol{T}}(4,:,:)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Let $\mathbf{p} = \text{vec}(\boldsymbol{A}^T) = \begin{bmatrix} a & b & c & d \end{bmatrix}^T$, $\mathbf{q} = \text{vec}(\boldsymbol{B}^T) = \begin{bmatrix} e & f & g & h \end{bmatrix}^T$.[2] Then the result $\mathbf{r} = \text{vec}(\boldsymbol{C}^T) = \begin{bmatrix} w & x & y & z \end{bmatrix}$ is the result of a tensor contraction

$$\mathbf{r} = \underline{\boldsymbol{T}} \otimes_2 \mathbf{p} \otimes_3 \mathbf{q}$$

If we study this expression elementwise, we find

$$w = r(1) = \sum_{j=1}^{4} \sum_{k=1}^{4} \underline{\boldsymbol{T}}(1, j, k) p(j) q(k) = ae + bg \text{ as expected.}$$

The power and beauty of this perspective, however, regards the *rank* of this tensor $\underline{\boldsymbol{T}}$. While a 4×4 matrix has a maximum rank of 4, this rank of this simple $4 \times 4 \times 4$ tensor is seven.[3] Since it has 8 non-zeros, and each non-zero corresponds to a rank 1 tensor, we know it has rank less or equal to than 8. Let $\underline{\boldsymbol{E}}_{ijk}$ be a tensor with a 1 in the i, j, k element and zeros elsewhere. Then

$$\underline{\boldsymbol{T}} = \underline{\boldsymbol{E}}_{111} + \underline{\boldsymbol{E}}_{123} + \underline{\boldsymbol{E}}_{212} + \underline{\boldsymbol{E}}_{224} + \underline{\boldsymbol{E}}_{331} + \underline{\boldsymbol{E}}_{343} + \underline{\boldsymbol{E}}_{432} + \underline{\boldsymbol{E}}_{444}$$

or as it should be written in terms of the original variable identities

$$\underline{\boldsymbol{T}} = \underline{\boldsymbol{E}}_{wae} + \underline{\boldsymbol{E}}_{wbg} + \underline{\boldsymbol{E}}_{xaf} + \underline{\boldsymbol{E}}_{xbh} + \underline{\boldsymbol{E}}_{yce} + \underline{\boldsymbol{E}}_{ydg} + \underline{\boldsymbol{E}}_{zcf} + \underline{\boldsymbol{E}}_{zdh}.$$

The second expression is easy to read off as an algorithm: add to w the product of ae, add to w the product of bg, and so on.

The factorization into a sum of 7 rank-1 factors is not stated by Strassen [1969], but is given implicitly by that algorithm. To give the factorization, we are going to use the shorthand notation $\mathbf{e}_a, \mathbf{e}_e, \mathbf{e}_w = \mathbf{e}_1$, $\mathbf{e}_b, \mathbf{e}_f, \mathbf{e}_x = \mathbf{e}_2$, and so on.[4] Then

$$\underline{\boldsymbol{T}} = \underline{\boldsymbol{M}}_1 + \underline{\boldsymbol{M}}_2 + \underline{\boldsymbol{M}}_3 + \underline{\boldsymbol{M}}_4 + \underline{\boldsymbol{M}}_5 + \underline{\boldsymbol{M}}_6 + \underline{\boldsymbol{M}}_7 \text{ where}$$

$$\begin{aligned}
\underline{\boldsymbol{M}}_1 &= (\mathbf{e}_w + \mathbf{e}_z) \otimes (\mathbf{e}_a + \mathbf{e}_d) \otimes (\mathbf{e}_e + \mathbf{e}_h) \\
\underline{\boldsymbol{M}}_2 &= (\mathbf{e}_y - \mathbf{e}_z) \otimes (\mathbf{e}_c + \mathbf{e}_d) \otimes \mathbf{e}_e \\
\underline{\boldsymbol{M}}_3 &= (\mathbf{e}_x + \mathbf{e}_z) \otimes \mathbf{e}_a \otimes (\mathbf{e}_f - \mathbf{e}_h) \\
\underline{\boldsymbol{M}}_4 &= (\mathbf{e}_w + \mathbf{e}_y) \otimes \mathbf{e}_d \otimes (-\mathbf{e}_e + \mathbf{e}_g) \\
\underline{\boldsymbol{M}}_5 &= (-\mathbf{e}_w + \mathbf{e}_x) \otimes (\mathbf{e}_a + \mathbf{e}_b) \otimes \mathbf{e}_h \\
\underline{\boldsymbol{M}}_6 &= \mathbf{e}_z \otimes (-\mathbf{e}_a + \mathbf{e}_c) \otimes (\mathbf{e}_e + \mathbf{e}_f) \\
\underline{\boldsymbol{M}}_7 &= \mathbf{e}_w \otimes (\mathbf{e}_b - \mathbf{e}_d) \otimes (\mathbf{e}_g + \mathbf{e}_h).
\end{aligned}$$

[2] The use of transpose here is avoidable with a reordering of the entries of $\boldsymbol{A}$ and $\boldsymbol{B}$ and $\boldsymbol{C}$.

[3] This is a challenging proof. Hopcroft and Kerr [1969]. Showing this involves what is now called *algebraic geometry*, Landsberg [2006].

Add more here.

[4] Just to be clear, we are going to associate the variable letter a with the index 1, b with the index 2, and so on. This step is a better way since the variable name to order mapping is entirely arbitrary.

This is tedious to verify by hand, so we check it with an implementation.

```
T = matrix_multiplication_tensor((2,2), (2,2), (2,2))
I4 = Diagonal(ones(4)); e1, e2, e3, e4 = eachcol(I4)
ea = e1; eb = e2; ec = e3; ed = e4
ee = e1; ef = e2; eg = e3; eh = e4
ew = e1; ex = e2; ey = e3; ez = e4
# we reverse order and reshape to make the kron work out correctly
tensorprod(s,u,v) = reshape(kron(v,u,s), (length(v), length(u), length(s)))
M1 = tensorprod(ew + ez, ea + ed, ee + eh)
M2 = tensorprod(ey - ez, ec + ed, ee)
M3 = tensorprod(ex + ez, ea, ef - eh)
M4 = tensorprod(ew + ey, ed, -ee + eg)
M5 = tensorprod(-ew + ex, ea + eb, eh)
M6 = tensorprod(ez, -ea + ec, ee + ef)
M7 = tensorprod(ew, eb - ed, eg + eh)
T7 = M1 + M2 + M3 + M4 + M5 + M6 + M7
println("Slices of T7 along the first dimension:")
for idx in axes(T,1)
  println("T7[$idx, :, :] = ")
  for idx1 in axes(T7,2); for idx2 in axes(T7,3);
    print(T7[idx, idx1, idx2], " "); end; println(); end
end
println("norm of T - T7 = ", norm(T-T7));
```

```
Slices of T7 along the first dimension:
T7[1, :, :] =
1.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
T7[2, :, :] =
0.0 1.0 0.0 0.0
0.0 0.0 0.0 1.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
T7[3, :, :] =
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0
T7[4, :, :] =
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0
0.0 0.0 0.0 1.0
norm of T - T7 = 0.0
```

This is exactly the same tensor as before.

Going back to multiplication

"But what does all of this analysis have to do with *matrix multiplication* and can we see something larger than 2 × 2" is what you are probably wondering. The answer is that we can now compute the result $\text{vec}(\boldsymbol{C}^T)$ with only 7 multiplications rather than the 8 in the standard algorithm and this is exactly what Strassen's idea produces. Note that

$$\mathbf{r} = \underline{\boldsymbol{T}} \otimes_2 \mathbf{p} \otimes_3 \mathbf{q} = \underline{\boldsymbol{M}}_1 \otimes_2 \mathbf{p} \otimes_3 \mathbf{q} + \underline{\boldsymbol{M}}_2 \otimes_2 \mathbf{p} \otimes_3 \mathbf{q} + \cdots$$

The first term $\underline{\boldsymbol{M}}_1 \otimes_2 \mathbf{p} \otimes_3 \mathbf{q}$ is

$$\underline{\boldsymbol{M}}_1 \otimes_2 \mathbf{p} \otimes_3 \mathbf{q} = \begin{bmatrix} (a+d)(e+h) \\ 0 \\ 0 \\ (a+d)(e+h) \end{bmatrix}$$

and the second term is

$$\underline{\boldsymbol{M}}_2 \otimes_2 \mathbf{p} \otimes_3 \mathbf{q} = \begin{bmatrix} 0 \\ 0 \\ (c+d)e \\ -(c+d)e \end{bmatrix}.$$

So the rank 7 factorization *produces* the same algorithm as Strassen's descriptions. This means that if there were a rank 6 factorization of this matrix, we'd have an algorithm with even fewer multiplications.

Handling a general matrix

Now, suppose we want to multiply two $n \times n$ matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ into a result $\boldsymbol{R} = \boldsymbol{PQ}$. We simply view them as 2 × 2 block matrices

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} \quad \boldsymbol{Q} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{F} \\ \boldsymbol{G} & \boldsymbol{H} \end{bmatrix} \quad \boldsymbol{R} = \begin{bmatrix} \boldsymbol{W} & \boldsymbol{X} & \boldsymbol{Y} & \boldsymbol{Z} \end{bmatrix}.$$

Now, multiplying $\boldsymbol{AE}$ is an $(n/2) \times (n/2)$ matrix multiplication whereas adding

37.1 AN IMPLEMENTATION OF THIS MATRIX MULTIPLICATION ROUTINE

37.2 THE PROBLEM WITH CANCELLATION

REFERENCES

- https://mathoverflow.net/questions/126164/strassens-algorithm. This is a great discussion going into some of the more exotic details of this result and what is assumed by various authors.

EXERCISES

1. Write out the $4 \times 4 \times 4 \times 4$ tensor that results for the triple multiplication $\boldsymbol{D} = \boldsymbol{ABC}$.
2. Exp

38 MATRIX COMPUTATIONS IN TRANSFORMERS: THE GPT-2 FUNCTION

Learning objectives

1. Appreciate the three major pieces of the transformer architecture: the token embedding, the sequence of transformers and perceptron layers, and the output rounding.
2. Recognize the matrix structure inside a transformer.
3. Run a flop count for a transformer architecture.

We cover the full transformer function for GPT-2 by describing the internal matrix computations. To begin, we make one notation alert.[1]

For this section, vectors are row *vectors!*

We make this choice for two reasons. First, this is standard for transformers. Second, it allows us to model a pipeline as *post-multiplication* by a matrix. Let **v** be a vector, then

$$\mathbf{v}\boldsymbol{T}_1\boldsymbol{T}_2$$

is the result of transforming it by matrix $\boldsymbol{T}_1$ and then $\boldsymbol{T}_2$. (Remember **v** is a row vector.) If we then had a collection of vectors

$$\boldsymbol{V} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_m \end{bmatrix} \text{ we can transform them all via } \boldsymbol{V}\boldsymbol{T}_1\boldsymbol{T}_2.$$

The input to a transformer is a sequence where each element comes from a set $\mathcal{T}$, which is usually called the *vocabulary* or *token dictionary*.[2] The result of a transformer is a probability distribution over $\mathcal{T}$ for each input token.

Since we assume $\mathcal{T}$ is a finite set, we can associate each element with a number from 1 to T, where T is the size of $\mathcal{T}$.[3] An equivalent input is then a sequence of token identifiers. For instance, in the standard byte-pair encoding tokenizer for text used in GPT-2[4],

the string "`Matrices are my friends and`" corresponds to the token sequence $[19044, 45977, 389, 616, 2460, 290]$ where

19044 →	`Ma`
45977 →	`trices`
389 →	` are`
616 →	` my`
2460 →	` friends`
290 →	` and`

The spacing isn't an accident. The token 389 represents the string "` and`" including the space.

38.1 OVERVIEW

The overall GPT-2 function starts with an input matrix $\boldsymbol{X}^{(0)}$ of dimension $m \times T$ build from a one-hot encoding of the tokenization sequence

[1] We really want you to remember, vectors for this section are row vectors!

[2] The ideas originated in natural language processing, so vocabulary originally reflected a set of words. The ideas are now more generally over tokens.

[3] PARAMETER
T is the the number of tokens in our description.

[4] Tokenization is really interesting, but beyond this lecture. Play around more with the tokenzier here: `https://observablehq.com/@simonw/gpt-tokenizer`

$t_1, t_2, \ldots, t_m$.[5] To explain that statement in more detail, let $\mathbf{e}_i$ be the one hot encoding of the token id i, so $\mathbf{e}_i \in \mathbb{R}^T$ with a 1 in the ith position and 0 elsewhere, then

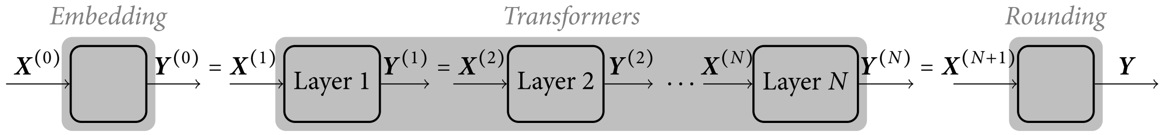
$$\boldsymbol{X}^{(0)} = \begin{bmatrix} \mathbf{e}_{t_1} \\ \mathbf{e}_{t_2} \\ \vdots \\ \mathbf{e}_{t_m} \end{bmatrix} \text{ for example } \boldsymbol{X} = \begin{bmatrix} \mathbf{e}_{19044} \\ \mathbf{e}_{45977} \\ \mathbf{e}_{389} \\ \mathbf{e}_{616} \\ \mathbf{e}_{2460} \\ \mathbf{e}_{290} \end{bmatrix}.$$

The GPT-2 function $\boldsymbol{F}$ maps $\boldsymbol{X}^{(0)} \to \boldsymbol{Y}$ where $\boldsymbol{Y}$ is $m \times T$ and rows of $\boldsymbol{Y}$ are probability distributions over the token set $\mathcal{T}$. Typically, $\boldsymbol{Y}$ is trained so that the *next token* predicted should be the output. So if we want to use this to predict a text completion or *what comes next*, then we'd use the mth row of $\boldsymbol{Y}$ and predict the next token.

The function $\boldsymbol{F}$ has the following form:

$$\boldsymbol{F}(\boldsymbol{X}) = \text{Round}(\text{Transformers}(\text{Embed}(\boldsymbol{X})))$$

Here is a graphical illustration.



The output from each box flows into the input to the next box, hence $\boldsymbol{Y}^{(i)} = \boldsymbol{X}^{(i+1)}$. We will state the structure of each block and cover the dimensions for each.[6]

38.2 THE TOKEN EMBEDDING

The first step of the transformer is that we associate each token in the sequence $[t_1, t_2, \ldots, t_m]$ with a vector in $\mathbb{R}^d$. The choice of d is usually called the embedding dimension and persists throughout the transformer sequence.[7] We can describe this in two ways. First, let $\mathbf{u}_i$ be the vector associated with token i. Then we are just going to compute the matrix:

$$\boldsymbol{Y}^{(0)} = \begin{bmatrix} \mathbf{u}_{t_1} \\ \mathbf{u}_{t_2} \\ \vdots \\ \mathbf{u}_{t_m} \end{bmatrix}.$$

That is, we lookup the embedding for each token and insert it as a row of this matrix. Alternatively (and equivalently), if $\mathbf{e}_i$ is the one hot encoding of the token id i, so $\mathbf{e}_i \in \mathbb{R}^T$ with a 1 in the ith position and 0 elsewhere,

[5] This is the construction used in GPT-2, but more generally, we can imagine other useful input matrices $\boldsymbol{X}^{(0)}$.

[6] PARAMETER
N is the number of transformer layers used.

[7] PARAMETER
d is the embedding dimension.

and if $\boldsymbol{W}_E$ is the matrix

$$\boldsymbol{W}_E = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_T \end{bmatrix},$$

then we have

$$\text{the input matrix } \boldsymbol{X}^{(0)} = \begin{bmatrix} \mathbf{e}_{t_1} \\ \mathbf{e}_{t_2} \\ \vdots \\ \mathbf{e}_{t_m} \end{bmatrix} \text{ and output } \boldsymbol{Y}^{(0)} = \boldsymbol{X}^{(0)} \boldsymbol{W}_E.$$

The GPT-2 function uses one extra idea at this stage that has been discarded in more modern LLMs: the positional encoding. This adds a *per position* vector into the embedding to allow the subsequence methods to distinguish where the token was in the sequence. Hence, the actual GPT-2 function uses

$$\boldsymbol{Y}^{(0)} = \boldsymbol{X}^{(0)} \boldsymbol{W}_E + \boldsymbol{W}_P$$

where $\boldsymbol{W}_P$ is an $m \times d$ matrix of positional information for each token location.[8] This matrix is called the positional encoding. This is no longer used in modern LLMs.

Since these weights need to be *fit* to data, this means we must impose an upper limit on how big m can be. The *context length* is this limit and is commonly denoted L.[9]

38.3 LAYERNORM ON EMBEDDINGS

Throughout the GPT-2 pipeline, we transform this set of embedding vectors represented by the matrix $\boldsymbol{X}^{(i)}$. We utilize a number of functions that apply to each embedding individually. These become row-wise functions, since each embedding is a row of the matrix of all embeddings. The first such function is called LayerNorm, which is a trainable normalization step. We'll mention the parameters when we see a specific instance of this function.

The LayerNorm function applies to each row of the matrix independently, or each *vector* in the sense of a matrix as a collection of vectors (which, recall, are row vectors in this section). Normalizing data is standard in methods like the power method. Here we associating a vector with each token and moving it through the pipeline and wish to normalize it, which seems a fairly reasonable idea. The normalization does an approximate *standardization* of the vector to make it zero mean and unit variance.[10] After standardization, LayerNorm scales them with a gain and

[8] WEIGHTS
$\boldsymbol{W}_P$ is the positional encoding.

[9] PARAMETER
L is the context length.

[10] We call this standardization because, if the data were from a normal distribution, this would translate the data into a standard $N(0, 1)$ distribution.

adds a bias. Formally, let $\mathbf{v}$ be a vector of data then

$$[\text{LayerNorm}(\mathbf{v})]_i = g_i \frac{v_i - \mu}{\sigma} + b_i \text{ where}$$

$$\mu = \frac{1}{d}\sum_i v_i, \quad \sigma = \sqrt{\frac{1}{d}\sum_i (v_i - \sigma)^2 + \varepsilon}.$$

This is an element-wise computation that depends on the vector mean μ and approximate variance σ that is adjusted to avoid *divide-by-zero* errors for constant data $\mathbf{v}$.[11] The trainable parameters here are: $\mathbf{g}$, a per-dimension gain, and $\mathbf{b}$, a per dimension bias.

There is a matrix expression that we can write for LayerNorm applied to a collection of embedding vectors that is *almost* purely algebraic. Let the diag operator here create a diagonal matrix with the diagonal of its input. Let $\boldsymbol{X}$ be an $m \times d$ matrix of these embeddings. Let $\boldsymbol{Z}$ be the mean-centered matrix[12]

$$\boldsymbol{Z} = \boldsymbol{X} - \tfrac{1}{d}\boldsymbol{X}\mathbf{e}^T\mathbf{e}$$

and we have

$$\text{LayerNorm}(\boldsymbol{X}) = \left(\tfrac{1}{d}\operatorname{diag}(\boldsymbol{Z}^T\boldsymbol{Z}) + \varepsilon\boldsymbol{I}\right)^{1/2}\boldsymbol{Z}\operatorname{diag}(\mathbf{g}) + \mathbf{b}^T\mathbf{e}.$$

At this point, we could substitute $\boldsymbol{Z}$ into this expression for a result purely in terms of $\boldsymbol{X}$. The following doesn't seem immediately useful, however

$$\text{LayerNorm}(\boldsymbol{X}) = \left(\tfrac{1}{d}\operatorname{diag}\big((\boldsymbol{X} - \tfrac{1}{d}\boldsymbol{X}\mathbf{e}^T\mathbf{e})^T(\boldsymbol{X} - \tfrac{1}{d}\boldsymbol{X}\mathbf{e}^T\mathbf{e})\big) + \varepsilon\boldsymbol{I}\right)^{1/2}(\boldsymbol{X} - \tfrac{1}{d}\boldsymbol{X}\mathbf{e}^T\mathbf{e})\operatorname{diag}(\mathbf{g}) + \mathbf{b}^T\mathbf{e}.$$

38.4 TRANSFORMER BLOCKS

Each of the N transformer blocks works the same way, it combines a multi-head attention mechanism and a two layer perceptron with a LayerNorm function before each.[13] Each layer has distinct weight matrices as well. To keep notation simple, we'll describe the input starting from the matrix $\boldsymbol{X}$, but do remember that these are the $\boldsymbol{X}^{(i)}$ inputs, which are the results of the previous layer.

The initial LayerNorm

We first transform the input with a LayerNorm[14]

$$\bar{\boldsymbol{X}} = \text{LayerNorm}(\boldsymbol{X}).$$

This has the effect of ensuring that subsequent steps in the transformer layer use reasonably scaled values. This introduces two weight vectors mentioned at right.

Multi-head attention

We next transform the input:[15]

[11] The value of ε in GPT-2 and GPT-3 is 10^{-5}. In other LLMs, such as LLaMA and DeepSeek, LayerNorm has been replaced with a similar function called RMSNorm with a similar value of ε that is 10^{-6} in most implementations.

[12] Seeing $\mathbf{e}^T\mathbf{e}$ as the *outer product matrix* of all ones is very different from the rest of this book with the row-vectors here.

[13] This is jargon, these are just simple functions with weight matrices and distinct structure.

[14] WEIGHTS
$\mathbf{b}_X, \mathbf{g}_X$

[15] WEIGHTS
$\boldsymbol{W}_Q^{(i)}, \boldsymbol{W}_K^{(i)}, \boldsymbol{W}_V^{(i)}$ and biases $\mathbf{b}_Q^{(i)}, \mathbf{b}_K^{(i)}, \mathbf{b}_V^{(i)}$ per layer

$$\boldsymbol{Q} = \bar{\boldsymbol{X}}\boldsymbol{W}_Q + \mathbf{e}^T\mathbf{b}_Q, \quad \boldsymbol{K} = \bar{\boldsymbol{X}}\boldsymbol{W}_K + \mathbf{e}^T\mathbf{b}_K, \quad \boldsymbol{V} = \bar{\boldsymbol{X}}\boldsymbol{W}_V + \mathbf{e}^T\mathbf{b}_V.$$

Then we interpret $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ as h blocks of dimension d_h, based on a column split.[16]

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{Q}_1 & \cdots & \boldsymbol{Q}_h \end{bmatrix} \quad \boldsymbol{K} = \begin{bmatrix} \boldsymbol{K}_1 & \cdots & \boldsymbol{K}_h \end{bmatrix} \quad \boldsymbol{V} = \begin{bmatrix} \boldsymbol{V}_1 & \cdots & \boldsymbol{V}_h \end{bmatrix}$$

This is the *multi-head attention* idea. Each matrix $\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i$ has dimensions $m \times d_h$. Usually, $d_h = d/h$, where the dimensions are chosen so that everything comes out to be an integer.

Next, compute

$$\boldsymbol{A}_i = \text{SoftMax}\left(\frac{1}{\sqrt{d_h}}\boldsymbol{Q}_i\boldsymbol{K}_i^T\right)\boldsymbol{V}_i$$

as the ith attention block. This gives a pairwise interaction among all the tokens. Each $\boldsymbol{A}_i$ is $m \times d_h$. We concatenate them all together[17]

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{A}_2 & \cdots & \boldsymbol{A}_h \end{bmatrix} \text{ then compute } \boldsymbol{O} = \boldsymbol{A}\boldsymbol{W}_O + \mathbf{e}^T\mathbf{b}_O.$$

The output $\boldsymbol{O}$ is a $m \times d$ matrix again. Following this, we combine the output with the input in what is often called a residual addition[18]

$$\boldsymbol{X}' = \boldsymbol{X} + \boldsymbol{O}.$$

The Two Layer Perceptron

A perceptron looks a lot like a graph convolutional network, but without the graph convolution. In a perceptron, we scale the input up to a larger size with a matrix (and bias), apply a nonlinearlity, then scale back down to an output dimension with another trainable matrix (and bias). The inner perceptron dimension is d_f.[19] The nonlinearity for GPT-2 is the GELU function we'll describe shortly. The process begins with another LayerNorm, applies the

$$\begin{aligned} \bar{\boldsymbol{X}}' &= \text{LayerNorm}(\boldsymbol{X}') \\ \boldsymbol{Z} &= \bar{\boldsymbol{X}}'\boldsymbol{W}_1 + \mathbf{e}^T\mathbf{b}_1 \\ \boldsymbol{Z}' &= \text{GELU}(\boldsymbol{Z}) \\ \boldsymbol{M} &= \boldsymbol{Z}'\boldsymbol{W}_2 + \mathbf{e}^T\mathbf{b}_2 \\ \boldsymbol{Y} &= \boldsymbol{M} + \boldsymbol{X}'. \end{aligned}$$

This step introduces the LayerNorm vectors, two more weight matrices, and two vectors.[20]

The GELU function from Hendrycks and Gimpel [2016] is a variation on the ReLU function, see figure 38.1. It is smooth, with no discontinuity in the gradient at 0. The exact GELU function is[21]

$$\text{GELU}(x) = \tfrac{1}{2}x(1 + \text{error function}(x/\sqrt{2}))$$

but since speed is crucial for ML training and the error function is expensive to compute, the following approximation is universally used including

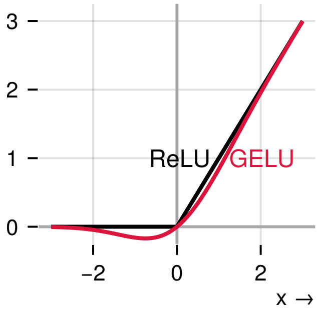


FIGURE 38.1 – ReLU and GELU are almost the same.

[16] A better way to think about this is that we have h matrices $\boldsymbol{W}_{Q,1}, \ldots, \boldsymbol{W}_{Q,h}$ for each set of query, key, and value matrices. But this is getting to be an index soup.

[17] WEIGHT $\boldsymbol{W}_O^{(i)}$ and bias $\mathbf{b}_O^{(i)}$ per layer

[18] This is more jargon.

[19] PARAMETER d_f is the inner perceptron dimension. This is commonly chosen as $4d$, which is the case in GPT-2.

[20] WEIGHTS $\mathbf{b}_P^{(i)}, \mathbf{g}_P^{(i)}, \boldsymbol{W}_1^{(i)}, \boldsymbol{W}_2^{(i)}, \mathbf{b}_1^{(i)}, \mathbf{b}_2^{(i)}$

[21] The error function is commonly written erf and is error function$(x) = 2/\sqrt{\pi}\int_0^x \exp(-t^2)\,dt$. Computational evaluations are expensive.

in GPT-2

$$\mathrm{GELU}(x) \approx \tfrac{1}{2}x\big(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3))\big).$$

The plot in figure 38.1 does not distinguish the difference between the two functions and so this is a reasonable approximation to make.

38.5 THE CAUSAL MASK IN THE TRANSFORMER

There is one additional detail in the transformer architecture. As the transformer executes, we want to make sure that past tokens are not impacted by future tokens. For this reason, we add a *causal mask* to the evaluation of the multi-head attention step.

The computation of $\boldsymbol{A}_i$ for each attention block includes this additional masking step

$$\boldsymbol{A}_i = \mathrm{SoftMax}(\frac{1}{\sqrt{d_h}}\boldsymbol{Q}_i\boldsymbol{K}_i^T + \boldsymbol{M})\boldsymbol{V}_i.$$

The mask $\boldsymbol{M}$ is just an $m \times m$ upper-triangular matrix of $-\infty$ values above the diagonal

$$\boldsymbol{M} = \begin{bmatrix} 0 & -\infty & -\infty & \cdots & -\infty \\ 0 & 0 & -\infty & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & -\infty \\ 0 & \cdots & \cdots & 0 & 0 \end{bmatrix}.$$

Adding these $-\infty$ values will cause those entries will be zeroed out in the SoftMax computation since $e^{-\infty} = 0$.

38.6 ROUNDING

The final step is rounding the output from the transformer back to tokens. This begins with another LayerNorm operation[22] to translate $\boldsymbol{X}^{(N+1)}$ to $\bar{\boldsymbol{X}}^{(N+1)}$.Then we need to return from our embedding space back to tokens. We use the transpose matrix $\boldsymbol{W}_E^T$ for this purpose

$$\boldsymbol{Y}_{\text{logits}} = \boldsymbol{W}_E^T\hat{\boldsymbol{X}}.$$

After the LayerNorm and $\boldsymbol{W}_E^T$ operation, the outputs are often called the *logit* outputs or *logits*. This term requires a brief aside. The term logit was invented as a short way to say *log odds* and came after the term *probit*, which stood for *probability unit*.[23] A helpful way to remember is logit is *logistic input* because, in the binary classification case, the logistic map is used to turn a real value z into a probability via the logistic function $p = \frac{1}{1+e^{-z}}$. Equivalently, $z = \log(p/(1-p))$ is called the *log odds* ratio for p. Logit as *logistic input* is ahistorical as the term was designed to sound like *probit*. Because SoftMax generalizes the *logistic transform* to multiclass classification, the term logits to describe this output stuck.

[22] WEIGHTS $\mathbf{g}^{(N+1)}, \mathbf{b}^{(N+1)}$

[23] For more on the history of probit and logit terms, see Cramer [2004].

To get back to actual tokens after we have $\boldsymbol{Y}_{\text{logits}}$, we apply a *temperature-dependent* SoftMax computation over rows to turn the values into $\boldsymbol{Y}_{\text{logits}}$ into token probabilities. This is

$$\boldsymbol{Y} = \text{SoftMax}_{\beta}(\boldsymbol{Y}_{\text{logits}})$$

Recall that SoftMax actually has a parameter β called the inverse temperature

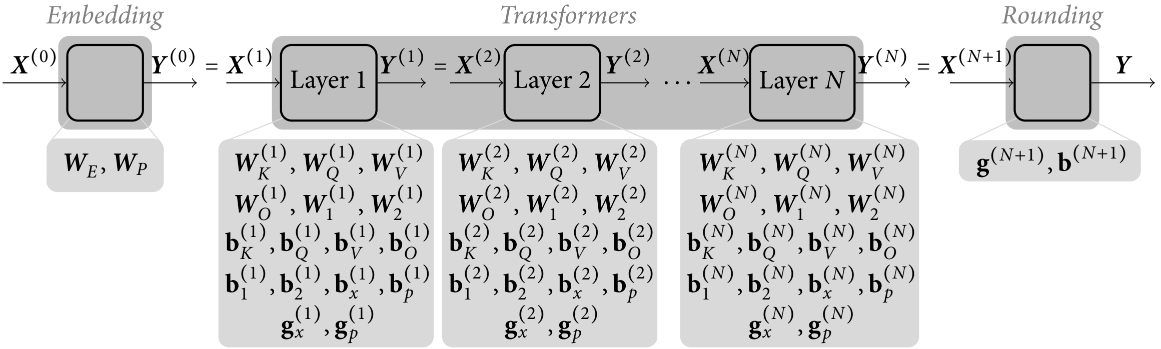
$$\text{SoftMax}_{\beta}(\mathbf{v}) = \left[\frac{\exp(\beta v_1)}{\sum_j \exp(\beta v_j)} \quad \frac{\exp(\beta v_2)}{\sum_j \exp(\beta v_j)} \quad \cdots \quad \frac{\exp(\beta v_d)}{\sum_j \exp(\beta v_j)} \right].$$

While in the transformer layers, we just use a simple SoftMax ($\beta = 1$), for the rounding step, exposing this parameter is helpful as it allows us to control how *random* the result is. Setting β close to 0 will result in high randomness whereas setting β close to ∞ will result in picking the largest probability often. This parameter is usually exposed as the temperature $1/\beta$ where temperature close to 0 suggests *greedily choosing large values* and high temperature suggests *high randomness*. This choice seems to better match our intuition for high randomness at high temperature.

When we are interested in text completion, then we simply look at the mth row of $\boldsymbol{Y}$ to get the probabilities of the next token. To generate the next token, we simply *sample* from the resulting probability distribution. From a usage point of view, it is often useful to have the logits as the output from the transformer function as they are easy to convert into the probabilities via SoftMax and this also makes it easy to adjust the temperature parameter. Consequently, whether the outputs of the GPT-2 function are the logits or the probabilities may depend on the implementation.

38.7 RECAP OF GPT-2 AND CONTEXT LENGTH

Let's look at the overall structure again, this time with all the weight matrices annotated listed.



Embedding
Transformers
Rounding
$\boldsymbol{X}^{(0)}$
$\boldsymbol{Y}^{(0)} = \boldsymbol{X}^{(1)}$
Layer 1
$\boldsymbol{Y}^{(1)} = \boldsymbol{X}^{(2)}$
Layer 2
$\boldsymbol{Y}^{(2)}$
$\cdots$
$\boldsymbol{X}^{(N)}$
Layer N
$\boldsymbol{Y}^{(N)} = \boldsymbol{X}^{(N+1)}$
$\boldsymbol{Y}$
$\boldsymbol{W}_E, \boldsymbol{W}_P$
$\boldsymbol{W}_K^{(1)}, \boldsymbol{W}_Q^{(1)}, \boldsymbol{W}_V^{(1)}$
$\boldsymbol{W}_O^{(1)}, \boldsymbol{W}_1^{(1)}, \boldsymbol{W}_2^{(1)}$
$\mathbf{b}_K^{(1)}, \mathbf{b}_Q^{(1)}, \mathbf{b}_V^{(1)}, \mathbf{b}_O^{(1)}$
$\mathbf{b}_1^{(1)}, \mathbf{b}_2^{(1)}, \mathbf{b}_x^{(1)}, \mathbf{b}_p^{(1)}$
$\mathbf{g}_x^{(1)}, \mathbf{g}_p^{(1)}$
$\boldsymbol{W}_K^{(2)}, \boldsymbol{W}_Q^{(2)}, \boldsymbol{W}_V^{(2)}$
$\boldsymbol{W}_O^{(2)}, \boldsymbol{W}_1^{(2)}, \boldsymbol{W}_2^{(2)}$
$\mathbf{b}_K^{(2)}, \mathbf{b}_Q^{(2)}, \mathbf{b}_V^{(2)}, \mathbf{b}_O^{(2)}$
$\mathbf{b}_1^{(2)}, \mathbf{b}_2^{(2)}, \mathbf{b}_x^{(2)}, \mathbf{b}_p^{(2)}$
$\mathbf{g}_x^{(2)}, \mathbf{g}_p^{(2)}$
$\boldsymbol{W}_K^{(N)}, \boldsymbol{W}_Q^{(N)}, \boldsymbol{W}_V^{(N)}$
$\boldsymbol{W}_O^{(N)}, \boldsymbol{W}_1^{(N)}, \boldsymbol{W}_2^{(N)}$
$\mathbf{b}_K^{(N)}, \mathbf{b}_Q^{(N)}, \mathbf{b}_V^{(N)}, \mathbf{b}_O^{(N)}$
$\mathbf{b}_1^{(N)}, \mathbf{b}_2^{(N)}, \mathbf{b}_x^{(N)}, \mathbf{b}_p^{(N)}$
$\mathbf{g}_x^{(N)}, \mathbf{g}_p^{(N)}$
$\mathbf{g}^{(N+1)}, \mathbf{b}^{(N+1)}$

The weights components have dimensions and usages

$\boldsymbol{W}_E : T \times d$	Token embedding
$\boldsymbol{W}_P : L \times d$	Position embedding
$\boldsymbol{W}_K^{(i)}, \boldsymbol{W}_Q^{(i)}, \boldsymbol{W}_K^{(i)}, \boldsymbol{W}_O^{(i)} : d \times d$	Attention matrices in layer i
$\mathbf{b}_K^{(i)}, \mathbf{b}_Q^{(i)}, \mathbf{b}_K^{(i)}, \mathbf{b}_O^{(i)} : d$ vector	Attention bias vectors in layer i
$\mathbf{g}_x^{(i)}, \mathbf{b}_x^{(i)} : d$ vector	Layer norm parameters for transformer output
$\boldsymbol{W}_1^{(i)}, \boldsymbol{W}_2^{(i)} : d \times d_f, d_f \times d$	Perceptron matrices
$\mathbf{b}_1^{(i)}, \mathbf{b}_2^{(i)} : d_f, d$ vector	Perceptron bias vectors
$\mathbf{g}_y^{(i)}, \mathbf{b}_y^{(i)} : d$ vector	Layer norm parameters for perceptron output
$\mathbf{g}^{(N+1)}, \mathbf{b}^{(N+1)} : d$ vector	Layer norm parameters for rounding

So we if read off the table sequentially, with some slight simplification, we have

$$(T + L)d + (4d^2 + 6d + 2d_f d + d_f + 3d)N + 2d \text{ parameters.}$$

We enumerate a set of standard configurations of GPT-2 in table 38.1.

Parameter	d	N	d_f	Total
Small	768	12	3072	124,439,808
Medium	1024	24	4096	354,823,168
Large	1280	36	5120	774,030,080
XL	1600	48	6400	1,557,611,200

TABLE 38.1 – Standard configurations of GPT-2. They all use $d_h = 64$, $T = 50,257$ and $L = 1024$ throughout all configurations. Notice also that $d_f = 4d$.

The parameter L is the maximum sequence length possible. This is also called the *context length* of a model.

38.8 A GPT-2 IMPLEMENTATION

In the supplementary codes, we have a fully functional implementation of GPT-2 that includes an approximate or simplified tokenization strategy. This enables us to test out the function.

Simple pieces of code

We begin with a few simple pieces of code that describe the LayerNorms, SoftMax, and GELU functions.

```
"""
    gelu_tanh(x)
Approximate GELU activation function using the tanh approximation from
Hendrycks and Gimpel (2016), "Gaussian Error Linear Units (GELUs)",
https://arxiv.org/abs/1606.08415. This is what’s used in all GPT-2 models.
"""
gelu_tanh(x) = 0.5x * (1 + tanh(sqrt(2/pi) * (x + 0.044715x^3)))
"""
```

```julia
    softmax!(x)
Normalize the entries of vector ‘x‘ in-place using a numerically stable
softmax. The routine subtracts the maximum entry from ‘x‘ to avoid
overflow before exponentiating and finally scales the result so the
entries sum to one. Returns the mutated input vector so it can be used in
broadcasted calls such as ‘softmax!.(eachrow(scores))‘.
"""
function softmax!(x) # in-place stable softmax
  x .-= maximum(x)
  @. x = exp(x)
  x ./= sum(x)
end
"""
    LayerNorm(gain, bias)
A layer normalization layer with learnable ‘gain‘ and ‘bias‘ parameters.
Use ‘layernorm(X, ln)‘ to apply the layer norm to each row of ‘X‘.
Use ‘layernorm!(X, ln)‘ to do the operation in-place on ‘X‘.
"""
struct LayerNorm{T}
  gain::Array{T,1}
  bias::Array{T,1}
end
"""
    layernorm!(X, ln)
Apply layer normalization to each row of ‘X‘ in-place using the
parameters in ‘ln::LayerNorm‘. Returns the mutated ‘X‘.
"""
function layernorm!(X, ln::LayerNorm)
  for i in axes(X,1)
    x = @view X[i,:]
    m = mean(x)
    v = var(x; corrected=false)
    @. x = ln.gain * ((x - m)/ sqrt(v + 1e-5)) + ln.bias
  end
  return X
end
"""
    layernorm(X, ln)
Apply layer normalization to each row of ‘X‘ using the parameters in
‘ln::LayerNorm‘. Returns a new array and does not modify ‘X‘.
"""
layernorm(X, ln::LayerNorm) = layernorm!(copy(X), ln)
```

Transformers

The main implementation is broken down into a transformer structure and `apply!` function. The `apply!` function implements the bulk of the work. This implementation outputs the logits before they are converted to probabilities via the SoftMax.

```
"""
    GPTTransformerLayer(QKV, bqkv, O, bO, W1, b1, W2, b2, lns)
A single transformer layer for GPT-2 with parameters:
- QKV: weights for query, key, value (d x 3*d)
- bqkv: biases for query, key, value (3*d)
- O: output projection weights after attention (d x d)
- bO: output projection bias after attention (d)
- W1: first weight matrix in feedforward (d x d_f)
- b1: first bias in feedforward (d_f)
- W2: second weight matrix in feedforward (d_f x d)
- b2: second bias in feedforward (d)
- lns: tuple of two LayerNorm layers

Use ‘apply!(X, layer)‘ to apply the layer to input ‘X‘ (sequence length x d_model).
"""
struct GPTTransformerLayer{T}
  QKV::Matrix{T}      # weights for Q, K, V
  bqkv::Vector{T}     # biases for Q, K, V
  O::Matrix{T}        # projection weights after attention
  bO::Vector{T}       # projection bias after attention
  W1::Matrix{T}       # first weight matrix in feedforward
  b1::Vector{T}       # first bias in feedforward
  W2::Matrix{T}       # second weight matrix in feedforward
  b2::Vector{T}       # second bias in feedforward
  lns::Tuple{LayerNorm{T},LayerNorm{T}}  # layer norms
end
"""
    apply!(X, layer)
Apply the transformer layer ‘layer‘ to input ‘X‘ (sequence length x d_model).
Returns the output array.
"""
function apply!(X, layer::GPTTransformerLayer)
  m, d = size(X)  # m = sequence length, d = em
  Xbar = layernorm(X, layer.lns[1])
  QKV = Xbar * layer.QKV .+ layer.bqkv'  # m x 3d
  Q = QKV[:, 1:d]; K = QKV[:, d+1:2d]; V = QKV[:, 2d+1:3d]
  A = similar(X); dh = 64 # multihead attention...
  for head_offset = 1:dh:d
```

```
    head_offsets = head_offset:head_offset+dh-1
    Qh = Q[:, head_offsets]; Kh = K[:, head_offsets]; Vh = V[:, head_offsets]
    scores = (Qh * Kh') ./ sqrt(dh)                          # Attention scores
    scores .= ifelse.(triu(ones(m,m),1) .== 1, -Inf, scores) # Causal mask
    softmax!.(eachrow(scores))
    Ah = scores * Vh
    A[:, head_offsets] .= Ah
  end
  # Project back to d-dim
  attn_proj = A * layer.O .+ layer.bO'
  X1 = X  .+ attn_proj  # residual connection
  X1bar = layernorm!(X1, layer.lns[2])
  # Feedforward network
  ff1 = gelu_tanh.(X1bar * layer.W1 .+ layer.b1')
  ff2 = ff1 * layer.W2 .+ layer.b2'
  X .+= ff2 .+ attn_proj  # in-place update  X = X1 + ff2
  return X
end
"""
    GPT2(E, P, layers, ln_final)
A GPT-2 model with parameters:
- E: token embedding matrix (vocab size x d_model)
- P: positional encoding matrix (max position x d_model)
- layers: vector of GPTTransformerLayer layers
- ln_final: final LayerNorm layer
Use `gpt2(token_sequence, model)` to apply the model to a sequence of token IDs (0 index based).
"""
struct GPT2{T}
  E::Array{T,2}       # token embedding
  P::Array{T,2}       # positional encoding
  layers::Vector{GPTTransformerLayer{T}}  # transformer layers
  ln_final::LayerNorm{T} # final layer norm
end
function gpt2(token_sequence::Vector, model::GPT2)
  T = size(token_sequence, 1)  # sequence length
  @assert T <= size(model.P, 1) "token sequence too long for positional encoding"
  X = model.E[token_sequence .+ 1, :] .+ model.P[1:T, :]  # initial embedding + pos enc
  for layer in model.layers
    X = apply!(X, layer)
  end
  X = layernorm!(X, model.ln_final) # Final layer norm
  return X * model.E'  # logit weights
end
```

Harness

The overall prediction and text completion harness does the *token-by-token* sequence prediction and probability sampling.

```
function sequence_predict(model::GPT2, start_sequence; maxlen=20, temperature=1.0)
  token_sequence = copy(start_sequence)
  for i in 1:maxlen
    logits = gpt2(token_sequence, model)
    last_logits = logits[end, :] ./ temperature
    softmax!(last_logits)
    next_token = sample(1:length(last_logits), Weights(last_logits))
    push!(token_sequence, next_token - 1)  # convert to 0-based
  end
  return token_sequence
end
function gpt2(input::String; kwargs...)
  token_sequence = encode(tokenizer, input)
  predicted_sequence = sequence_predict(gpt2model, token_sequence; kwargs...)
  return decode(tokenizer, predicted_sequence)
end
```

Examples

We can begin by checking what our original input sequence predicts in terms of the logit values

```
Y = gpt2([19044, 45977, 389, 616, 2460, 290], gpt2model)
```

```
6×50257 Matrix{Float32}:
  -31.1059   -30.3404   -33.0877  ...   -38.8288   -31.0339
  -88.2155   -86.3531   -88.1646       -96.2014   -88.5457
 -103.011   -104.173   -107.832       -109.346   -104.391
  -91.8463   -91.6158   -93.8959       -97.6051   -93.2191
  -78.2604   -81.6379   -85.5505       -94.0848   -81.6429
 -103.487   -104.962   -106.44    ...  -107.99    -103.169
```

and also in terms of tokens.

```
sequence_predict(gpt2model,
  [19044, 45977, 389, 616, 2460, 290];
  maxlen=11, temperature=0.0001)'
```

```
1×17 adjoint(::Vector{Int64}) with eltype Int64:
 19044  45977  389  616  2460  290  314  423  587  1762  319  606
      329  257  981  783  13
```

The raw token numbers are not very informative. So the `gpt2` function makes it easier.

```
gpt2("Matrices are my friends and"; maxlen=11, temperature=1e-4)
```

```
"Matrices are my friends and I have been working on them for a
     while now."
```

A pleasant sentiment from a set of matrices if I may offer a comment.

The small GPT-2 model used here is fairly primitive. Yet it can still complete some patterns.

```
gpt2("
1, 2, 3, 4, 5,"; maxlen=50, temperature=0.2)
```

```
"\n1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
     19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,"
```

```
gpt2("
A, B, C, D, E,"; maxlen=50, temperature=0.2)
```

```
"\nA, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U,
     V, W, X, Y, Z.\n\nA, B, C,"
```

The next two examples can succeed or fail depending on the randomness or temperature. They are not predicted correctly by a greedy assignment. Consequently, we fix the randomness.

```
Random.seed!(4)
gpt2("12, 21, 13, 31, 14, 41, 15, 51, 16,";
  maxlen=20, temperature=0.2)
```

```
"12, 21, 13, 31, 14, 41, 15, 51, 16, 61, 67, 71, 73, 76, 78, 79,
     80, 81, 82,"
```

```
Random.seed!(7)
gpt2("Let \$A\$ be a matrix. A matrix \$A\$ is symmetric if \$A =";
  maxlen=10, temperature=0.5)
```

```
"Let \$A\$ be a matrix. A matrix \$A\$ is symmetric if \$A = 1\$.
     In this case, \$A\$ is"
```

The second example is interesting. The output is not very meaningful, but the model nonetheless closes the TeX math environment.

That said, many of the examples I tried were unsatisfactory. For the GPT-2 small model, do not be surprised by bad output. In the original GPT-2 paper, Radford et al. [2019], they used examples from the larger models that were superior to these including translation.

38.9 CHANGES IN MORE MODERN ARCHITECTURES

There are two primary changes in modern architecture. The first is that they no longer use positional embeddings $\boldsymbol{W}_P$ at the initial step. Instead, they apply a position embedding within the inner-products of the attention mechanism. The second is that adding explicit bias terms has fallen out of favor.

Rotary position embeddings

The GPT-2 function begins with

$$\boldsymbol{Y}^{(0)} = \boldsymbol{W}_E \boldsymbol{X}^{(0)} + \boldsymbol{W}_P.$$

We noted a few challenges and limitations with these above and the field has moved onto new ideas instead.

Using a fixed position embedding matrix $\boldsymbol{P}$ was suggested by the original paper on attention mechanism [Vaswani et al., 2017]. They used[24]

$$\boldsymbol{P}_{ij} = \begin{cases} \sin((i-1)/10000^{(j-1)/d}) & j \text{ is odd} \\ \cos((i-1)/10000^{(j-2)/d}) & j \text{ is even.} \end{cases}$$

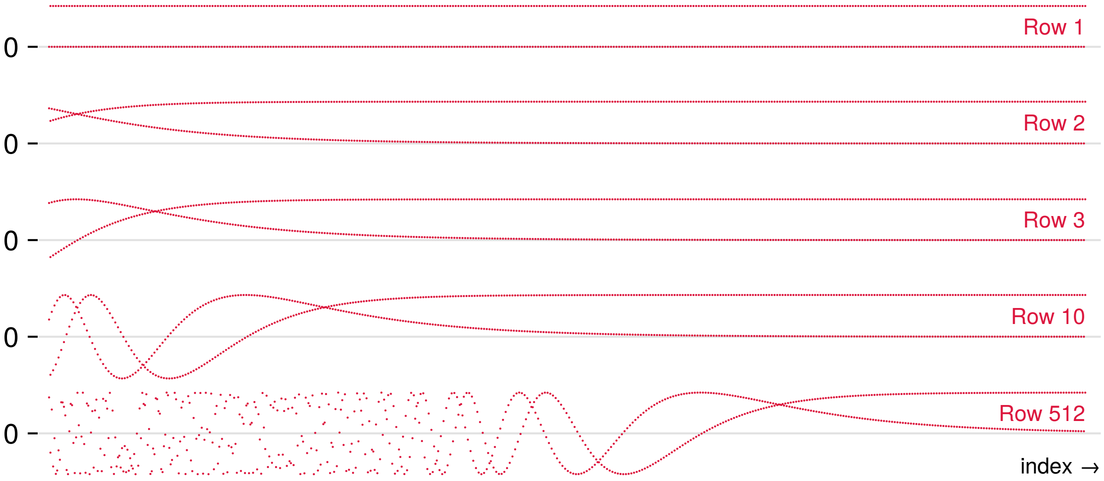


FIGURE 38.2 – The entries the fixed positional embedding matrix from the original paper on attention mechanisms shows an intermixed sine and cosine pattern. This figure shows 5 rows of the matrix. The initial rows vary slowly.

The values along a row form sine and cosine pairs, see figure 38.2. This is an odd matrix with a key interesting property. If we take row vectors $\mathbf{p}_i$ and $\mathbf{p}_j$ then we have

$$\langle \mathbf{p}_i, \mathbf{p}_j \rangle = \sum_{k=1}^{d/2} \sin\left(\tfrac{i-1}{f(k)}\right) \sin\left(\tfrac{j-1}{f(k)}\right) + \cos\left(\tfrac{i-1}{f(k)}\right) \cos\left(\tfrac{j-1}{f(k)}\right),$$

with $f(k) = 10000^{2(k-1)/d}$. Using a small identity,[25] this simplifies to

$$\langle \mathbf{p}_i, \mathbf{p}_j \rangle = \sum_{k=1}^{d/2} \cos\left(\tfrac{i-j}{f(k)}\right).$$

[24] This paper was originally described for a zero-indexed matrix and looks a bit strange on the one-indexed matrix.

[25] Recall – or lookup – $\cos(\alpha - \beta) = \cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\beta)$.

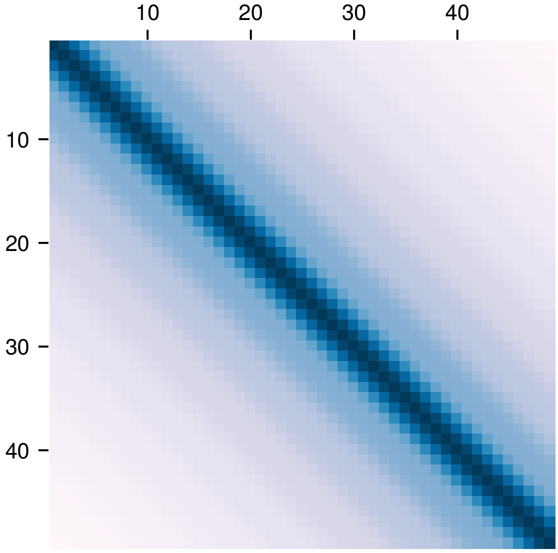


FIGURE 38.3 – A 49 × 49 matrix of inner-products from the positional embedding used in the original transformers paper Vaswani et al. [2017] shows a clear banded pattern such that each entry is proportional to $|i - j|$, the difference in position. Computed using 49 rows with $d = 768$.

Hence the difference here is proportional to $i - j$ rather than the actual positions. The full matrix of inner-products is beautiful (figure 38.3

A limitation of using this choice of $\boldsymbol{P}$ is that position information is only used at the very first step and will likely be lost over the many layers of computation. The idea of rotary position embeddings method, or RoPE, will allow us to apply this within the attention mechanism of each transformer layer.

Let $\boldsymbol{S} = \boldsymbol{Q}\boldsymbol{K}^T$ be the matrix of attention logit values. Each $S_{ij} = \langle \mathbf{q}_i, \mathbf{k}_j \rangle$. We would like this computation to reflect that tokens far away should have less relevance. The RoPE technique for this task adapts the sine and cosine values, but places them into matrices.

The final function is no longer a simple matrix expression[26]

$$S_{ij} = \langle \mathbf{q}_i \boldsymbol{R}(i), \mathbf{k}_j \boldsymbol{R}(j) \rangle.$$

The matrices $\boldsymbol{R}(i)$ and $\boldsymbol{R}(j)$ are $d_h \times d_h$ *position dependent* encoding matrices. They are composed of a set of plane rotations[27]

$$\boldsymbol{R}(i) = \begin{bmatrix} \cos(\frac{i-1}{f(1)}) & -\sin(\frac{i-1}{f(1)}) & & & & & \\ \sin(\frac{i-1}{f(1)}) & \cos(\frac{i-1}{f(1)}) & & & & & \\ & & \cos(\frac{i-1}{f(2)}) & -\sin(\frac{i-1}{f(2)}) & & & \\ & & \sin(\frac{i-1}{f(2)}) & \cos(\frac{i-1}{f(2)}) & & & \\ & & & & \cos(\frac{i-1}{f(3)}) & -\sin(\frac{i-1}{f(3)}) & \\ & & & & \sin(\frac{i-1}{f(3)}) & \cos(\frac{i-1}{f(3)}) & \\ & & & & & & \ddots \end{bmatrix}$$

using the same $f(k)$ function from before, but with d_h instead of d since we are operating *within* a single attention head. Note that $\boldsymbol{R}(i)^T \boldsymbol{R}(j)$ has

Double check the position of the minus sign here.

[26] Unpacking this expression shows that there is a *different matrix*, $\boldsymbol{R}(i)$, applied to each row of the embedding. Then we compute inner-products.

[27] See lecture 16 for more on these rotations.

the same property that the result only depends on $i - j$.

To illustrate the impact, we consider a case where the vectors $\mathbf{q}_i$ and $\mathbf{k}_j$ are all constant. Without the RoPE embeddings, we would find that $S_{ij} = d$, a constant for all pairs i, j. With the RoPE embeddings we compute $S_{ij} = \langle \mathbf{e}\boldsymbol{R}(i), \mathbf{e}\boldsymbol{R}(j) \rangle$ instead for visualization. The result is equally beautiful (figure 38.5).

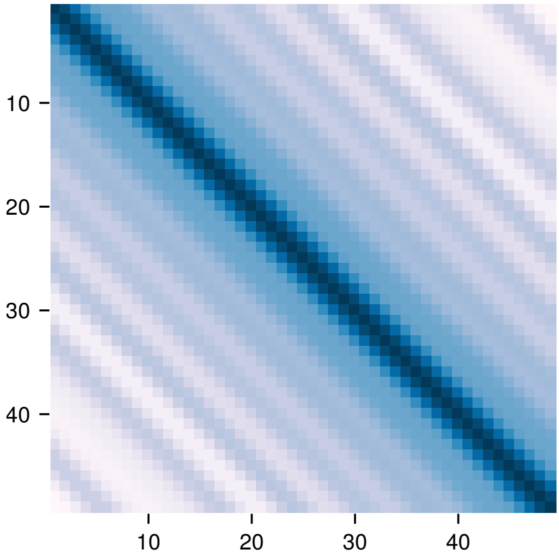


FIGURE 38.4 – A 49 × 49 matrix of inner-products that result from a rotary position embedding (RoPE) within the attention mechanism of a transformer. This shows a heatmap of the matrix $S_{ij} = \langle \mathbf{q}_i\boldsymbol{R}(i), \mathbf{k}_j\boldsymbol{R}(j) \rangle$ where $d = 64$. In comparison with the earlier fixed position embedding, there are more "waves".

In practice, we never form the matrices $\boldsymbol{R}(i)$. Rather, these are applied implicitly since they have enough structure as in the following code.

```
"""
  rope_rotate!(v, i::Int)
In-place Rotary Positional Embedding (RoPE) rotation of a
vector ‘v‘ at position ‘i‘.  Rotated ‘v‘ (in-place).
"""
function rope_rotate!(v, i::Int)
  d = length(v)
  @assert iseven(d) "d must be even"
  for j in 1:2:d
    # i,i+1 form a 2D plane (convert to 0-based index for exponent)
    ϑ = i / (10000.0 ^ ((j-1)/d))
    c = cos(ϑ); s = sin(ϑ)
    x = v[j]; y = v[j+1]
    v[j] =  c*x - s*y
    v[j+1] =  s*x + c*y
  end
  return v
end
```

The choice of the value 10000 appears to be arbitrary. In the following figure we compare different choices of this value along the main anti-diagonal (from the lower-left to the upper-right) of the matrix.

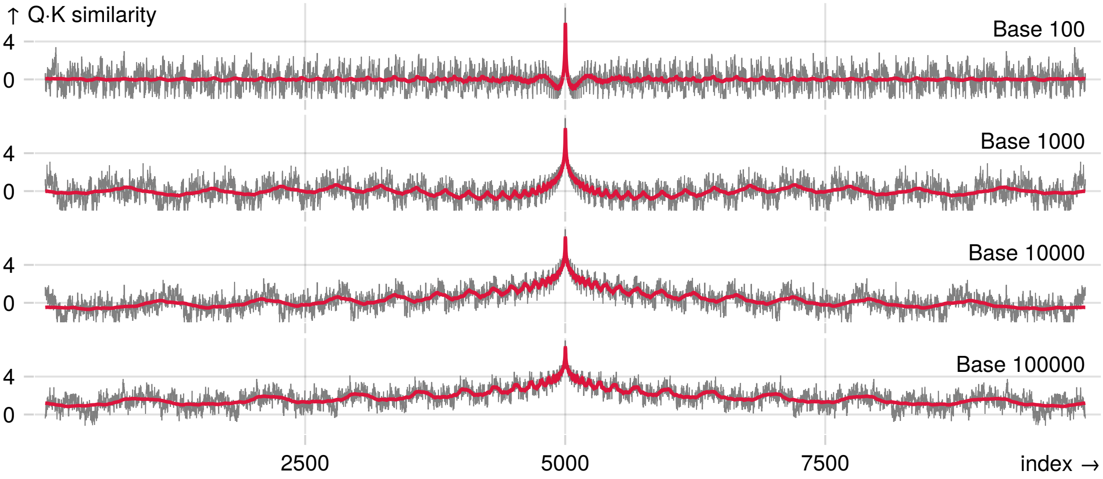


FIGURE 38.5 – We vary the base of the RoPE denominator so that $f(k) = \text{base}^{2(k-1)/d}$ where base is 100, 1000, 10000, or 100000. The jagged grey line shows the main anti-diagonal of the matrix S_{ij} with a *constant* vector $\mathbf{q}_i$ and $\mathbf{k}_j$. The red line shows the smoothed line. Increasing the value of the base increases the region where the red line is non-zero and has the effect of increasing the *effective* context size.

Removing bias terms

The GPT-2 function has 8 bias terms per layer: one for each of the key, query, value, and output matrices. Then one for each of the two layers of the perceptron. Then one for each of the two layer norms. There is a real sense in which these bias terms should be superfluously. This originates in an idea called *homogenous coordinates* in computer graphics and is a key tool in *projective geometry*. Despite the names, the idea is purely a matrix algebra idea.

The computation of the key matrix in GPT-2 is

$$\boldsymbol{K} = \bar{\boldsymbol{X}}\boldsymbol{W}_K + \mathbf{e}^T\mathbf{b}_K.$$

Suppose, instead, we had simply appended an additional dimension onto each row of $\bar{\boldsymbol{X}}$ and added $\mathbf{b}_K$ as the last row of $\boldsymbol{W}_K$.

$$\hat{\boldsymbol{X}} = \begin{bmatrix} \bar{\boldsymbol{X}} & \mathbf{e}^T \end{bmatrix} \qquad \hat{\boldsymbol{W}}_K = \begin{bmatrix} \boldsymbol{W}_K \\ \mathbf{b}_K \end{bmatrix}.$$

Then the result of

$$\hat{\boldsymbol{X}}\hat{\boldsymbol{W}}_K = \bar{\boldsymbol{X}}\boldsymbol{W}_K + \mathbf{e}^T\mathbf{b}_K = \boldsymbol{K}.$$

Modern LLMs, however, do not use this idea explicitly. Instead, they have found little performance difference with and without bias terms added. Consequently, it seems likely that the training procedure has the ability to *learn* these bias terms without us explicitly introducing them. It probably uses some implicit variation on these ideas or encoding.

The LayerNorm operations are replaced with an operation called RMSNorm, where RMS stands for *root mean square* and its designed to scale the vector such that it has variance around one. This involves removing the mean centering step and solely focus on scaling the variance. Let **v** be a vector of data then

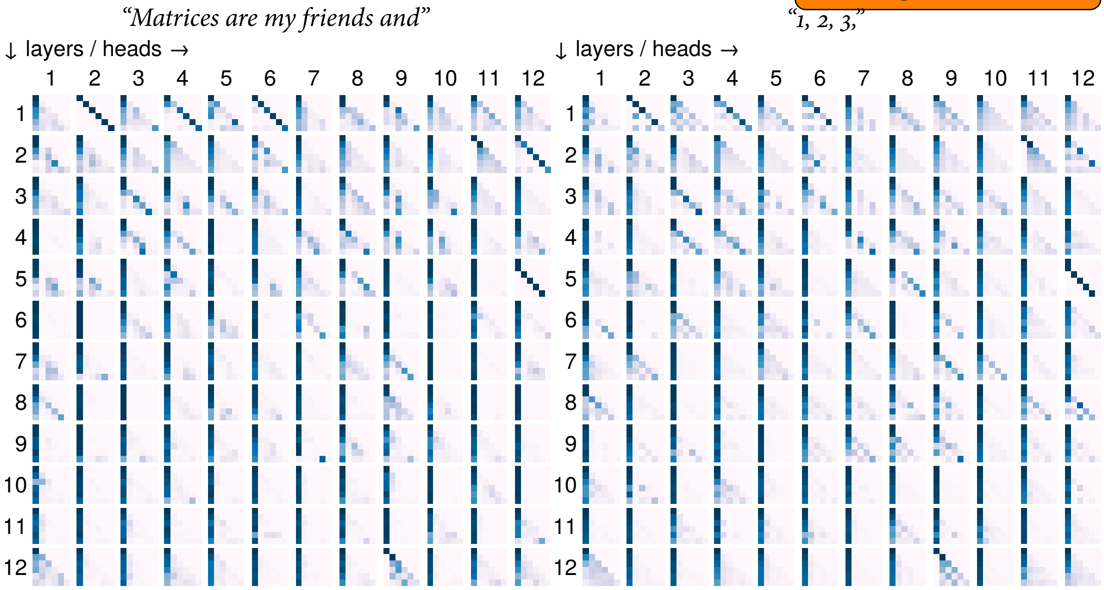
$$[\mathrm{RMSNorm}(\mathbf{v})]_i = g_i \frac{v_i}{\sigma} \quad \text{where} \qquad \sigma = \sqrt{\frac{1}{d}\sum_i v_i^2 + \varepsilon}.$$

Another small reason for these changes is that they reduce the number of floating point operations, which is important for getting fast evaluation times.

38.10 MATRICES ACROSS LAYERS

One of the interesting pieces to study are the token-token activations involved in the attention mechanisms. We compare two phrases where to understand the differences. The first phrase has an important word at the beginning. The second is a sequence broken up with commas. Although reading these structures is debatable, there does appear to be some alternating structure in the second set of activations. In both cases, activations seem to

Finish this section, make sure the plot is correct



"Matrices are my friends and"

"1, 2, 3,"

FIGURE 38.6 – At left, the attention matrices SoftMax$(\boldsymbol{K}_i \boldsymbol{Q}_i^T + \boldsymbol{M})$ for each of the 12 heads for each of the 12 layers when evaluating the GPT-2 function on the 6 tokens for "Matrices are my friends and". At right, the same data for the 6 tokens of "1, 2, 3".

EXERCISES

1. (Easy) Show that the splitting suggested by Footnote 16 gives equivalent matrices $\boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_h$ for suitable choices of the dimensions.
2. Let $L = 5, d = 512, h = 8, T = 65536, d_f = 2048$. Assume the element-wise function in the MLP takes 2 floating point mulitplications. How many floating point multiplications does evaluating GPT-2 consume?
3. It looks tempting to try and exploit the matrix structure in $\boldsymbol{K}\boldsymbol{Q}^T$ to simplify the transformer by eliminating the pair of matrices. Explain why this will not simplify the transformer for the standard GPT-2 architecture. Explain why this would be a good idea if $d_h \geq d/2$.
4. One odd aspect about the multihead attention layer is the *concatenation* step. Show that we can write $\boldsymbol{A}\boldsymbol{W}_O = \sum_{j=1}^{h} \boldsymbol{A}_j \boldsymbol{W}_{O,j}$ for some partition of $\boldsymbol{W}_O$.
5. Show that the convergence of the power method for eigenvalues and eigenvectors is not impacted by moving the normalization step from the end of the iteration to the start of the iteration. Why does this differ for the LLMs when they move their LayerNorm functions?
6. Suppose someone wanted to investigate transformers where elements are from $\mathbb{C}$ instead of $\mathbb{R}$. This is an interesting idea, but is unlikely to succeed. Show that a GPT-2 like architecture over $\mathbb{C}$ is equivalent to a GPT-2-like architecture over $\mathbb{R}$ using twice the dimensions of each problem. (Note, this probably isn't strictly true, but there's a good reason why we wouldn't expect it to be much different, that's what I want you to present.)
7. Suppose we use the token sequence for

   “`Matrices are my friends and `”,

   which is $[19044, 45977, 389, 616, 2460, 290, 220]$. This string differs by adding the space at the end. Does that change the next token generated assuming we use the most likely token?
8. When evaluating the GPT-2 function, how much storage space is required beyond the weights themselves? Specifically, if the input is $m \times d$, this takes md values. We already have the function of the number of weights. Try and optimize the evaluation to take a few additional values as possible.
9. We are going to compare *random tokens* to *language tokens*. In the file , we have a collection of token sequences sampled at random from XXX, one of the big token databases on the Hugging Face data repository. Suppose we have token sequences $T_1, \ldots, T_k$ in this file. For each token sequence, we associate a set of matrices $\boldsymbol{X}_1^{(1)}, \ldots \boldsymbol{X}_1^{(N)}$ for the input to each layer of the GPT-2 architecture.

Make this file

Let's assemble matrices

$$\boldsymbol{A}^{(1)} = \begin{bmatrix} \boldsymbol{X}_1^{(1)} \\ \boldsymbol{X}_2^{(1)} \\ \cdots \end{bmatrix} \quad \boldsymbol{A}^{(2)} = \begin{bmatrix} \boldsymbol{X}_1^{(2)} \\ \boldsymbol{X}_2^{(2)} \\ \cdots \end{bmatrix} \quad \cdots \quad \boldsymbol{A}^{(N)} = \begin{bmatrix} \boldsymbol{X}_1^{(N)} \\ \boldsymbol{X}_2^{(N)} \\ \cdots \end{bmatrix},$$

that is, we vertically concatenate the layer 1 matrices from all the different token sequences. Note that each token sequence is a set of token embedding vectors, so the matrices $\boldsymbol{A}^{(i)}$ have large collections of vector embeddings. Look at the SVD of these matrices to study how the singular values change with layer in the transformer. Compare that with a collection of randomly generated tokens with the same length as the sequences T_1. What are the challenges in computing the SVD for it?

10. We are going to do the same thing as the previous exercise except we are going to look at the SVD for *all matrices* together that is

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}^{(1)} & \boldsymbol{A}^{(2)} & \cdots & \boldsymbol{A}^{(N)} \end{bmatrix}.$$

What are the singular values of this dataset? What are the challenges in computing the SVD for it?

11. Show that we can write the *RoPE* evaluation in terms of a tensor $\underline{\boldsymbol{B}}$ such that

$$\operatorname{vec}(\boldsymbol{S}) = \underline{\boldsymbol{B}} \otimes_1 \operatorname{vec}(\boldsymbol{Q}) \otimes_2 \operatorname{vec}(\boldsymbol{K}).$$

The material from lecture 37 may be helpful.

12. Show that the original fixed position embedding matrix from Vaswani et al. [2017] can be written as

$$\boldsymbol{P} = \begin{bmatrix} \sin(\boldsymbol{A}) & \cos(\boldsymbol{A}) \end{bmatrix} \cdot \text{a permutation matrix}$$

for some matrix $\boldsymbol{A}$ with an element wise sin and cos function.

COLOPHON VIII

38 BIBLIOGRAPHY

N. Berger, C. Borgs, J. T. Chayes, and A. Saberi. *On the spread of viruses on the internet*. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 301–310. 2005. Cited on page 37.

J. Cramer. *The early origins of the logit model*. Studies in History and Philosophy of Science Part C: Studies in History and Philosophy of Biological and Biomedical Sciences, 35 (4), pp. 613–626, 2004. `doi:10.1016/j.shpsc.2004.09.003`. Cited on page 226.

R. Ghosh, S.-h. Teng, K. Lerman, and X. Yan. *The interplay between dynamics and networks: Centrality, communities, and cheeger inequality*. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1406–1415. 2014. `doi:10.1145/2623330.2623738`. Cited on page 37.

D. Hendrycks and K. Gimpel. *Gaussian error linear units (gelus)*. arXiv, cs.LG, p. 1606.08415, 2016. Cited on page 225.

J. E. Hopcroft and L. R. Kerr. *Some techniques for proving certain simple programs optimal*. In *10th Annual Symposium on Switching and Automata Theory (swat 1969)*, pp. 36–45. 1969. `doi:10.1109/swat.1969.21`. Cited on page 216.

L. Katz. *A new status index derived from sociometric analysis*. Psychometrika, 18 (1), pp. 39–43, 1953. `doi:10.1007/BF02289026`. Cited on page 37.

J. M. Landsberg. *The border rank of the multiplication of two by two matrices is seven*. 2006. Cited on page 216.

D. B. Larremore, B. K. Fosdick, K. M. Bubar, S. Zhang, S. M. Kissler, C. J. E. Metcalf, C. Buckee, and Y. Grad. *Estimating sars-cov-2 seroprevalence and epidemiological parameters with uncertainty from serological surveys*. medRxiv, 2020. `arXiv:https://www.medrxiv.org/content/early/2020/06/22/2020.04.15.20067066.full.pdf`, `doi:10.1101/2020.04.15.20067066`. Cited on page 37.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. *Language models are unsupervised multitask learners*. Technical report, OpenAI, 2019. OpenAI Technical Report. Cited on page 233.

V. Strassen. *Gaussian elimination is not optimal.* Numerische Mathematik, 13 (4), pp. 354–356, 1969. `doi:10.1007/bf02165411`. Cited on pages 215 and 216.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention is all you need.* arXiv, cs.CL, p. 1706.03762, 2017. Cited on pages 234, 235, and 240.

APPENDIX *VIII*

A

BETTER NAMES IN MATRIX COMPUTATIONS

NEUMANN SERIES	geometric series or matrix geometric series
HANKEL MATRIX	left shift matrix
TOEPLITZ MATRIX	right shift matrix
HESSENBERG MATRIX	bulge triangular
JACOBI METHOD	simultaneous variable updates / solves / improvement / relaxation parallel variable updates / solves / improvement / relaxation
GAUSS-SEIDEL METHOD	sequential variable updates / solves / improvement / relaxation
GAUSS-SEIDEL METHOD	sequential variable updates / solves / improvement / relaxation
CESÁRO SUM OR MEAN	limit of averages