Matrix
Methods
and
Programs

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

$$
\begin{array}{ccc}
\text{computers} & & \text{mathematics} \\
\& & \leftrightarrow & \& \\
\text{programs} & & \text{matrices} \\
\& & \leftrightarrow & \& \\
\text{codes} & & \text{algebra} \\
\& & \leftrightarrow & \& \\
\text{systems} & & \text{analysis}
\end{array}
$$

or put differently, we use

$$
\begin{array}{cccc}
\text{matrices} & \text{mathematics} & \text{algebra} & \text{analysis} \\
& \text{to} & & \\
\text{describe} & \text{study} & \text{improve} & \text{understand} \\
& \text{the behavior of} & & \\
\text{computers} & \text{programs} & \text{codes} & \text{systems} \\
& \text{on interesting problems} & &
\end{array}
$$

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 some person 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. These are all discussed throughout and we have appendix 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." But Stigler also wasn't the first one to mention this.

CONTENTS

MATRIX PROBLEMS & STRUCTURE

I

WHAT IS A MATRIX? $\textit{\textbf{1}}$

Let's start with the elephant in the room.

What is a matrix?

What are the things that we are going to learn to compute with in this class. Presumably you are taking it because you have some type of interest in using matrices.

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 Purdue's college of engineering or science "What is a matrix" and you'll probably get some of the answers that we'll discuss below.

- A matrix is a 2-dimensional array, or table, of numbers.
- A matrix is a linear transformation.
- A matrix is a coordinate representation of a linear transformation between vector spaces.
- 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.
- It's a table of numbers where linear algebraic manipulations make sense.
- I know it when I see it.
- a matrix stores vectors
- something used to simplify mathematical calculations

1.1 A STARTING EXAMPLE

Is the following a matrix?

$$\begin{bmatrix} 4 & 9 & 6 & 0 & 4 & 5 & 3 \\ 4 & 9 & 4 & 1 & 9 & 9 & 7 \\ 4 & 9 & 4 & 6 & 0 & 1 & 0 \\ 4 & 9 & 6 & 1 & 7 & 6 & 1 \\ 4 & 9 & 4 & 8 & 7 & 9 & 8 \\ 4 & 9 & 6 & 2 & 3 & 9 & 9 \\ 4 & 9 & 4 & 6 & 0 & 1 & 3 \\ 4 & 9 & 6 & 9 & 4 & 3 & 2 \\ 4 & 9 & 4 & 9 & 0 & 2 & 5 \\ 4 & 9 & 4 & 6 & 0 & 0 & 5 \\ 4 & 9 & 4 & 6 & 0 & 0 & 9 \end{bmatrix}$$
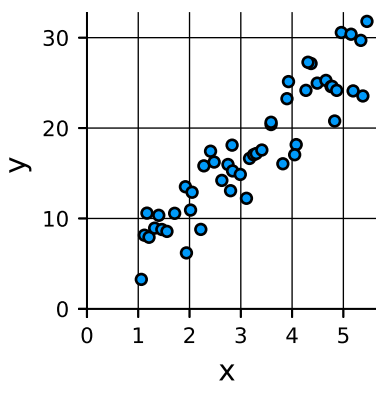
No! This is a table of telephone numbers, expanded by digit into columns, not a matrix.

1.2   AN 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 for our problem are pairs: $(y_1, x_1), ..., (y_N, x_N)$.[1] We can assemble them into a first matrix:

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

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

x	y
1.06	3.26
1.17	10.59
1.12	8.15
1.21	7.92
1.32	8.93
1.4	10.35
1.46	8.78
1.56	8.57
1.71	10.57
1.92	13.51
1.94	6.19
2.02	10.93
2.05	12.9
2.28	15.82
2.22	8.79
2.41	17.44
2.48	16.24
2.63	14.21
2.75	15.97
2.8	13.07
2.83	18.11
2.84	15.26
2.99	14.87
3.17	16.64
3.11	12.23
3.25	17.05
3.3	17.18
3.41	17.57
3.59	20.39
3.59	20.64
3.82	16.05
3.93	25.14
3.9	23.25
4.08	18.17
4.05	17.05
4.27	24.17
4.37	27.14
4.31	27.28
4.49	24.97
4.66	25.27
4.76	24.59
4.83	20.78
4.78	24.61
4.87	24.17
4.96	30.57
5.19	24.11
5.15	30.38
5.38	23.55
5.34	29.7
5.46	31.8

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.[2] 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 translate this problem with notation to eliminate the indices. Let $\mathbf{x}, \mathbf{y}$, and $\mathbf{c}$ be the vectors:

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

then

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

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 A be the matrix:

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

Then we can write our function f as:

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

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 A and the goal is to produce coefficients $\mathbf{c}$ that constitute a linear relationship.
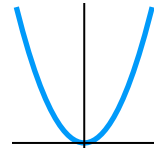
EXAMPLE 1.1 Hint that there is a matrix. Note that we can reparameterize the line as:

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

in which case we have

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

or

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

which where 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 T. ◆

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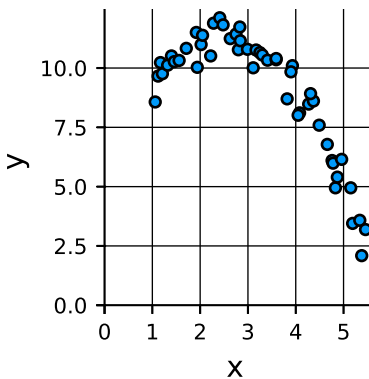
Then it's clear from the picture that a linear model isn't appropriate. We can still solve the problem we just posed, but there may be other models of our data that are appropriate. Such as a quadratic:

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

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

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

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

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

Here, the data are.

x	y
1.06	8.57
1.17	10.23
1.12	9.66
1.21	9.76
1.32	10.12
1.4	10.51
1.46	10.27
1.56	10.32
1.71	10.83
1.92	11.5
1.94	10.03
2.02	10.99
2.05	11.38
2.28	11.89
2.22	10.51
2.41	12.12
2.48	11.82
2.63	11.24
2.75	11.43
2.8	10.77
2.83	11.73
2.84	11.15
2.99	10.79
3.17	10.76
3.11	10.01
3.25	10.65
3.3	10.55
3.41	10.33
3.59	10.35
3.59	10.4
3.82	8.7
3.93	10.1
3.9	9.84
4.08	8.11
4.05	8.01
4.27	8.48
4.37	8.61
4.31	8.92
4.49	7.59
4.66	6.78
4.76	6.1
4.83	4.95
4.78	5.99
4.87	5.4
4.96	6.15
5.19	3.45
5.15	4.95
5.38	2.09
5.34	3.58
5.46	3.19

1.3 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 class. There are a great many problems in science, engineering, biology, and everywhere that can be turned into common matrix problems.

1.4 COMMON MATRIX PROBLEMS

The three canonical and most common matrix problems are:
· Linear systems of equations: find $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{b}$
· Least squares problems: find $\mathbf{x}$ to minimize $\|A\mathbf{x} - \mathbf{b}\|^2$
· Eigenvalue problems: find $\mathbf{x}, \lambda$ such that $A\mathbf{x} = \lambda\mathbf{x}$
These problems have deep relationships.

1.4.1 *Solving least squares via a linear system.*

We can turn the least squares problem

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

into a related linear system.[3]

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 :

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

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 $0 \leq \alpha \leq 1$.

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

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:

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

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

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

◆

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

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

This is zero when

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

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

1.4.2 *From an eigenvector problem to a linear system*

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

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

This isn't the standard type of linear system of equations, but it is a valid problem!

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

a matrix is a table of 2d numbers where linear algebraic operations make sense on the rows or columns

So 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.

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.

In this case, better may mean any of these:

faster   more accurately   more reliably, ...

We will see this soon!

1.5   EXERCISE

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 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.

3. 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.

4. Let A be the matrix from part 1. Consider the new matrix $B = A^T A$. Explain why this is or isn't a matrix, using the ideas from class.

5. Let A be the matrix from part 2. Consider the new matrix $B = A^T A$. Explain why this is or isn't a matrix, using the ideas from class.

6. Let A be the matrix from part 3. Consider the new matrix $B = A^T A$. Explain why this is or isn't a matrix, using the ideas from class.

2 NOTATION

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

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

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

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. Sometimes, we'll use a capital greek letter. 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 $A_{1,1}$.

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.

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

$$A : n \times n.$$

Sometimes this is nicer to write on the board.

2.2 VECTORS

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

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

Vectors are denoted by *lowercase, bold* letters. 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.*

2.3 SCALARS

Lower-case greek letters are scalars.

2.4 OPERATIONS

TRANSPOSE Let $A : m \times n$, then

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 :

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

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:

$$I_n$$

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

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The identity matrix has the property that $AI = A$ for any matrix 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

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

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 J to represent the *matrix of all ones*

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

This isn't needed however, as we can easily write this as the rank-1 matrix $J = \mathbf{e}\mathbf{e}^T$, 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

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

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.

$$A : m \times n = \begin{bmatrix} \mathbf{r}_1^T \\ \mathbf{r}_2^T \\ \vdots \\ \mathbf{r}_m^T \end{bmatrix}.$$

Here, each $\mathbf{r}_i \in \mathbb{R}^n$.

Using the column partitioning:

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

And with the row partitioning:

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

Another useful partitioned representation of a matrix is into blocks:

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

or

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

Here, the sizes "just have to work out" in the vernacular. Formally, all $A_{i,\cdot}$ must have the same number of rows and all $A_{\cdot,j}$ must have the same number of columns.

2.7 EXERCISES

1. Identify the following:

$$\mathbf{f}, z_1, \mathbf{x}_1, \alpha, \beta, C, C_1, \Sigma, B_{i,j}, \mathbf{b}_{i,j}$$

2. What are the results of $\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. Let $\mathbf{e}_i$ be the vector with all zeros and a 1 in the ith entry. Let A be an $m \times n$ matrix. Give an expression for the rth row of A as a result of a matrix vector product. Give an expression for the cth column of A as a result of a matrix vector product.

4. 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} = $?

5. $\mathbf{x} = \begin{bmatrix} 1.5 & 2 & -3 \end{bmatrix}^T$. (Assume $\mathbf{e}$ is 4×1.) $\mathbf{e}\mathbf{x}^T = $? $\mathbf{x}\mathbf{e}^T = $?

6. $\mathbf{x} = \begin{bmatrix} -5 & 4 & 2 \end{bmatrix}^T$. (Assume $\mathbf{e}_i$ is 3×1.) $\mathbf{e}_2\mathbf{x}^T = $? $\mathbf{x}\mathbf{e}_1^T = $?

STRUCTURE IN MATRICES

3

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

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

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 APPROACH FOR LEAST-
-SQUARES   [1] To find $\mathbf{x}$ such that $\|\mathbf{b} - A\mathbf{x}\|_2^2$ is minimized, we can solve the normal equations

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

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

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

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

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

[1] See the future lectures on why there are usually better ways to solve these problems.

[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.

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.

This type of input arises frequently, and so it has acquired a name. The general form is called a *Hankel* matrix.[3]

> **Definition 3.1**
> A matrix is a *Hankel 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 Hankel matrix is symmetric.
> · An $m \times n$ Hankel matrix is defined by $m + n - 1$ numbers.

Thus, as an example, we could work out how to solve linear equations with Hankel 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.

[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.

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.

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]

[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.

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:

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

$$x_{\{-1\}} = 1 + \tfrac{1}{2}x_{\{-2\}} + \tfrac{1}{2}x_{\{0\}}$$

$$\vdots$$

$$x_{\{+5\}} = 1 + \tfrac{1}{2}x_{\{+4\}} + \tfrac{1}{2}x_{\{+6\}}.$$

This gives us an overall linear system of equations:

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

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

[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.

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.

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

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

The case in example 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 Example **??** 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 Example **??**. 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.

[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 $o(n^2)$.

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.

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]

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

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

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.

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

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 $x_2 5, x_3 1, x_3 3, x_3 9$.

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.

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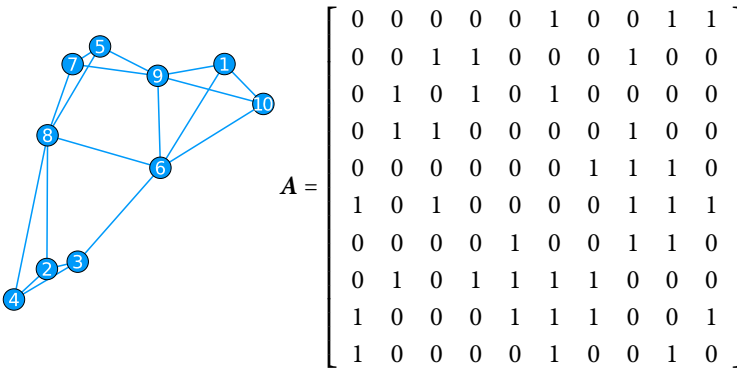1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
$$

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 $A^T A$ 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:

$$B = F^T F$$

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

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

Definition 3.7
A matrix A is symmetric positive definite if

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

3.4 FAST OPERATORS AND DATA-SPARSE MATRICES

Consider the matrix

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

This is the same matrix as in Example **??** 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 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 **C** instead by changing the two lines `return 0` to `return 0.1`.[9]

3.4.1 *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 $\quad A\mathbf{x} = \mathbf{b} \qquad$ least squares $\quad \min \|A\mathbf{x} - \mathbf{b}\| \qquad$ eigenvalues $\quad A\mathbf{x} = \lambda\mathbf{x}$

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.

3.4.2 *Data sparse.*

The matrix **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.

[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.

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.

TOEPLITZ A Toeplitz matrix is one where $A_{i,j} = A_{i+1,j+1}$ whenever the latter index is valid. For example

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

HANKEL A *Hankel matrix* is one where $A_{i,j} = A_{i-1,j+1}$ whenever $i-1$, $j+1$ is a valid index into the matrix.

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

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

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
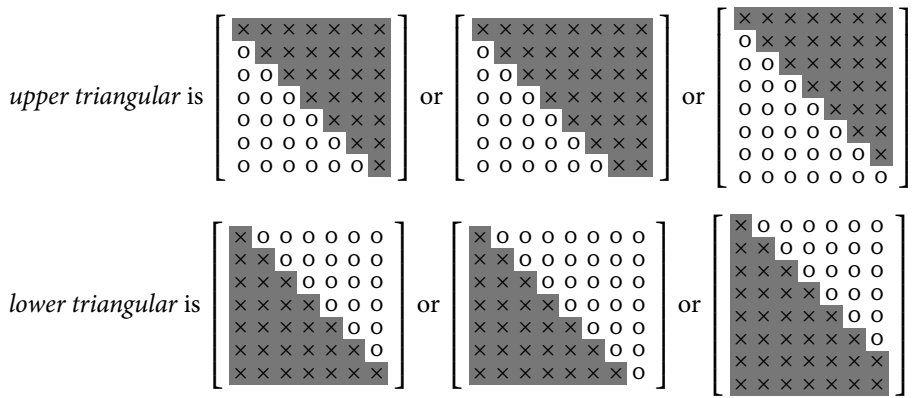
These are often square.

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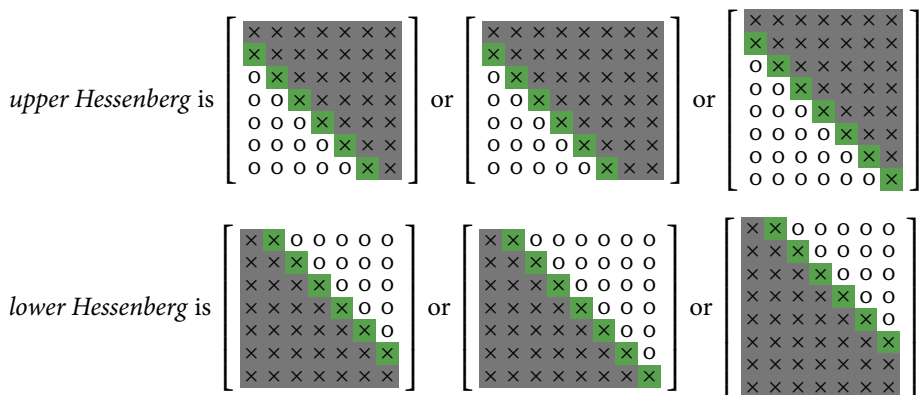
These are often *almost square* with one additional row or column.

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

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

for a square or rectangular matrix B.

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 A be a square matrix, then A is an M-matrix if A^{-1} is
> elementwise non-negative.

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

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

matrix in Example **??** 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 $A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$. This matrix expresses the permutation

$2 \rightarrow 1, 3 \rightarrow 2, 1 \rightarrow 3$. We can see this by: $A \begin{bmatrix} 0.5 \\ -0.5 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 1 \\ 0.5 \end{bmatrix}$.

3.9 ORTHOGONAL MATRICES

A matrix is orthogonal if $A^T A = I$. We'll get into why orthogonal matrices are special soon.

Examples The identity matrix I is orthogonal. We'll see more about orthogonal matrices soon – it's a very special structure!

3.10 EXERCISES

1. Consider the matrix from example 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 $A = \begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix}$. Let $\mathbf{x} = \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix}$ be an eigenvector of A with eigenvalue λ. Show that there is an eigenvector $\mathbf{w}$ with eigenvalue $-\lambda$ also involving a combination of $\mathbf{y}$ and $\mathbf{z}$.

A MATRIX MODEL OF VIRAL SPREAD

<div style="text-align: right">*4*</div>

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!

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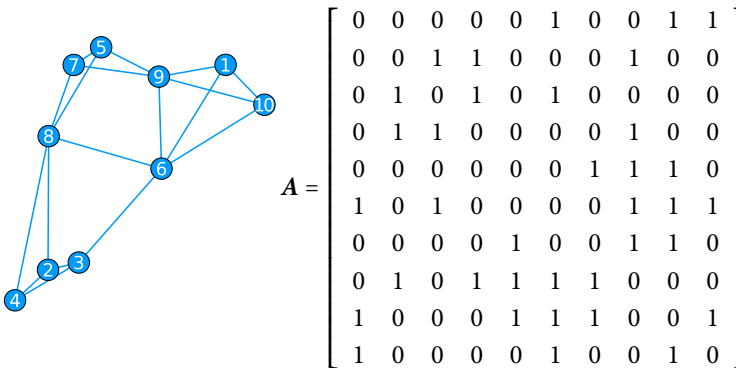
[1] In standard mathematical epidemiology literature, this would be a susceptible, infected (SI) model. This is highly simplistic!

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

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 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]

[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!

4.1   A FIRST MODEL THAT ISN'T QUITE RIGHT, BUT IS A USEFUL START.

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.

Each contact j in the neighbors of i infects i based on a simple random trial that occurs with probability $\rho P(j$ 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 A that is easiest to explain in terms of code. Let $\mathbf{x}$ be the vector $P(i$ is infected at time $t)$ for all i and $\mathbf{y}$ be the vector $P(i$ is infected at time $t + 1)$ for all i. Then

```
1  function evolve(x::Vector, p::Real, A::AbstractMatrix)
2    log_not_infected = log.(1 .- p.*x)
3    y = 1 .- exp.(A*log_not_infected)
4    y = max.(y, x)
5  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

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

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]

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

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 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 $\rho P(j$ 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

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

This suggests an even simpler iteration.

```
1  function evolve_approx(x::Vector,  p::Real, A::AbstractMatrix)
2    y = p.*(A*x)
3  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

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

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

$P(i$ is infected at time $t + 1)$

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

$$\underbrace{\phantom{= \Big(1 - \prod_{j \in \text{neighbors}(i)} \big(1 - \rho P(j)\big)\Big)}}_{\text{infected via neighbors}} \qquad \underbrace{}_{\text{actually infected in the previous step.}}$$

and

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

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

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

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

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

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

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

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

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

4.3   THE EIGENANALYSIS

As we will see in this class, the *eigenvectors of A* determine the behavior of both powers of ρA and $(\rho A + I)$ 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 $n_i^{(t)} = P(i$ 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

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

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 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 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

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

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

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

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

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:

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

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 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].

CANDYLAND & WORKING WITH SPARSE MATRICES

5

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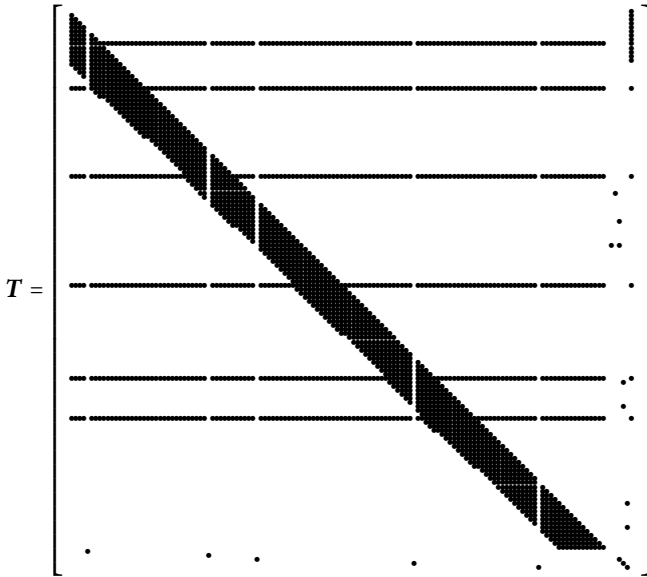
In this case, the game of Candyland can be modeled with a 140×140 matrix T.[1] If we show the matrix with a small • for each nonzero entry, then it looks like[2]

[1] The data files to recreate T are available on the the course website.

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

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 $T = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & \dots & \mathbf{t}_{140} \end{bmatrix}$ 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:

Probability that player is in state i after two steps

$$= \sum_k \text{Probability that player is in state } k \text{ after one step and moves from } k \text{ to } i.$$

$$= \sum_k T(i, k) \mathbf{t}_{140}(k)$$

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

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

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:

Probability that player is in state i after three steps

$$= \sum_k \text{Probability that player is in state } k \text{ after two steps and moves from } k \text{ to } i.$$

$$= \sum_k T(i,k)\mathbf{p}_2(k).$$

Again, by grouping everything together, we get:

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

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

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

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:

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

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

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

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

Hence, the expected length of the Candyland game is:

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

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 p = e₁₄₀ because we start in state 140.
EX ← 0
For length = 1 to maximum game length considered
    p ← Tp
    EX ← EX + length · p₁₃₄
return EX
```

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

[4] In Julia, the code is

```
1  function candylandlength(T, ma
2    n = size(T,1)
3    (p = zeros(n))[140] = 1
4    ex = 0.0
5    for l=1:len
6      p = T*p
7      ex += length*p[134]
8    end
9    return ex
10 end
```

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.

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

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

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

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

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

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

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:

```
1  function indexed_sparse_matrix_vector_product(x,I,J,V,m,n)
2    y = zeros(m)
3    for nzi=1:length(I)
4      i,j,v = I[nzi], J[nzi], V[nzi]
5      y[i] += v*x[j]
6    end
7    return y
8  end
```

This algorithm can be translated to many other languages too.

5.3 ELIMINATING REDUNDANT DATA STOR-AGE: COMPRESSED SPARSE ROW AND COL-UMN 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]
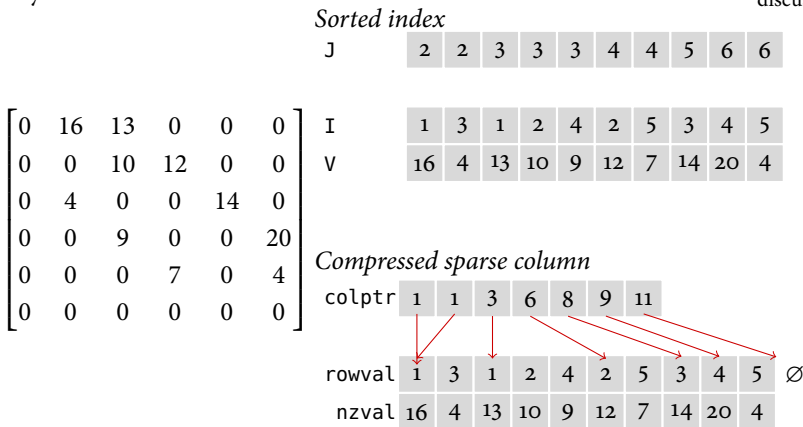
This figure shows that when the data are sorted by increasing column index. Then there are multiple values with the same column in adjacent entries of the J array. We can *compress* these into a list of pointers. This means that we create a new array called `colptr` that stores the starting index for all the entries in I, V arrays associated with a given column.

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

This means if $\text{colptr}[j] = \text{colptr}[j+1]$ 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]

```julia
1  function indexed_sparse_matrix_vector_product(x,colptr,rowval,nzval,m,n)
2    y = zeros(m)
3    for j=1:n
4      for nzi=colptr[j]:colptr[j+1]-1
5        i,v = rowval[nzi], nzval[nzi]
6        y[i] += v*x[j]
7      end
8    end
9    return y
```

```
10    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

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

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

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

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	
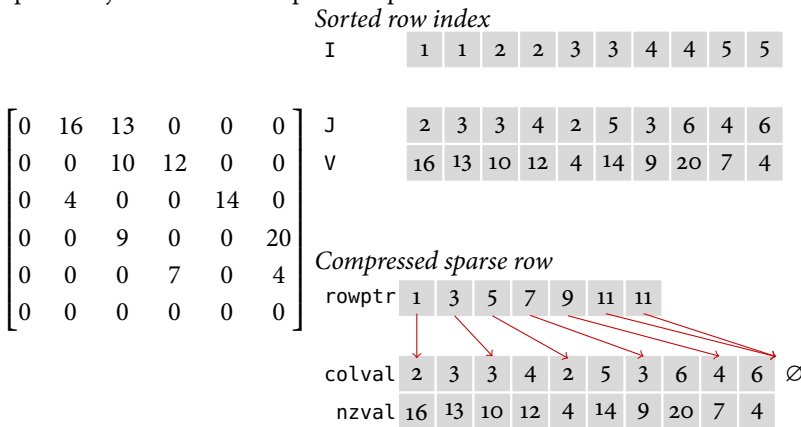
Entries of row i are stored in $\dfrac{\texttt{colval}\big[\texttt{rowptr}[i]\big]...\texttt{colval}\big[\texttt{rowptr}[j+1]-1\big]}{\texttt{nzval}\big[\texttt{rowptr}[j]\big]...\texttt{nzval}\big[\texttt{rowptr}[j+1]-1\big]}$.

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 **y** happen in the same index.

```
1  function indexed_sparse_matrix_vector_product(x,rowptr,colval,nzval,m,n)
2    y = zeros(m)
3    for i=1:m
4      yi = 0.0
5      for nzi=rowptr[j]:rowptr[j+1]-1
6        j,v = colval[nzi], nzval[nzi]
7        yi += v*x[j]
8      end
9      y[i] = yi
```

```
10      end
11      return y
12    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 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

[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.

[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}}[]`.

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.

MATRIX & VECTOR NORMS

6

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.

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

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*

6.1   VECTOR NORMS

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.

[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.

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:

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

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

[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.

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

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 (*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:

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

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

Theorem 6.4

Informally, all vector 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 $C_1 \leq C_2$ such that

$$C_1 f(\mathbf{x}) \leq g(\mathbf{x}) \leq C_2 f(\mathbf{x}).$$

Note that these constants can depend on the dimension n.

PROOF This is just a sketch, but the essence of the result is here; it requires just a little bit more analysis to fully state. They key we look at how these functions map unit-vectors to get the extreme values. Everything else follows from straightforward (but not simple) analysis. The values in the theorem are:

$$C_1 = \begin{array}{ll} \text{maximize} & f(\mathbf{x}) \\ _{\mathbf{x}} \\ \text{subject to} & g(\mathbf{x}) \leq 1 \end{array} \quad \text{and } C_2 = \begin{array}{ll} \text{maximize} & g(\mathbf{x}) \\ _{\mathbf{x}} \\ \text{subject to} & f(\mathbf{x}) \leq 1 \end{array}.$$

Note that because of the scalar property, the extreme must occur on a boundary of the feasible set, i.e. where $f(x) = 1$. (If this isn't obvious, a quick proof by contradiction should help: If there is a point inside that gets the max, then we can scale it and make $f(\mathbf{x})$ (say) bigger and also $g(\mathbf{x})$ bigger, so it can't be optimal.) This is why we get the values of C_1 and C_2 in the above proof. ■

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

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

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

Consequently, suppose, for a vector norm $f(\mathbf{x})$, you show that $f(\mathbf{x}_k - \mathbf{y}) \to 0$. Then we know that $C_2 f(\mathbf{x}) \ge g(\mathbf{x})$ and also that $C_2 f(\mathbf{x}_k - \mathbf{y}) \to 0$. Since $g(\mathbf{x}) \ge 0$, 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 $X \in \mathbb{R}^{m \times n}$ is any function $f(X) \to \mathbb{R}$ that satisfies:
>
> 1. $f(X) \ge 0$ (*non-negative*)
> 2. $f(X) = 0$ if and only if $X = 0$ (*zero-sensitive*)
> 3. $f(\alpha X) = |\alpha| f(X)$ for any scalar α (*1-homogeneous*)
> 4. $f(X + Y) \le f(X) + f(Y)$ (*triangle inequality*).

6.2.1 Element-wise norms

Note that if vec X is any way of turning X into a vector by organizing the mn elements of X into a single array, then $f(\text{vec}(X))$ 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
>
> $$\|X\|_F = \sqrt{\sum_{ij} |X_{ij}|^2} = \|\text{vec}(X)\|_2 = \sqrt{\text{trace}(A^T A)}.$$

Here, we used $\text{trace}(A) = \sum_{i=1}^{\min(m,n)} A_{i,i}$, 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:

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

Proof that $f(X)$ is a matrix norm

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

$$f(X + Y) = \underset{x \neq 0}{\text{maximize}} \ f((X + Y)x)/f(x) \ \leq \ \underset{x \neq 0}{\text{maximize}} \ f(Xx)/f(x) + f(Yx)/f(x)$$

$$\leq \ \underset{x \neq 0}{\text{maximize}} \ f(Xx)/f(x) \ + \ \underset{x \neq 0}{\text{maximize}} \ f(Yx)/f(x)$$

$$\leq f(X) + f(Y)$$

The operator induced norms are harder to reason about.

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

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

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

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

which is the maximum row 1-norm.

Here's my picture to remember these.

$\|A\|_1 =$	$\|A\|_1 \neq$	$\|A\|_\infty \neq$	$\|A\|_\infty =$
looks okay	looks wrong	looks wrong	looks okay
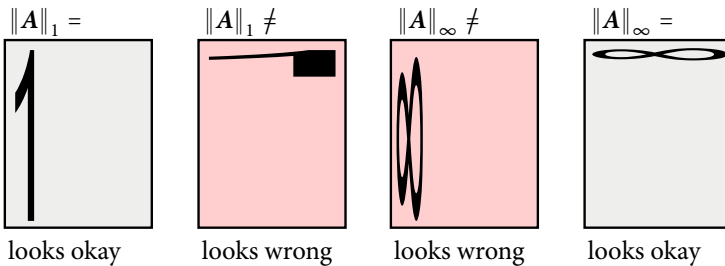
6.2.3 Additional matrix norms

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 $m \geq n$.

> **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
> **A**. Then the *nuclear norm* also called the *trace norm* is the

matrix norm based on the function

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

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

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

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 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

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

Note that both Shatten and Ky-Fan norms are *vector norms* applied to the vector of singular values **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 NORM PROPERTIES

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

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

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

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

6.3.2 *Submultiplicative*

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

$$f(Ax) \leq f(A)f(x)$$

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

Definition 6.11
A matrix-norm $f(A)$ is sub-multiplicative if:

$$f(AB) \leq f(A)f(B).$$

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

6.4   EXERCISES

1. Let $x \in \mathbb{C}^n$. Decompose x into the real and imaginary parts: $x = y + iz$ where $y \in \mathbb{R}^n$ and $z \in \mathbb{R}^n$. Show that $\|x\|_2 = \| \begin{bmatrix} y \\ z \end{bmatrix} \|_2$.
2. Let P be a permutation matrix. So Px reorders the elements of x. Find a vector-norm function on length 2 vectors where $\|x\| \neq \|Px\|$.
3. (This requires knowledge of the SVD.) Show that the Schatten and Ky-Fan norms are orthogonally invariant.
4. Consider the following function:

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

   (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:

   $$g(A) = \sigma f(A)$$

   is a sub-multiplicative matrix-norm.

THE FIELD OF VALUES

6

6.5 IMPORTANT CLASSES OF MATRICES

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

EXAMPLE 6.13 A circulant matrix is a non-symmetric, but normal matrix, such as

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

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.

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

Then the eigenpairs of 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. $\qquad \blacklozenge$

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

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

in fact, any triangular matrix with a constant diagonal is non-normal and non-diagonalizable. $\qquad \blacklozenge$

We have the following set of implications

Symmetric or Hermitian $\quad\Rightarrow\quad$ Normal $\quad\Rightarrow\quad$ Diagonalizable

1. Let A be a symmetric matrix and let D^{-1} be a diagonal matrix. Show that any matrix $D^{-1}mA$ is diagonalizable.

2. This example is from Trefthen and Embree's book on non-normal matrices.[4] Let

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

Then A is diagonalizable.

SIMPLE ITERATIVE ALGORITHMS

II

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 M be a matrix with $\rho(M) < 1$, them the sequence of matrices
> $$M^k \to 0 \text{ as } k \to \infty.$$

Does Steepest Descent obey this? Not quite.

$$
\begin{aligned}
\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \alpha^{(k)}\mathbf{g}^{(k)} \\
&= \mathbf{x}^{(k)} - \alpha^{(k)}(A\mathbf{x}^{(k)} - \mathbf{b}) \\
&= (I - \alpha^{(k)}A)\mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{b}
\end{aligned}
\tag{6.1}
$$

So for any solution and any $\alpha^{(k)}$, we have $A\mathbf{x} = \mathbf{b}$ implies that

$$\mathbf{x}^{(k+1)} - \mathbf{x} = (I - \alpha^{(k)}A)\mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{b} = (I - \alpha^{(k)}A)\mathbf{x}^{(k)} + \alpha^{(k)}(A\mathbf{x}) - \mathbf{x} = (I - \alpha^{(k)}A)(\mathbf{x}^{(k)} - \mathbf{x})$$

7

SIMPLE ITERATIVE METHODS

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:

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

or as I like to think of them

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

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.

7.1 REVIEW OF LINEAR SYSTEMS OF EQUATIONS

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

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

represents a set of equations

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

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

[1] This section should be a review.

infinite set of solutions. And if $m > n$, the system is called *over determined* 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 $\mathbf{y}$

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

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

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

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

$$-x + y - z = 2$$
$$-x + y - z = 3.$$

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

Here is a fun case to consider. Let $A = \mathbf{y}\mathbf{y}^T$. When does $A\mathbf{x} = \mathbf{b}$ have a solution? When does it have no solution? Describe a procedure to find the 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(A) is n (or m since $m = n$)
· A is invertible
· the columns of A are linearly independent
· the rows of A are linearly independent
· the determinant of A is one
· the eigenvalues of A are all non-zero

· the singular values of A are all non-zero.

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

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

7.2 A FIRST METHOD

This isn't the order I'm hoping to do these in eventually, but because of the homework, I want to go over this 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.

> **Theorem 7.1 (The Neumann Series)**
> If that $A^k \to 0$, then
>
> $$\sum_{k=0}^{\infty} A^k = (I - A)^{-1}$$

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

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

Consequently,

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

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

7.3 OVERVIEW

Over the next few classes, we are going to see a bunch of different perspectives on this same algorithm.

7.4   CHECKING A POSSIBLE SOLUTION

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

Let $A\mathbf{x} = \mathbf{b}$ 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:

$$\text{error} = \mathbf{y} - \mathbf{x}$$
$$\text{error} = \|\mathbf{y} - \mathbf{x}\|$$
$$\text{relative error} = \|\mathbf{y} - \mathbf{x}\|/\|\mathbf{x}\|$$

$$\text{residual} = \mathbf{b} - A\mathbf{y}$$
$$\text{residual} = A\mathbf{y} - \mathbf{b}$$
$$\text{residual} = \|A\mathbf{y} - \mathbf{b}\|$$
$$\text{relative residual} = \|A\mathbf{y} - \mathbf{b}\|/\|\mathbf{b}\|$$

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 7.2**
> Let $\mathbf{y}$ be any vector, then the *error* $\mathbf{e} = \mathbf{y} - \mathbf{x}$ and *residual* $\mathbf{r} = A\mathbf{x} - \mathbf{b}$ 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 7.3**
> Using the notation from Theorem 7.2, let $\|\cdot\|$ be a sub-multiplicative norm.[2] Then
>
> $$\|\mathbf{e}\| \leq \|A^{-1}\|\,\|\mathbf{r}\|.$$

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

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.

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
$1e^{0}$	$5e^{-1}$	$5e^{-1}$	$4e^{-1}$	$4e^{-1}$	$3e^{-1}$	$3e^{-1}$	$3e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$9e^{-2}$	$5e^{-2}$	$3e^{-2}$	$2e^{-2}$	$1e^{-2}$	$7e^{-3}$	$4e^{-3}$	$3e^{-3}$	$2e^{-5}$	$1e^{-7}$	$8e^{-10}$	$5e^{-12}$
$1e^{0}$	$1e^{0}$	$8e^{-1}$	$8e^{-1}$	$6e^{-1}$	$6e^{-1}$	$5e^{-1}$	$5e^{-1}$	$5e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$7e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$5e^{-3}$	$4e^{-5}$	$2e^{-7}$	$2e^{-9}$	$1e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$7e^{-1}$	$7e^{-1}$	$6e^{-1}$	$4e^{-1}$	$2e^{-1}$	$1e^{-1}$	$9e^{-2}$	$5e^{-2}$	$3e^{-2}$	$2e^{-2}$	$1e^{-2}$	$7e^{-3}$	$5e^{-5}$	$3e^{-7}$	$2e^{-9}$	$1e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$6e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$6e^{-5}$	$4e^{-7}$	$2e^{-9}$	$2e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$6e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$6e^{-5}$	$4e^{-7}$	$2e^{-9}$	$2e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$6e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$6e^{-5}$	$4e^{-7}$	$2e^{-9}$	$2e^{-11}$
$1e^{0}$	$1e^{0}$	$1e^{0}$	$9e^{-1}$	$9e^{-1}$	$8e^{-1}$	$8e^{-1}$	$7e^{-1}$	$7e^{-1}$	$6e^{-1}$	$4e^{-1}$	$2e^{-1}$	$1e^{-1}$	$9e^{-2}$	$5e^{-2}$	$3e^{-2}$	$2e^{-2}$	$1e^{-2}$	$7e^{-3}$	$5e^{-5}$	$3e^{-7}$	$2e^{-9}$	$1e^{-11}$
$1e^{0}$	$1e^{0}$	$8e^{-1}$	$8e^{-1}$	$6e^{-1}$	$6e^{-1}$	$5e^{-1}$	$5e^{-1}$	$5e^{-1}$	$5e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$7e^{-2}$	$4e^{-2}$	$2e^{-2}$	$1e^{-2}$	$9e^{-3}$	$5e^{-3}$	$4e^{-5}$	$2e^{-7}$	$2e^{-9}$	$1e^{-11}$
$1e^{0}$	$5e^{-1}$	$5e^{-1}$	$4e^{-1}$	$4e^{-1}$	$3e^{-1}$	$3e^{-1}$	$3e^{-1}$	$3e^{-1}$	$2e^{-1}$	$1e^{-1}$	$9e^{-2}$	$5e^{-2}$	$3e^{-2}$	$2e^{-2}$	$1e^{-2}$	$7e^{-3}$	$4e^{-3}$	$3e^{-3}$	$2e^{-5}$	$1e^{-7}$	$8e^{-10}$	$5e^{-12}$
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.

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.
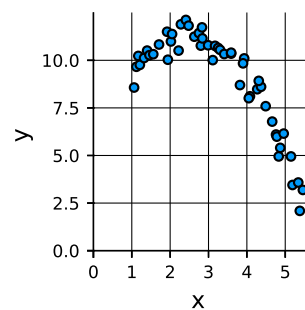
FIGURE 7.1 – We get a least squares problem to find a quadratic fit to this data $c_3 x^2 + c_2 x + c_1$. This gives a 3×3 linear system via the normal equations.

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	$4e^{2}$	$-3e^{6}$	$4e^{10}$	$-4e^{14}$	$5e^{18}$	$-6e^{22}$	$8e^{26}$	$-9e^{30}$	$1e^{35}$	$7e^{75}$	$5e^{116}$	$3e^{152}$	$2e^{198}$	$1e^{238}$	$8e^{279}$	NaN
0	$1e^{3}$	$-1e^{7}$	$2e^{11}$	$-2e^{15}$	$2e^{19}$	$-3e^{23}$	$3e^{27}$	$-4e^{31}$	$3e^{35}$	$3e^{76}$	$2e^{117}$	$1e^{158}$	$8e^{198}$	$5e^{238}$	$3e^{280}$	NaN
0	$4e^{3}$	$-6e^{7}$	$7e^{11}$	$-8e^{15}$	$1e^{20}$	$-1e^{24}$	$2e^{28}$	$-2e^{32}$	$2e^{36}$	$1e^{77}$	$8e^{115}$	$5e^{158}$	$4e^{199}$	$2e^{240}$	$2e^{281}$	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
$5e^{2}$	$-3e^{6}$	$4e^{10}$	$-4e^{14}$	$5e^{18}$	$-6e^{22}$	$8e^{26}$	$-9e^{30}$	$1e^{35}$	$-1e^{39}$	$9e^{79}$	$6e^{120}$	$4e^{162}$	$2e^{203}$	$1e^{243}$	$1e^{384}$	NaN
$1e^{3}$	$-1e^{7}$	$2e^{11}$	$-2e^{15}$	$2e^{19}$	$-3e^{23}$	$3e^{27}$	$-4e^{31}$	$3e^{35}$	$-6e^{39}$	$4e^{80}$	$2e^{121}$	$1e^{163}$	$1e^{203}$	$4e^{243}$	$4e^{384}$	NaN
$5e^{3}$	$-6e^{7}$	$7e^{11}$	$-8e^{15}$	$1e^{20}$	$-1e^{24}$	$2e^{28}$	$-2e^{32}$	$2e^{36}$	$-2e^{40}$	$2e^{81}$	$1e^{122}$	$7e^{162}$	$4e^{203}$	$3e^{244}$	$2e^{385}$	NaN

In this case, we can compute $\rho(I - A) = 12047.41494842964$, 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 $\rho(I - A) > 1$. 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:

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

Then the iteration is:

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

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 $\rho(I - \alpha A) < 1$. Let λ be an eigenvalue of A. This means we need that $|1 - \alpha\lambda| < 1$ for all eigenvalues of A.

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

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

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

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

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

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

This is because if 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 $f(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1 - \alpha)f(\mathbf{y})$. A global minimizer is any point $\mathbf{x}^*$ where $f(\mathbf{x}^*) \leq f(\mathbf{x})$ 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.

> **Theorem 7.4**
> Let $f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{x}$. Then $f(\mathbf{x})$ is convex if A is symmetric positive definite.

There is a stronger result to prove here too.

PROOF From the definition

$$f(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) = (\alpha\mathbf{x} + (1 - \alpha)\mathbf{y})^T A(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) - (\alpha\mathbf{x} + (1 - \alpha)\mathbf{y})^T\mathbf{b}$$

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

∎

STEEPEST DESCENT *&* GRADIENT DESCENT

8

8.1 LINEAR SYSTEMS AND QUADRATIC FUNCTION MINIMIZATION

We are studying quadratic function minimization because this turns out to a good way to understand how to solve $A\mathbf{x} = \mathbf{b}$ for symmetric positive definite matrices 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 $A\mathbf{x} = \mathbf{b}$*

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

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

This is because if 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.

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

PROOF   From the definition

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

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

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

Note that we can simplify this to

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

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

8.1.3   *The gradient*

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

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,

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

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

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

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

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

8.1.4 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 $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x}$ where $\|\mathbf{x} - \mathbf{x}^*\| \leq \varepsilon$ 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: $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x}$.[1]

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

> *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.

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

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

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

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

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

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

■

8.1.6 *Finding the minimizer*

*If the gradient is not zero, then we can always
reduce the function by moving a sufficiently 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!

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

For the moment, we'll let

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

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

So if this result is going to be true, we need $(\alpha/2\mathbf{g}^T A\mathbf{g} + \alpha\mathbf{g}^T\mathbf{g})$ for α small enough. Let

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

Hence, $\alpha/2\mathbf{g}^T A\mathbf{g} \leq \rho\alpha/2\mathbf{g}^T\mathbf{g}$. Thus, if $\rho\alpha/2 \leq 1$ or $\alpha \leq 2/\rho$ we have

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

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

$$\text{Richardson} \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \underbrace{(\mathbf{b} - A\mathbf{x}^{(k)})}_{\text{residual}}$$

$$\text{Steepest Descent} \quad \mathbf{x}^{(k)} = \mathbf{x}^{(k)} - \alpha \underbrace{\mathbf{g}(\mathbf{x})}_{\text{gradient}} = \mathbf{x}^{(k)} - \alpha(A\mathbf{x} - \mathbf{b}).$$

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 $A\mathbf{x} = \mathbf{b}$ (which was based on the Neumann series).

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

8.2.3 *A coordinate-wise strategy.*

8.3 EXERCISES

1. (I'm not sure if this is true, and it could be difficult.). Let $A\mathbf{x} = \mathbf{b}$ be a diagonally dominant M matrix, but where A is not symmetric. This means that $A^{-1} \geq 0$. Suppose also that $\mathbf{b} \geq 0$. Develop an algorithm akin to steepest descent for this problem. Ideas include looking at functions like $f(x) = \mathbf{e}^T A\mathbf{x} - \mathbf{e}^T \mathbf{b}$

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

9

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

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

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

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

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

at a solution.[1]

Then one strategy to *improve* an inaccurate solution to $A\mathbf{x} = \mathbf{b}$ 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

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

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

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

update x_2 and then we'll use row 2 to update x_1. The update looks like:

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

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

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 $\mathbf{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ after a few iterations.

9.2 THE GENERAL EXAMPLE AND THE JA-COBI 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 M. Let 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 M.[3] We can use any matrix 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:

$$x_1^{\text{new}} = \frac{1}{A_{M_1,1}}\left(b_{M_1} - \sum_{j \neq 1} A_{M_1,j} x_j\right)$$

$$\vdots$$

$$x_i^{\text{new}} = \frac{1}{A_{M_i,i}}\left(b_{M_i} - \sum_{j \neq i} A_{M_i,j} x_j\right)$$

$$\vdots$$

$$x_n^{\text{new}} = \frac{1}{A_{M_n,n}}\left(b_{M_n} - \sum_{j \neq n} A_{M_i,j} x_j\right)$$

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

[3] Technically, and looking ahead just a few paragraphs, the matrix M is a permutation matrix, we will use this insight in a moment!
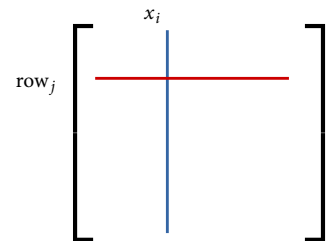
FIGURE 9.1 – A general setup for the equations described by rows of a system. We need $A_{j,i} \neq 0$ to update x_i based on equation j. We also cannot use equation j to update any other variables, nor can we use other equations to update x_i. The map matrix M encodes the set of updates.

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

$$\mathbf{x}^{\text{new}} = D_M^{-1} \text{some vector}$$

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 N_M be the matrix with entries:

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

$$\mathbf{x}^{\text{new}} = D_M^{-1}(\mathbf{b}_M - N_M \mathbf{x}^{\text{old}}).$$

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

```
1  """
2      jacobi_iteration_map_!(y,A,x,M) sets y to be the next Jacobi
3      iteration from x with map M
4  """
5  function jacobi_iteration(y,A,x,M=1:length(x)))
6    for i=1:length(x)
7      y[i] -= (b[M[i] - A[M[i],:]'*x - A[M[i],i]*x[i]) / A[M[i],i]
8    end
9  end
```

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

Most derivations of the Jacobi iteration assume that 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 P such that $P \odot A$ is non-singular to build the matrix M for the above iteration to work.[4]

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

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

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

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

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

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 M, let us simply *permute* the three rows of 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:

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

With this setup, we can easily write the iteration

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

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

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

We can also apply the same permutation to A as in

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

This corresponds with solving a linear system

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

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

$$D_M = \text{diagonal elements of } PA \text{ in a diagonal matrix.}$$

Then also,

$$N_M = PA - D_M.$$

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 P such that PA has non-zero diagonal elements. 2. Let D be the diagonal elements of PA. 3. Let N be the matrix $PA - D$

Then the Jacobi method implements the iteration:

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

This makes it much simpler 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 $P = I$.   ◆

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 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:

$$\mathbf{x}^{(k+1)} - \mathbf{x} = D_M^{-1}(\mathbf{b} - N_M\mathbf{x}^{(k)}) - \mathbf{x}.$$

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

$$\mathbf{x}^{(k+1)} - \mathbf{x} = D_M^{-1}(\mathbf{b} - N_M\mathbf{x}^{(k)} - \mathbf{b} + N_M\mathbf{x}) = -D_M^{-1}N_M(\mathbf{x}^{(k)} - \mathbf{x}).$$

Consequently,

$$\mathbf{x}^{(k+1)} = (-D_M^{-1}N_M)^{k+1}(\mathbf{x}^{(0)} - \mathbf{x}).$$

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

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

```
1  """
2      jacobi_iteration!(y,A,x,M) updates x "like" Jacobi, but in-place, with map M
3  """
4  function jacobi_iteration!(A,b,x,M=1:length(x))
5    for i=1:length(x)
6      x[i] -= (b[M[i]] - A[M[i],:]'*x - A[M[i],i]*x[i]) / A[M[i],i]
7    end
8  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.

$$x_1 = \frac{1}{A_{M_1,1}} \left(b_{M_1} - \sum_{j \neq M_1} A_{M_1,j} x_j\right)$$

$$\dots$$

$$x_i^{\text{new}} = \frac{1}{A_{M_i,1}} \left(b_{M_i} - \sum_{j \neq M_i} A_{M_i,j} x_j\right)$$

$$\dots$$

9.8 ANALYZING GAUSS-SEIDEL

Test

9.9 THE GAUSS-SEIDEL AND STEEPEST DE-SCENT METHOD

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

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

$$x_1^{\text{new}} = \left(b_1 - \sum_{j>1} A_{1,j} x_j^{\text{old}}\right)$$

$$\dots$$

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

Recall the iteration for coordinate descent:

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

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:

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

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

EIGENVALUES & THE POWER METHOD

10

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 A. The most general definition of an eigenvalue of a matrix is a value λ such that $\det(A - \lambda I) = 0$. 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 A are critical directions in the quadratic function

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

that are invariant to transformations.

For a symmetric matrix A, then the eigenvalues of A are the *stationary points* of the following optimization problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximize}} \quad & \mathbf{x}^T A \mathbf{x} \\ \text{subject to} \quad & \|\mathbf{x}\|^2 = 1 \end{aligned} \tag{10.1}$$

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 $\mathbf{x}^T A \mathbf{x}$ and the constraint $\|\mathbf{x}\|^2 = 1$

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

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 EIGEN-VALUES

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:

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

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

$$\mathbf{x}^{(k+1)} = (\boldsymbol{A}\mathbf{x})/\|\mathbf{x}\|.$$

[1] — TODO – work out this derivation more. Can we show that $\gamma_k \to \infty$ is a natural step?

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

$$\mathbf{x}^{(k+1)} = \left(A\mathbf{x}^{(k)}\right)/\|\mathbf{x}^{(k)}\|.$$

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

$$\lambda^{(k)} = {\mathbf{x}^{(k)}}^{T} A\mathbf{x}^{(k)}.$$

This quantity can often be computed with minimal overhead because we need to compute the vector $A\mathbf{x}^{(k)}$ to get the next iterate of the power method.

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 $\mathbf{x}^{(k)} = M^k \mathbf{x}^{(0)}$ 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 $\mathbf{x}^{(k)} = A^k\mathbf{x}^{(0)}/\|A^k\mathbf{x}^{(0)}\|$.

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: $\mathbf{x}^{(k)} = A^k\mathbf{x}^{(0)}/\|A^k\mathbf{x}^{(0)}\|$. This also means that $\mathbf{x}^{(k)} = \rho_k A^k\mathbf{x}^{(0)}$ for some scalar ρ_k. Thus,

$$\mathbf{x}^{(k+1)} = \rho_k A^{k+1}\mathbf{x}^{(k)}/\|\rho_k A^{k+1}\mathbf{x}^{(k)}\|.$$

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

$$A\mathbf{x}^{(k)} - \lambda^{(k)}\mathbf{x}^{(k)} \approx 0$$

FINITELY TERMINATING ALGORITHMS III

III · FINITELY TERMINATING ALGORITHMS

ELIMINATION METHODS FOR LINEAR SYSTEMS

11

Thus far, we've seen methods that solve $A\mathbf{x} = \mathbf{b}$ 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 $A\mathbf{x} = \mathbf{b}$ into another system $B\mathbf{y} = \mathbf{d}$, 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 $A\mathbf{x} = \mathbf{b}$, 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 $A\mathbf{x} = \mathbf{b}$ then we can write this out and highlight the first row as follows:[1]

$$A = \begin{bmatrix} \alpha & \mathbf{c}^T \\ \mathbf{d} & R \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} \beta \\ \mathbf{f} \end{bmatrix}.$$

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

[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.

expression

$$\gamma\mathbf{d} + R\mathbf{y} = \mathbf{f}.$$

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 $\mathbf{y}$

$$\gamma(\mathbf{y})\mathbf{d} + R\mathbf{y} = \frac{1}{\alpha}\left(\beta - \mathbf{c}^T\mathbf{y}\right)\mathbf{d} + R\mathbf{y} = \mathbf{f}.$$

By re-arranging the equations, we arrive at the following linear system:

$$(R - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T)\mathbf{y} = \mathbf{f} - \frac{\beta}{\alpha}\mathbf{d}.$$

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

$$(R - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T)\mathbf{y} = \begin{bmatrix} -4.5 & -3 & -1.5 \\ 3 & -6 & -2 \\ -1 & 3 & 1 \end{bmatrix}\mathbf{y} = \begin{bmatrix} 2 \\ 3 \\ -2 \end{bmatrix}$$

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 A was $n \times n$, then this method takes the system

$$(A, \mathbf{b}) \quad \text{to} \quad (R - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T, \mathbf{f} - \frac{\beta}{\alpha}\mathbf{d})$$

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.

```
1  function elimination_solve(A::Matrix, b::Vector)
2    m,n = size(A)
3    @assert(m==n, "the system is not square")
4    @assert(n==length(b), "vector b has the wrong length")
5    if n==1
6      return [b[1]/A[1]]
7    else
8      R = A[2:end,2:end]
9      c = A[1,2:end]
10     d = A[2:end,1]
11     alpha = A[1,1]
12     y = elimination_solve(R-d*c'/alpha,
```

```
13                  b[2:end]-b[1]/alpha*d)
14          gamma = (b[1] - c'*y)/alpha
15          return pushfirst!(y,gamma)
16      end
17  end
```

This idea is called *variable elimination*. We elminate the variable y from the system of equation $A\mathbf{x} = \mathbf{b}$ by solving for its expression and them substituting that solution into the rest of the equations.

11.2 VARIABLE ELIMINATION AS A MATRIX EXPRESSION

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. Again, let $A = \begin{bmatrix} \alpha & \mathbf{c}^T \\ \mathbf{d} & R \end{bmatrix}$. Then

$$\underbrace{\begin{bmatrix} 1 & 0 \\ -\mathbf{d}/\alpha & I \end{bmatrix}}_{=L_1} \underbrace{\begin{bmatrix} \alpha & \mathbf{c}^T \\ \mathbf{d} & R \end{bmatrix}}_{=A} \underbrace{\begin{bmatrix} 1 & -\mathbf{c}^T/\alpha \\ 0 & I \end{bmatrix}}_{=U_1} = \begin{bmatrix} \alpha & 0 \\ 0 & R - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T \end{bmatrix}.$$

Note that L_1 is a non-singular matrix of the form:

$$L_1 = I - \mathbf{v}e_1 \text{ where } v_1 = 0.$$

This means that $L_1^{-1} = I + \mathbf{v}e_1$, which can be verified because $L_1 L_1^{-1} = I$. Likewise, $U_1 = I - e_1\mathbf{u}^T$ where $\mathbf{u}_1 = 0$. It's inverse is $I + e_1\mathbf{u}^T$ as well. Using these matrices, we can transform:

$$A\mathbf{x} = \mathbf{b} \rightarrow L_1 A U_1 U_{-1}\mathbf{x} = L_1\mathbf{b}.$$

If we expand this block-wise, then we get:

$$\begin{bmatrix} \alpha & 0 \\ 0 & R - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T \end{bmatrix} \begin{bmatrix} \gamma + \frac{1}{\alpha}\mathbf{c}^T\mathbf{y} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\mathbf{d}/\alpha & I \end{bmatrix} \begin{bmatrix} \beta \\ \mathbf{f} \end{bmatrix},$$

which is exactly our reduced system.

Consequently, we can express our entire sequence of reductions as follows:

$$L_{n-1}L_{n-2}\cdots L_1 A U_1 U_2 \cdots U_{n-1} = \begin{bmatrix} \alpha_1 & & & \\ 0 & \alpha_2 & & \\ 0 & 0 & \ddots & \\ 0 & 0 & 0 & \alpha_n \end{bmatrix} = D$$

or

$$A = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1} D U_{n-1}^{-1} U_{n-2}^{-1} \cdots U_1^{-1}.$$

It turns out that these elimination matrix L_i^{-1} and 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 $L_i^{-1} = (I + \mathbf{v}\mathbf{e}_i)$ with $v_1 = v_2 = \ldots = v_i = 0$ and so

$$L_i^{-1} L_j^{-2} = (I + \mathbf{v}_i \mathbf{e}_i^T)(I + \mathbf{v}_j \mathbf{e}_j^T) = I + \mathbf{v}_i \mathbf{e}_i^T + \mathbf{v}_j \mathbf{e}_j^T + \mathbf{v}_i \mathbf{e}_i^T \mathbf{v}_j \mathbf{e}_j^T = I + \mathbf{v}_i \mathbf{e}_i^T + \mathbf{v}_j \mathbf{e}_j^T \text{ when } i < j.$$

This enables us to quickly compute these as follows:

```
1   function myreduce_all(A::Matrix)
2     A = copy(A) # save a copy
3     n = size(A,1)
4     L = Matrix(1.0I,n,n)
5     U = Matrix(1.0I,n,n)
6     d = zeros(n)
7     for i=1:n-1
8       alpha = A[i,i]
9       d[i] = alpha
10      U[i,i+1:end] = A[i,i+1:end]/alpha
11      L[i+1:end,i] = A[i+1:end,i]/alpha
12      A[i+1:end,i+1:end] -=
13        A[i+1:end,i]*A[i,i+1:end]'/alpha
14    end
15    d[n] = A[n,n]
16    return L,U,d
17  end
18  L,U,d = myreduce_all(A)
19  L*Diagonal(d)*U - A
```

This is what is most commonly called the *LU* decomposition of a matrix.

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

$$0x + y = 5$$

$$x = \frac{5 - y}{0}$$

break!

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.[3] In this case, we can simply swap rows

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} 6 \\ 5 \end{bmatrix}.$$

After which we have

$$x + y = 6 \rightarrow x = 6 - y$$

$$y = 5$$

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 11.2 (Pivoting)**
> Pivoting reorders the equations (rows) of 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 11.3**
> If 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

$$L_{k-1} L_{k-2} \cdots L_1 A U_1 U_2 \cdots U_{k-1} = \begin{bmatrix} d_1 & & & & \\ & d_2 & & & \\ & & \ddots & & \\ & & & d_{k-1} & \\ & & & & R \end{bmatrix}$$

By our assumption, we are in the scenario where there is no equation (row) of R to eliminate the kth variable. The kth variable is involved in the first column of R. This means that the first column of R is entirely zero. This implies that R is singular because it has a column that is entirely zero.

Now, A is non-singular, as are the products

$$L_{k-1} L_{k-2} \cdots L_1 \text{ and } U_1 U_2 \cdots U_{k-1}.$$

Consequently, the left-hand side is non-singular, which means the right hand side must be as well. However, our assumption implied that R was singular, which is how we arise at the contradiction. ∎

[3] 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.

This gives rise to the following algorithm for solving a system of linear equations.

```
1   function solve_with_simple_pivoting!(A::Matrix, b::Vector)
2     m,n = size(A)
3     @assert(m==n, "the system is not square")
4     @assert(n==length(b), "vector b has the wrong length")
5     if n==1
6       return [b[1]/A[1]]
7     else
8       # let's make sure we have an equation
9       # that we can eliminate!
10      α = A[1,1]
11      newrow = 1
12      if α == 0
13        for j=2:n # search the current column
14          if A[j,1] != 0
15            newrow = j
16            break
17          end
18        end
19        if newrow == 1
20          error("the system is singular")
21        end
22      end
23      # swap rows 1, and newrow
24      if newrow != 1
25        tmp = A[1,:]
26        A[1,:]  .= A[newrow,:]
27        A[newrow,:]  .= tmp
28        b[1], b[newrow] = b[newrow], b[1]
29      end
30      D = A[2:end,2:end]
31      c = A[1,2:end]
32      d = A[2:end,1]
33      α = A[1,1]
34      y = solve_with_simple_pivoting!(D-d*c'/α, b[2:end]-b[1]/α*d)
35      γ = (b[1] - c'*y)/α
36      return pushfirst!(y,γ)
37    end
38  end
```

11.4   PARTIAL PIVOTING

Add failure case example with singular matrix

SYMMETRIC POSITIVE DEFINITE SYSTEMS & VARIABLE ELIMINATION

12

12.1 SYMMETRIC & SYMMETRIC POSITIVE DEFINITE SYSTEMS

Note that if A is symmetric, then $\mathbf{d} = \mathbf{c}$ and hence $R - \frac{1}{\alpha}\mathbf{d}\mathbf{c}^T = R - \frac{1}{\alpha}\mathbf{c}\mathbf{c}^T$ is symmetric as well.

In fact, if A is symmetric positive definite, then $\mathbf{z}^T A \mathbf{z} > 0$ for any $\mathbf{z}$. We can show that $D - \frac{1}{\alpha}\mathbf{c}\mathbf{c}^T$ is *also* symmetric positive definite too! To do so, we will show that $\mathbf{g}^T R \mathbf{g} - \frac{1}{\alpha}(\mathbf{c}^T\mathbf{g})^2 > 0$ for any $\mathbf{g}$. We consider using a specially chosen vector $\mathbf{z}$ applied to the equation for A

$$\mathbf{z}^T A \mathbf{z} = \begin{bmatrix} \rho \\ \mathbf{g} \end{bmatrix}^T \begin{bmatrix} \alpha & \mathbf{c}^T \\ \mathbf{c} & R \end{bmatrix} \begin{bmatrix} \rho \\ \mathbf{g} \end{bmatrix} = \alpha\rho^2 + 2\rho\mathbf{c}^T\mathbf{g} + \mathbf{g}^T D \mathbf{g}.$$

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 $\mathbf{g}^T R \mathbf{g} - \frac{1}{\alpha}(\mathbf{c}^T\mathbf{g})^2$. To do so, let $\rho = -(\mathbf{c}^T\mathbf{g})/\alpha$. Then

$$\alpha\rho^2 + 2\rho\mathbf{c}^T\mathbf{g} + \mathbf{g}^T D \mathbf{g} = (\mathbf{c}^T\mathbf{g})^2/\alpha - 2(\mathbf{c}^T\mathbf{g})^2/\alpha + \mathbf{g}^T D \mathbf{g} = \mathbf{g}^T D \mathbf{g} - (\mathbf{c}^T\mathbf{g})^2/\alpha > 0$$

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 α = A[i,i] is non-zero at each step. In this case, we produce:

$$A = LDU.$$

Now, because A is symmetric, we have

$$A = A^T = (LDU)^T = U^T D L^T.$$

This strongly hints that $L = U^T$. This result is indeed correct, as can be verified by looking at the each step of the algorithm on a symmetric A and noting that we preserve symmetry at each step as shown above. However, in general, we cannot assume that any symmetric matrix can [1] T

89

be decomposed like this without pivoting. [1]here is a more general LDL^T factorization that can be computed in such cases.   A simple counter example is: $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$.

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

$$A = LDL^T.$$

Moreover, because 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 D. As shown above, after each elimination step, the matrix remains positive definite as well.

Because D is strictly positive, we can take the square root of each entry and compute

$$A = LD^{1/2}(D^{1/2}L)^T = FF^T \text{ or } F^TF.$$

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.

GENERAL VARIABLE ELIMINATION 13

This is a more general discussion on the ideas in the previous sections.
When we are elminating

ELIMINATION METHODS FOR LEAST SQUARES

14

14.1 THE SIMPLE ELIMINATION SOLVE

We can also use variable elimination for least squares problems. Consider

$$\text{minimize} \quad \|A\mathbf{x} - \mathbf{b}\|.$$

Partition $A = \begin{bmatrix} \mathbf{a} & C \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} \gamma \\ \mathbf{y} \end{bmatrix}$. Then

$$\text{minimize} \quad \|\gamma\mathbf{a} + C\mathbf{y} - \mathbf{b}\|.$$

We proceed as follows, suppose we know $\mathbf{y}$. Let $\mathbf{d} = C\mathbf{y} - \mathbf{b}$. 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

$$\text{minimize} \quad \|\gamma(\mathbf{y})\mathbf{a} + C\mathbf{y} - \mathbf{b}\| \quad \rightarrow \quad \text{minimize} \quad \left\|\frac{1}{\mathbf{a}^T\mathbf{a}}\mathbf{a}^T(C\mathbf{y} - \mathbf{b})\mathbf{a} + C\mathbf{y} - \mathbf{b}\right\|.$$

We can simplify this expression to

$$\text{minimize} \quad \left\|\left(I - \frac{1}{\mathbf{a}^T\mathbf{a}}\mathbf{a}^T\mathbf{a}\right)(C\mathbf{y} - \mathbf{b})\right\|.$$

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
```

14.2   A MATRIX VERSION

> **TODO** See if we can get something better here...

The matrix structure in this problem is already slightly apparent. Let $T = (I - \frac{1}{\mathbf{a}^T\mathbf{a}}\mathbf{a}^T\mathbf{a}^T)$. Then we have

$$\text{least-squares}(A, \mathbf{b}) \rightarrow \text{least-squares}(TAS, T\mathbf{b}).$$

Here 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 T is a special type of matrix called a *projection*. A projection matrix is any matrix where $T^2 = T$. It represents a projection onto a subspace, so $T^2 = T$ because the projection of a projection is the same projection. For this matrix T it's just a few lines of algebra to verify that $T^2 = T$.

> **TODO** Add these lines

This is a small issue, though, because $T\mathbf{a} = 0$ and so $TA = \begin{bmatrix} 0 & TC \end{bmatrix}$. 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 Q.

To be entirely precise, let $A = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix}$. Let $T_1, \ldots T_n$ be the matrix $I - \mathbf{q}_i\mathbf{q}_i^T$ formed in the least squares elimination algorithm at the ith call. Then we have:

$$\mathbf{q}_i = \frac{1}{\|T_{i-1}\cdots T_1\mathbf{a}_i\|} T_{i-1}\cdots T_1\mathbf{a}_i.$$

In a bit of remarkable luck, the matrix 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

15

There is another approach to solving the least squares problems

$$\text{minimize} \quad \|\mathbf{b} - A\mathbf{x}\|$$

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 A's columns, or the range of 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 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 A.

More formally, the QR factorization of a tall $m \times n$ matrix A (with $m \geq n$) is a pair of matrices Q and R such that:

· $A = QR$
· Q is square $m \times m$ and orthogonal
· 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]

15.1 REVIEW OF GRAM-SCHMIDT

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 Q such that

$$\begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{bmatrix} = QC$$

for some non-singular, square matrix C. The Gram-Schmidt process begins with the first vector $\mathbf{x}$ and sets the first column of Q to be $\mathbf{x}/\|\mathbf{x}\|$.

[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.

This review is meant to remind you of stuff you hopefully learned in previous linear algebra classes.

Then we *project-out* any component of **x** on the other vectors. The matrix $P(\mathbf{x}) = \mathbf{I} - \frac{\mathbf{x}\mathbf{x}^T}{\mathbf{x}^T\mathbf{x}}$ is a projector[2] to the space orthogonal to the vector **x**. That is, $\mathbf{x}^T P(\mathbf{x})\mathbf{y} = \mathbf{x}^T\mathbf{y} - \frac{\mathbf{x}^T\mathbf{x}}{\mathbf{x}^T\mathbf{x}}\mathbf{x}^T\mathbf{y} = 0$. Hence, we compute $\mathbf{y}_1 = P(\mathbf{x})\mathbf{y}$, $\mathbf{z}_1 = P(\mathbf{x})\mathbf{z}$. The next vector $\mathbf{q}_2 = \mathbf{y}_1/\|\mathbf{y}_1\|$. and we project $\mathbf{z}_2$ via $P(\mathbf{y}_1)$. This gives us three vectors:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{x}/\|\mathbf{x}\| & \mathbf{y}_1/\|\mathbf{y}_1\| & \mathbf{z}_2/\|\mathbf{z}_2\| \end{bmatrix}$$

where $\mathbf{y}_1 = P(\mathbf{x})\mathbf{y}$ and $\mathbf{z}_2 = P(\mathbf{y}_1)P(\mathbf{x})\mathbf{z}$. We can write this as a matrix equation as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{bmatrix} = \begin{bmatrix} \mathbf{x}/\|\mathbf{x}\| & \mathbf{y}_1/\|\mathbf{y}_1\| & \mathbf{z}_2/\|\mathbf{z}_2\| \end{bmatrix} \begin{bmatrix} \|\mathbf{x}\| & C_{1,2} & C_{1,3} \\ 0 & \|\mathbf{y}_1\| & C_{2,3} \\ 0 & 0 & \|\mathbf{z}_2\| \end{bmatrix}$$

where $C_{i,j}$ arises from the projection operations. Consider $C_{1,2}$, which we get from $\mathbf{y}_1 = P(\mathbf{x})\mathbf{y} = \mathbf{y} - \frac{\mathbf{x}^T\mathbf{y}}{\mathbf{x}^T\mathbf{x}}\mathbf{x}$, 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 **A** via successive orthogonalization, which can be written:

$$\mathbf{A} = \mathbf{Q}\mathbf{R}$$

for an $m \times n$ matrix **Q** and a square upper-triangular matrix $n \times n$ matrix **R**. This is often called a "thin" QR factorization because the matrix **Q** isn't square but is *tall* instead.

15.2   GENERALIZING TO QR

The idea with the full QR factorization is that we can extend a "thin" QR factorization to a square matrix **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 orthogonal as well. Of course, because these are orthogonal, we don't need to use them to write the matrix **A**, so the "tail" of **R** becomes zero.

15.3   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 $\|\mathbf{x}\| = \|\mathbf{Q}\mathbf{x}\| = \|\mathbf{Q}^T\mathbf{x}\|$ for any square orthogonal matrix **Q**.

[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, $\mathbf{P}^2 = \mathbf{P}$ for any projector matrix!

Hence, let $A = QR$ be any full QR factorization with a square matrix Q, then

$$\|\mathbf{b} - A\mathbf{x}\| = \|Q^T\mathbf{b} - Q^T A\mathbf{x}\| = \|\hat{\mathbf{b}} - R\mathbf{x}\| = \left\| \begin{bmatrix} \hat{\mathbf{b}}_1 \\ \hat{\mathbf{b}}_2 \end{bmatrix} - \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \mathbf{x} \right\|.$$

Here, we used $R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$ where R_1 is the first set of n rows of R. Because 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 $A\mathbf{x}$. Hence, the best we can do to minimize the expression is to set $\mathbf{x}$ so that $\hat{\mathbf{b}}_1 = R_1\mathbf{x}$.

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 b̂₁ = first n rows of Qb when Q is full,
   or b̂₁ = QᵀB when Q is m × n.
Solve R₁x = b̂₁.
Return x
```

15.4   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:

$$Q = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}.$$

Let's see how to pick Q for $\mathbf{v}$.

An obvious way is to try and compute θ in the above expression such that

$$Q(\theta)\mathbf{v} = \gamma\mathbf{e}_1$$

for some γ.

However, there is a better way to do this! Note that $Q(\theta)$ only has two unknowns, $c = \cos\theta$ and $s = \sin\theta$. To compute 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 $\mathbf{Q}(\theta)$ is called a Givens rotation.

15.5 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

15.6 GIVENS ROTATIONS IN JULIA

TODO

15.7 COMPUTING QR FOR A COLUMN

Consider computing a QR factorization for a $n \times 1$ vector $\mathbf{v}$ now. By the definition, we have:

$$\mathbf{Q}\mathbf{v} = \gamma\mathbf{e}_1.$$

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

15

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.

Program 1:
$$\begin{array}{ll} \underset{\mathbf{x}}{\text{maximize}} & \|A\mathbf{x}\|_2 \\ \text{subject to} & \|\mathbf{x}\|_2 = 1 \end{array}$$

Program 2:
$$\begin{array}{ll} \underset{\mathbf{x}}{\text{maximize}} & \mathbf{x}^T A \mathbf{x} \\ \text{subject to} & \|\mathbf{x}\|_2 = 1. \end{array}$$

For program 2, we also assume that 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.

15.8 THE EIGENVALUE DECOMPOSITION

Let A be a symmetric matrix.

15.9 THE SINGULAR VALUE DECOMPOSITION

We will following a similar deflation procedure.

15.10 SOLVING THE SUBPROBLEMS

15.11 INVARIANT SUBSPACES

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

16

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!

16.1 MEMORY REQUIREMENTS

To run the Richardson method or steepest descent method (lecture 8), we require:

1. a way to multiple 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]

[1] These can often be done in place as well.

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.)

16.2 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 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.

16.3 A WARM UP: THE FLOP COUNT OF MATRIX-MULTIPLY.

Consider the following algorithm for multiplying two matrices.

```
1  function matmul(A::Matrix, B::Matrix)
2    m,k = size(A)
3    k,n = size(B)
4    C = similar(A, m, n)
5    fill!(C, 0)
6    for j=1:n
7      for i=1:m
8        for r=1:k
9          C[i,j] += A[i,r]*B[r,j]
10       end
11     end
12   end
13   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.

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.

16.4 THE FLOP COUNT OF LU

Let's consider something more interesting, the pivoted LU decomposition of a matrix. Our code is

```
1   function myreduce_all_pivot(A::Matrix)
2     A = copy(A) # save a copy
3     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!

SENSITIVITY & CONDITIONING *17*

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 31, but the matrix is simple to state:

$$A = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & \cdots \\ 1/2 & 1/3 & 1/4 & 1/5 & \cdots \\ 1/3 & 1/4 & 1/5 & 1/6 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \\ & & & & & A_{ij} = \frac{1}{i+j-1} \end{bmatrix}$$

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.

```
1  function hilbert(n)
2    return [1/(i+j-1) for i=1:n, j=1:n]
3  end
4  n = 15
5  A = hilbert(n)
6  x = ones(n)
7  b = A*x
8  bt = b  + eps(1.0)*rand(StableRNG(1), n)
9  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
```

17.1 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.)

$$A_{ij}^{-1} = (-1)^{i+j} \binom{n+i-1}{n-j} \binom{n+j-1}{n-i} \binom{i+j-2}{i-1}^2$$

This gets large extremely quickly.[1]

[1] It seems like a good exercise in combinatorics to work out what the one-norm of this matrix is!

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.

17.2 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 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* $\mathbf{b}$ actually gives a large perturbation to $f(x)$. Hence, we *should* see large changes to $\mathbf{x}$ even for small changes in $\mathbf{b}$.

[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.

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 $\mathbf{f}$ be a length N vector of these function values $f_i = f(x_i)$. Then

$$\mathbf{b} \approx \frac{1}{N} \underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_N^{n-1} \end{bmatrix}}_{=M} \underbrace{\begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_N) \end{bmatrix}}_{\mathbf{f}} \quad \text{or more compactly} \quad \mathbf{b} = M\mathbf{f}.$$

The matrix 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 $\mathbf{b}$, then $M^{-1}\mathbf{b}$ gives the change to the function values that $\mathbf{b}$ represents.

17.3 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 $\mathbf{m}(\mathbf{b}) = A^{-1}\mathbf{x}$

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 17.1

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]

$$\kappa(\mathbf{x}; \mathbf{m}) = \frac{\|(\nabla\mathbf{m})(\mathbf{x})\|}{\|\mathbf{m}(\mathbf{x})\|} \|\mathbf{x}\|.$$

[3] Note that the Jacobian is $\lim \mathbf{d} \to 0 \|\mathbf{m}(\mathbf{x} + \mathbf{d}) - \mathbf{m}(\mathbf{x})\| \|\mathbf{d}\|$.

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 17.2 What is the condition number of the simple act of multiplying two numbers? The map $m(x, y) = xy$. ◆

17.4 THE CONDITION NUMBER OF A LINEAR SYSTEM

For the map $\mathbf{m}(\mathbf{b}) = A^{-1}\mathbf{b}$ we have:

$$\nabla\mathbf{m} = A^{-1} \quad \text{and} \quad A\mathbf{x} = \mathbf{b}$$

in which case we have

$$\kappa(\mathbf{b}; \mathbf{m}) = \|A^{-1}\| \frac{\|\mathbf{b}\|}{\|A^{-1}\mathbf{b}\|}$$

but in terms of the solution **x** we have:

$$\kappa(\mathbf{b}; \mathbf{m}) = \|A^{-1}\| \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}.$$

Note that for any vector **x** and an operator induced norm, we have:

$$\frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A\|$$

by the definition of an operator-induced norm. Thus

$$\kappa(\mathbf{b}; \mathbf{m}) \leq \|A\| \|A^{-1}\|.$$

The condition number of solving a linear system arises so often that we give it a special name.

> **Definition 17.3 (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
>
> $$\kappa(A) = \|A\| \|A^{-1}\|$$
>
> which gives the condition number for any operator induced matrix norm. The choice of norm is typically the 2-norm unless otherwise specified.

CONDITIONING OF LEAST SQUARES & THE PSEUDOINVERSE

18

Consider a least squares problem with a full rank A

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|A\mathbf{x} - \mathbf{b}\| \ .$$

There are two natural maps to consider:

$$\text{a map from } \mathbf{b} \text{ to } \mathbf{x} \quad \mathbf{x} = (A^T A)^{-1} A^T \mathbf{b}$$

$$\text{a map from } \mathbf{b} \text{ to } A\mathbf{x} \quad \mathbf{y} = A(A^T A)^{-1} A^T \mathbf{b}.$$

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]

[1] Trefethen and Bau, Lecture 18, Theorem 18.1
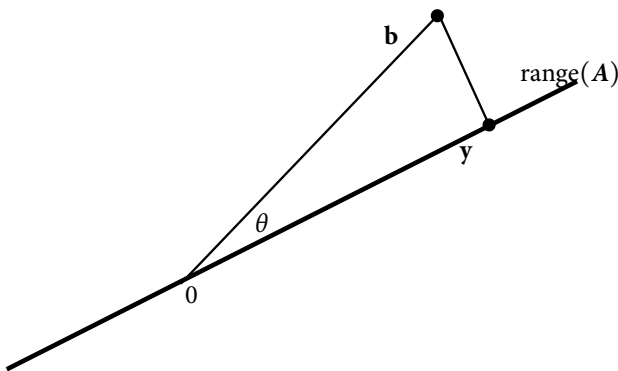
A least squares problem involves the interaction of $\mathbf{b}$ with range(A). The angle between the two is θ. If θ is zero, then $\mathbf{b}$ is in the range of A and the residual of the least squares problem is 0.

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 $\cos\theta = 0$.

18.1 THE PSEUDOINVERSE.

It's easiest to analyze this problem by establishing a single quantity for $(A^T A)^{-1} A^T$. This is the pseudoinverse.

$$A^+ = (A^T A)^{-1} A^T.$$

However, we will write this in terms of the SVD of A. Let $A = U\Sigma V^T$. Then

$$(A^T A)^{-1} = (V\Sigma^T \Sigma V^T)^{-1} = V(\Sigma^T \Sigma)^{-1} V^T$$

$$A^+ = (A^T A)^{-1} A^T = V(\Sigma^T \Sigma)^{-1} V^T V\Sigma^T U^T = V(\Sigma^T \Sigma)^{-1}\Sigma^T U^T.$$

The term $(\Sigma^T \Sigma)^{-1}\Sigma^T$ 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:

$$(\Sigma^T \Sigma)^{-1} \text{ is } n \times n, \text{ diagonal } = \begin{cases} \frac{1}{\sigma_i^2} & \text{on diagonal} \\ 0 & \text{otherwise.} \end{cases}$$

Hence, the entire matrix is:

$$(\Sigma^T \Sigma)^{-1}\Sigma^T = n \times m, \text{ diagonal } = \begin{cases} \frac{1}{\sigma_i} & \text{on diagonal} \\ 0 & \text{otherwise.} \end{cases}$$

This defines the pseudo-inverse of a diagonal matrix, which we often write:

$$\Sigma^+ = (\Sigma^T \Sigma)^{-1}\Sigma^T.$$

To recap: the pseudoinverse of A is

$$A^+ = V\Sigma^+ U^T.$$

18.2 THE CONDITION NUMBER OF LEAST SQUARES IN TERMS OF THE PSEUDOINVERSE.

Let $\mathbf{x} = A^+\mathbf{b}$. Then the condition number of the least squares vector $\mathbf{x}$ in terms of $\mathbf{b}$ is:

$$\kappa(\mathbf{b}) = \frac{\|A^+\| \|\mathbf{b}\|}{\|\mathbf{x}\|} = \|A^+\| \frac{\|\mathbf{b}\|}{\|\mathbf{y}\|} \frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \frac{\|A^+\|}{\cos\theta} \frac{\|\mathbf{y}\|}{\|\mathbf{x}\|}.$$

This second term $\frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}$ measures how large $\mathbf{y}$ is compared with $\mathbf{x}$. Clearly, this is less than $\|A\|$. So we have

$$\kappa(\mathbf{b}) \leq \|A^+\| \|A\| \frac{1}{\cos\theta}.$$

However, if we let $\frac{1}{\eta} = \frac{\|A\mathbf{x}\|}{\|A\|\|\mathbf{x}\|}$, then $\frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \|A\|\frac{1}{\eta}$. Consequently, we have

$$\kappa(\mathbf{b}) = \|A^+\|\|A\| \frac{1}{\eta \cos \theta}.$$

Because the term $\|A^+\|\|A\|$ arises just as frequently as $\|A^{-1}\|\|A\|$, we extend the definition of $\kappa(A)$ with a rectangular matrix A.

18.3 THE CONDITION NUMBER OF THE LEAST SQUARES PREDICTION.

Let $\mathbf{y} = AA^+\mathbf{b}$. Then

$$\mathbf{y} = U\Sigma V^T V\Sigma^+ U^T \mathbf{b} = U\Sigma\Sigma^+ U^T \mathbf{b}.$$

The matrix AA^+ is an orthogonal projection onto $\mathrm{range}(A)$. What this means for us is that $\|AA^+\| = 1$ for the operator induced 2-norm. Consequently

$$\kappa(\mathbf{b}) = \|AA^+\| \frac{\|\mathbf{b}\|}{\|\mathbf{y}\|} = \frac{1}{\cos \theta}.$$

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

<div style="text-align: right;">

19

</div>

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 = √x

for i=1 to 60
   x = x²
```

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:

$$|\tilde{y} - y|/|y| \text{ is something like } \kappa(y)|\delta|.$$

Let's put this slightly more formally now.

Definition 19.1

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]

² Throughout this note, u is the machine precision.

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.

BACKWARDS STABILITY OF LU DECOMPOSITION

20

Warning We will adopt slightly non-standard notation. The quantity Δ is a matrix, as are any symbols prefixed with δ, such as δL or δU. Upper tildes, like $\tilde{L}$ 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

$$(A + \Delta)\tilde{\mathbf{x}} = \mathbf{b}.$$

In words, we compute the solution to a perturbed system where the perturbation is the matrix Δ. If so, then, if $A\mathbf{x} = \mathbf{b}$, we have:[1]

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\rho\kappa(A)}{1 - \rho\kappa(A)}$$

where $\rho = \|\Delta\|/\|A\|$.

Thus, if we can show that $\|\Delta\|$ is small, we will have a nice bound on the error of the solution.

[1] It's a good exercise to stitch this bound together from $\tilde{\mathbf{x}} + A^{-1}\Delta\tilde{\mathbf{x}} = \mathbf{x}$.

FRAMEWORK

This is a complicated operation. When we solve $A\mathbf{x} = \mathbf{b}$ with Gaussian elimination, we have three steps:

1. Factoring $A = LU$. We will show that we find: $\tilde{L}\tilde{U} = A + E$. We will assume that partial pivoting is used, although, we will assume the permutation is known up front.

2. Next we have to solve $L\mathbf{y} = \mathbf{b}$, or on the computer,

$$(\tilde{L} + \delta L) \underbrace{(\mathbf{y} + \delta\mathbf{y})}_{=\tilde{\mathbf{y}}} = \mathbf{b}.$$

In other words, we again solve a perturbed system exactly.

3. Finally, we have to solve $U\mathbf{x} = \mathbf{y}$. But on the computer, this is now:

$$(\tilde{U} + \delta U)(\mathbf{x} + \delta\mathbf{x}) = \tilde{\mathbf{y}} = \mathbf{y} + \delta\mathbf{y}.$$

Together, these results show that

$$\mathbf{b} = (\tilde{L} + \delta L)(\tilde{U} + \delta U)(\mathbf{x} + \delta\mathbf{x}).$$

But, we have:

$$(\tilde{L} + \delta L)(\tilde{U} + \delta U) = \tilde{L}\tilde{U} + \delta L\tilde{U} + \tilde{L}\delta U + \delta L\delta U = A + E + \delta L\tilde{U} + \tilde{L}\delta U + \delta L\delta U.$$

Thus,

$$(A + \Delta)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b}$$

with $\Delta = E + \delta L\tilde{U} + \tilde{L}\delta U + \delta L\delta U$.

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: $\tilde{L}\tilde{U} = A + E$ for some 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 A. We'll represent this as a sequence of changes to the matrix

$$A = A^{(1)} \xrightarrow{\text{zero 1st column}} A^{(2)} \xrightarrow{\text{zero 2st column}} A^{(3)} \rightarrow \cdots \rightarrow A^{(n-1)}.$$

Thus, $A^{(k)}$ 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 $B^{(k)}$ 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 $i \leq k$, or 2) it's zero in the lower-triangle with $j < k$.

[2] This is a straightforward observation if you look at LU in exact arithmetic:

$$A^{(k)} = \begin{bmatrix} U & C \\ 0 & D \end{bmatrix},$$

where U is $(k-1)$-by-$(k-1)$.

So we'll divide our analysis into two cases that correspond to these two outcomes. First, suppose we are in the upper-triangle, so $j \geq i$. Then,

$$B_{ij}^{(2)} = B_{ij}^{(1)} - \tilde{L}_{i,1} B_{1,j}^{(1)} + \mu_{ij}^{(2)}$$
$$B_{ij}^{(3)} = B_{ij}^{(2)} - \tilde{L}_{i,2} B_{2,j}^{(2)} + \mu_{ij}^{(3)}$$
$$\dots$$
$$B_{ij}^{(i)} = B_{ij}^{(i-1)} - \tilde{L}_{i,i-1} B_{i,j}^{(i-1)} + \mu_{ij}^{(i)}.$$

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 $A + E$, 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

$$B_{ij}^{(2)} = B_{ij}^{(1)} - \tilde{L}_{i,1} B_{1,j}^{(1)} + \mu_{ij}^{(2)}$$
$$B_{ij}^{(3)} = B_{ij}^{(2)} - \tilde{L}_{i,2} B_{2,j}^{(2)} + \mu_{ij}^{(3)}$$
$$\dots$$
$$B_{ij}^{(j)} = B_{ij}^{(j-1)} - \tilde{L}_{i,j-1} B_{j-1,j}^{(j-1)} + \mu_{ij}^{(j)}.$$

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:

$$B_{ij}^{(1)} = 0 + \sum_{k=1}^{j} \tilde{L}_{i,k} B_{k,j}^{(k)} + E_{ij}$$

[3] I think there might be an index mistake in here, be wary.

where $E_{ij} = \sum_{k=2}^{j+1} \mu_{ij}^{(k)}$.

Thus, what we compute on the computer is:

$$\tilde{L}\tilde{U} = \begin{bmatrix} 1 & & & & \\ \tilde{L}_{2,1} & 1 & & & \\ \tilde{L}_{3,1} & \tilde{L}_{3,2} & 1 & & \\ \vdots & \vdots & & \ddots & \\ \tilde{L}_{n,1} & \tilde{L}_{n,2} & \cdots & \cdots & 1 \end{bmatrix} \begin{bmatrix} B_{1,1}^{(1)} & B_{1,2}^{(1)} & B_{1,3}^{(1)} & \cdots & B_{1,n}^{(1)} \\ & B_{2,2}^{(2)} & B_{2,3}^{(2)} & \cdots & B_{2,n}^{(2)} \\ & & B_{3,3}^{(3)} & & \vdots \\ & & & \ddots & \vdots \\ & & & & B_{n,n}^{(n)} \end{bmatrix}.$$

Using our equations that we derived, we can show:

$$\tilde{L}\tilde{U} = \boldsymbol{B}^{(1)} + \boldsymbol{E} = \boldsymbol{A} + \boldsymbol{E}.$$

Bounding the errors

Now, we need to bound each element in $\boldsymbol{E}$. We have:

$$\tilde{L}_{i,k} = \text{fl}\left(B_{ik}^{(k)}/B_{kk}^{(k)}\right) = (B_{ik}^{(k)}/B_{kk}^{(k)})(1 + \eta_{ik})$$

and

$$\text{fl}\left(\tilde{L}_{i,k}B_{k,j}^{(k)}\right) = (\tilde{L}_{i,k}B_{k,j}^{(k)})(1 + \theta_{ij}^{(k)}),$$

so that:

$$B_{ij}^{(k+1)} = \text{fl}\left(B_{ij}^{(k)} - (\tilde{L}_{i,k}B_{k,j}^{(k)})(1 + \theta_{ij}^{(k)})\right) = \left(B_{ij}^{(k)} - (\tilde{L}_{i,k}B_{k,j}^{(k)})(1 + \theta_{ij}^{(k)})\right)(1 + \phi_{ij}^{(k)}).$$

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

$$|A_{ij}^{(2)}| \leq |A_{ij}^{(1)}| + |A_{kj}^{(1)}| \leq 2a$$

$$|A_{ij}^{(3)}| \leq |A_{ij}^{(2)}| + |A_{kj}^{(2)}| \leq 4a$$

$$\vdots$$

$$|A_{ij}^{(n)}| \leq 2^{n-1}a.$$

We'll return to this bound in a second. It's pretty absurd.

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]

[4] Recall that $\frac{1}{1-u} = 1 + u + u^2 + \ldots$.

$$|\mu_{ij}^{(k+1)}| \leq Ga\frac{u}{1-u}+Gau = Ga(2u-u^2)(1+u+u^2+\ldots) \approx 2uGa+O(u^2).$$

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:

$$|E| \leq 2uGa\begin{bmatrix} 0 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & 1 & 1 & \cdots & \cdots & 1 \\ 1 & 2 & 2 & \cdots & \cdots & 2 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & n-2 & n-2 \\ 1 & 2 & 3 & \cdots & n-1 & n-1 \end{bmatrix} + O(u^2).$$

Thus, $A + E = \tilde{L}\tilde{U}$.

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

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

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 $\hat{A}$ be an arbitrary matrix and let A be a slight random perturbation of $\hat{A}$. We prove that it is unlikely that A has a large condition number. Using this result, we prove that it is unlikely that A has large growth factor under Gaussian elimination without pivoting. By combining these results, we show that the smoothed precision necessary to solve $A\mathbf{x} = \mathbf{b}$, for any $\mathbf{b}$, using Gaussian elimination without pivoting is logarithmic. Moreover, when $\hat{A}$ 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]

$$T\mathbf{v} = \mathbf{h}$$

where 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:

$$v_1 = h_1/T_{11}$$
$$\vdots$$
$$v_k = \frac{h_k - T_{k,1}v_1 - T_{k,2}v_2 - \cdots - T_{k,k-1}v_{k-1}}{T_{kk}}.$$

Let $\tilde{v}_k$ be the computed value in floating point. In floating point, this gives us:

$$\tilde{v}_k = \left(\frac{\left(h_k - \sum_{i=1}^{k-1} T_{k,i}\tilde{v}_i(1 + \omega_{k,i}) \right)(1 + \alpha_k)}{T_{k,k}} \right)(1 + \tau_k)$$

$$= \frac{h_k - \sum_{i=1}^{k-1} T_{k,i}\tilde{v}_i(1 + \omega_{k,i})}{T_{k,k}/(1 + \alpha_k)(1 + \tau_k)}.$$

With some more *coffee-shop manipulations*, we arrive at:

$$\sum_{i=1}^{k} T_{k,i}\tilde{v}_i(1 + \lambda_{k,i}) = h_k$$

[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.

or

$$T\tilde{\mathbf{v}} + \begin{bmatrix} \lambda_{1,1} T_{1,1} & & \\ \lambda_{2,1} T_{2,1} & \lambda_{2,1} T_{2,1} & \\ \vdots & \vdots & \ddots \end{bmatrix} \tilde{\mathbf{v}} = \mathbf{h}.$$

Equivalently,[6]

$$(T + \delta T)\mathbf{v} = \mathbf{h}$$

where

$$|\delta T| \le u \begin{bmatrix} |T_{1,1}| & & & & \\ |T_{2,1}| & 2|T_{2,2}| & & & \\ 2|T_{3,1}| & |T_{3,2}| & 3|T_{3,3}| & & \\ \vdots & \vdots & & \ddots & \\ (n-1)|T_{n,1}| & \cdots & \cdots & |T_{n,n-1}| & n|T_{n,n}| \end{bmatrix} + O(u^2)$$

Thus, for solving a triangular system, we have errors:

$$(T + \delta T)\mathbf{v} = \mathbf{h} \text{ where } |\delta T_{ij}| \le nut + O(u^2) \text{ and } |T_{ij}| \le t.$$

Overall errors

In our framework, the solution $\tilde{\mathbf{x}} = \mathbf{x} + \delta x$ satisfies:

$$(A + \Delta)\mathbf{x} = \mathbf{b}$$

where $\Delta = E + \delta L\tilde{U} + \tilde{L}\delta U + \delta L\delta U$. Let's look at the maximum errors in each of these terms:

$$\max_{ij} |\delta \tilde{L}_{ij}| \le nu + O(u^2)$$

$$\max_{ij} |\delta \tilde{U}_{ij}| \le nuGa + O(u^2).$$

This tells us:

$$\max_{ij} |Delta_{ij}| \le \max_{ij} |E_{ij}| + \max_{ij} |(\delta L\tilde{U})_{ij}| + \max_{ij} |(\tilde{L}\delta U)_{ij}| + \max_{ij} |(\delta L\delta U)_{ij}|$$

$$\le 2uGan + n^2 Gau + n^2 Gau + O(u^2).$$

Thus,

$$\|\Delta\|_\infty \le 2n^2(n+1)uGa.$$

This merits a theorem.

Theorem 20.1

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

$$\rho \ge 2n^2(n+1)G$$

because $\|A\|_\infty \geq a$. Going back to the beginning, we now have:

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\rho\kappa(A)}{1 - \rho\kappa(A)} = \frac{2n^2(n+1)G\kappa(A)}{1 - 2n^2(n+1)G\kappa(A)}$$

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

$$\mathbf{x}^{(k)} - \mathbf{x} \approx M^k(\mathbf{x}^{(x)} - \mathbf{x})$$

for some matrix $M^k \to 0$. We saw these for linear systems and eigenvalues.

The one exception was steepest descent (and Nesterov accelerated steepest descent), where we saw a polynomial

THE MATRIX POWERS SUBSPACE, AKA THE KRYLOV SUBSPACE

21

21.1 MOTIVATION

Recall the first method we saw to solve a linear system of equations:

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

where we conceptually multiplied by the inverse

$$(A)^{-1} \approx I + (I - A) + (I - A)^2 + \ldots$$

to get the algorithm:

$$\mathbf{x}^{(k)} = \sum_{j=0}^{k} (I - A)^j \mathbf{b}.$$

Let's call this the Neumann-series algorithm for linear systems.

This converged as long as $\rho(I - A) < 1$. 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]

First, note that:

$$\mathbf{x}^{(k)} = \begin{bmatrix} \mathbf{b} & (I - A)\mathbf{b} & \ldots & (I - A)^k \mathbf{b} \end{bmatrix} \mathbf{e}.$$

That is, we can represent the kth iteration as a (simple!) linear combination of the basis vectors

$$(\mathbf{b}, (I - A)\mathbf{b}, \ldots, (I - A)^k \mathbf{b}).$$

This means that, for some vector $\mathbf{c}$, we can write:

$$\mathbf{x}^{(k)} = \begin{bmatrix} \mathbf{b} & A\mathbf{b} & A^2\mathbf{b} & \ldots A^k\mathbf{b} \end{bmatrix} \mathbf{c}.$$

Let's work this out, which will give us a lead on our next perspective.

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.

[1] The ideas behind these methods were independently described around the same time by both Krylov and Lanczos.

Lemma 21.1

Consider the kth iteration from a Neumann-series based approach, where $\mathbf{x}^{(k)} = \sum_{j=0}^{k} (\mathbf{I} - \mathbf{A})^j \mathbf{b}$. Then we can write $\mathbf{x}^{(k)} = \sum_{j=0}^{k} c_j \mathbf{A}^j \mathbf{b}$ for some coefficients $c_0, \ldots, c_k$.

PROOF The proof follows from the binomial expansion:

$$(\mathbf{I} - \mathbf{A})^k \mathbf{b} = \sum_{j=0}^{k} \binom{k}{j} (-\mathbf{A})^j \mathbf{b}.$$

But a more useful realization is as follows:

$$(\mathbf{I} - \mathbf{A})^k \mathbf{b} = \text{polynomial}(\mathbf{A})\mathbf{b}.$$

In which case, the theorem is just giving a change of basis between polynomials in powers of $(1 - x)$ and x.[2]   ∎

[2] See the discussion section 21.1.1.

Just to be clear, let's state the other result as well.

Corollary 21.2

Consider the kth iteration from a Neumann-series based approch, where $\mathbf{x}^{(k)} = \sum_{j=0}^{k} (\mathbf{I} - \mathbf{A})^j \mathbf{b}$, then $\mathbf{x}^{(k)} = p(\mathbf{A})\mathbf{b}$ for some polynomial $p(x) = \sum_{j=0}^{k} c_j x^j$.

21.1.1 *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

[3] Much more on polynomials will be discussed in a future chapter on Orthogonal Polynomials, lecture 24. Read more there now if you wish.

$$p(x) : \mathbb{R} \to \mathbb{R} \text{ where } p(x) = c_0 + c_1 x + c_2 x^2 + \cdots c_k x^k.$$

The degree of the polynomial is the highest power. So $p(x) = 5 + 2x + 3x^2$ is a degree 3 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, $g_j(x) = x^j$.

132   V · SUBSPACE METHODS

21.1.2   *Subspaces and Polynomials*

Consider $\mathbf{x}^k$ from the Neumann series

Subspaces

The subspace view is that

$$\mathbf{x}^{(k)} = \begin{bmatrix} \mathbf{b} & A\mathbf{b} & A^2\mathbf{b} & \cdots & A^{k-1}\mathbf{b} \end{bmatrix} \mathbf{c}$$

to indicate that $\mathbf{x}^{(k)}$ is a specific linear combination of the basis vectors from the matrix powers subspace

$$\begin{bmatrix} \mathbf{b} & A\mathbf{b} & A^2\mathbf{b} & \cdots & A^{k-1}\mathbf{b} \end{bmatrix}.$$

Polynomials

The polynomial view is that

$$\mathbf{x}^{(k)} = \mathbf{b} + (I - A)\mathbf{b}+$$
$$(I - A)^2\mathbf{b} + \cdots+$$
$$(I - A)^{k-1}\mathbf{b}$$
$$= \operatorname{poly}(A)\mathbf{b}$$

where $\operatorname{poly}(A) \approx A^{-1}$.

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 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 A^{-1}.

21.2   THE MATRIX POWERS SUBSPACE

The matrix powers subspace is the set of vectors

$$\mathbb{K}_k(A, \mathbf{b}) = \operatorname{span}(\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \dots A^k\mathbf{b}).$$

This is typically called the *Krylov subspace*. Hence, the Neumann method just uses a specific element of $\mathbb{K}_k(A, \mathbf{b})$ 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 21.3**
> Let A be full rank. Suppose that $A^k\mathbf{b} \in \mathbb{K}_{k-1}(A, \mathbf{b})$. Then the solution of $A\mathbf{x} = \mathbf{b}$ is contained within $\mathbb{K}_{k-1}(A, \mathbf{b})$ as well.

PROOF   Let X be any basis for $\mathbb{K}_{k-1}(A, \mathbf{b})$. Then we have that $A^k\mathbf{b} = X\mathbf{y}$ for some vector $\mathbf{y}_k$. Consequently, we also have that $A^{k+1}\mathbf{b} = X\mathbf{y}_{k+1}$. Hence, for any set of powers beyond k, they exist in the basis X. The

simplest way to prove this is to appeal to a slightly fancy result involving the Cayley-Hamilton theorem. [4] Note that, by the Cayley-Hamilton theorem, there is a degree n polynomial $p(A)$ such that $p(A) = A^{-1}$. (See section below for an explicit derivation.) Hence, we by the assumptions of the theorem, we have that $p(A)\mathbf{b}$ is in the subspace too. ∎

The reason this theorem is nice is because is says we never need to be concerned about singular X. If X is singular, then we have solved our linear system!

21.2.1 *A polynomial expression for the matrix inverse*

21.2.2 *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

$$X = \begin{bmatrix} \mathbf{b} & A\mathbf{b} & A^2\mathbf{b} & \dots A^k\mathbf{b} \end{bmatrix}$$

as that is how the subspace is defined. The problem with this basis, however, is that X becomes very ill-conditioned as k gets large.

Let's see this for a *diagonal* linear system! Suppose that

$$A_n = \begin{bmatrix} 1 & & & & \\ & 1/2 & & & \\ & & 1/4 & & \\ & & & 1/8 & \\ & & & & \ddots \end{bmatrix}$$

where A_n is n-by-n.

Then suppose that $\mathbf{b} = \mathbf{e}$, so we get the vector of all ones. We have that

$$X = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1/2 & 1/4 & \dots & 1/(2^k) \\ 1 & 1/4 & 1/16 & \dots & 1/(4^k) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1/(2^{n-1}) & 1/(2^{n-1})^2 & \dots & 1/(2^{n-1})^k \end{bmatrix}.$$

For k large (think $k \geq 52$ for floating point), then

$$A^k\mathbf{b} = \begin{bmatrix} 1 \\ \leq \varepsilon \\ \leq \varepsilon^2 \\ \leq \varepsilon^4 \\ \vdots \end{bmatrix} \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix}$$

Further note that for subsequent k, $A^{k+1}\mathbf{b} \approx A^k\mathbf{b}$ and so the matrix X is nearly singular because it has columns that are the same.

[4] The Cayley-Hamilton Theorem states that there is a degree n polynomial $q(x)$ such that $q(A) = 0$. (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 $q(A)A^{-1} = 0$ too, but $q(A) = c_n A^n + \dots c_0 I$ so $q(A)A^{-1} = c_n A^{n-1} + c_0 A^{-1} = 0$, which we can solve for A^{-1} to get a degree $n - 1$ polynomial for the inverse.

A good way to characterize this is via the ill-conditioning of the matrix. We plot the condition number of X as a function of k.

This example is hardly unique.

Up next: A better basis for the subspace

What we'd ideally like is an orthogonal basis for $\mathbb{K}_k(A, \mathbf{b})$. 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.
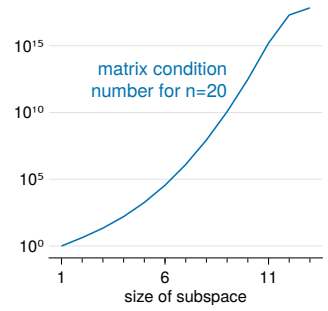
FIGURE 21.1 – The matrix X quickly becomes ill-conditioned with only a few entries from the matrix power basis.

ORTHOGONAL BASES FOR THE MATRIX POWERS SUBSPACE, AKA THE ARNOLDI AND LANCZOS PROCESSES

22

The Krylov or matrix power subspace is

$$K_k = \text{span}(\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \ldots, A^k\mathbf{b}).$$

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, $X_k = \begin{bmatrix} \mathbf{b} & A\mathbf{b} & A^2\mathbf{b} & \ldots & A^k\mathbf{b} \end{bmatrix}$ is ill-conditioned. Note that there is already structure in the matrix X. We have $X_{k+1} = \begin{bmatrix} \mathbf{b} & AX_k \end{bmatrix}$. We also have[1]

$$X_{k+1} = AX_k \begin{bmatrix} 0 \\ I_k \end{bmatrix}.$$

An orthogonal matrix always has condition number 1. We now seek to understand how to build a well-conditioned basis for this subspace.

22.1 ORTHOGONALIZING AT EACH STEP

The first idea we consider is building the orthogonalization iteratively at each step.[2]

Let $X_k = \begin{bmatrix} \mathbf{b} & A\mathbf{b} & A^2\mathbf{b} & \ldots & A^k\mathbf{b} \end{bmatrix}$. Then suppose we have $X_k = Q_k R_k$ as the thin-QR factorization of this matrix where R_k is square. We can straightforwardly update from

$$X_k, Q_k R_k \quad \text{to} \quad X_{k+1}, Q_{k+1}, R_{k+1}.$$

Let $X_{k+1} = \begin{bmatrix} Q_k R_k & \mathbf{y} \end{bmatrix}$ be a block partitioning of this matrix. Then $Q_k^T X_{k+1} = \begin{bmatrix} R_k & Q_k^T \mathbf{y} \end{bmatrix}$. Consider the difference pieces of the last column of this matrix:

$$Q_k^T \mathbf{y} = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{bmatrix}.$$

If we build a Householder matrix V_{k+1} where

$$V_{k+1}\mathbf{z} = \begin{bmatrix} \mathbf{z}_1 \\ \pm \|\mathbf{z}_2\| \mathbf{e}_1 \end{bmatrix}$$

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 $A\mathbf{x} = \mathbf{b}$ simplifies when expressed in the Arnoldi or Lanczos basis for the Krylov subspace.

[1] The matrix $\begin{bmatrix} 0 \\ I_k \end{bmatrix}$ has size $k+1$ by k and is a "upward shift"

$$\begin{bmatrix} 0 \\ I_k \end{bmatrix} = \begin{bmatrix} \mathbf{e}_2 & \mathbf{e}_3 & \cdots & \mathbf{e}_{k+1} \end{bmatrix}$$

or

$$\begin{bmatrix} 0 \\ I_k \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ 0 & \cdots & \ddots & \cdots \\ 0 & \cdots & 0 & 1 \end{bmatrix}$$

[2] This ideas has been asked by students.

then we have

$$X_{k+1} = \underbrace{Q_k V_{k+1}}_{=Q_{k+1}} \underbrace{\begin{bmatrix} R_k & \begin{bmatrix} \mathbf{z}_1 \\ \pm\|\mathbf{z}_2\|\mathbf{e}_1 \end{bmatrix} \end{bmatrix}}_{=R_{k+1}}.$$

Note that the matrix R is not involved in building the orthogonal basis Q we are interested in. So we need not worry about maintaining R.

This gives us an easy algorithm to build Q_k as a product of orthogonal reflectors matrices.

```
1  function successive_orthogonalization(A,b,k)
2    # determine the element type of reflector
3    FloatType = unify_type(eltype(A),eltype(b))
4    Qs = OrthogonalReflector{FloatType}[]
5    push!(Qs, reflector(FloatType, b, index=1))
6    y = copy(b)
7    for i in 1:k
8      y = A*y
9      z = reflector_transpose_product(Qs, y) # compute Qk'*y
10     push!(Qs, reflector(FloatType, y, index=i+1))
11   end
12   return Qs
13 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 Q.

However, there is an even bigger simplification possible. Recall also that $AX_k = X_{k+1} \begin{bmatrix} 0 \\ I_k \end{bmatrix}$. Using our QR decomposition:

$$AX_k = AQ_k R_k = Q_{k+1} R_{k+1} \begin{bmatrix} 0 \\ I_k \end{bmatrix}.$$

The matrix R_k is non-singular because otherwise, the Krylov subspace would be complete. This means that

$$AQ_k = Q_{k+1} R_{k+1} \begin{bmatrix} 0 \\ I_k \end{bmatrix} R_k^{-1}.$$

Finally,

the matrix $R_{k+1} \begin{bmatrix} 0 \\ I_k \end{bmatrix} R_k^{-1}$ is an upper Hessenberg matrix,

$$H_{k+1} R_{k+1} \begin{bmatrix} 0 \\ I_k \end{bmatrix} R_k^{-1} = \text{TODO Add picture of upper Hessenberg.}$$

Using this idea gives us the Arnoldi process where we attempt to directly compute the matrix H_{k+1}.

22.2 THE ARNOLDI PROCESS

The idea with the Arnoldi process is that we can build an orthogonal basis for

the Krylov subspace K_k directly from $A, \mathbf{b}$ without QR.

That is, we can identify the matrix Q_k that is an orthogonal directly from the expression:

$$AQ_k = Q_{k+1}H_{k+1}, \quad \mathbf{q}_1 = \mathbf{b}/\|\mathbf{b}\|, \quad Q_k^T Q_k = I, \quad Q = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_k \end{bmatrix}.$$

This, perhaps surprisingly, can be done rather straightforwardly and constructively just by enumerating constraints and conditions.

Consider the case for $k = 1$

$$A\mathbf{q}_1 = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 \end{bmatrix} \begin{bmatrix} H_{1,1} \\ H_{2,1} \end{bmatrix}.$$

We are given $\mathbf{q}_1$ by the construction of starting with $\mathbf{b}$. Hence, we can compute $\mathbf{y} = A\mathbf{q}_1$. 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

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 $\mathbf{q}_1, \mathbf{q}_2$.

$$A\begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 \end{bmatrix} \begin{bmatrix} H_{1,1} & H_{1,2} \\ H_{2,1} & H_{2,2} \\ 0 & H_{3,2} \end{bmatrix}.$$

We begin by computing $\mathbf{y} = A\mathbf{q}_2$. 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

```
1  function arnoldi(A,b,k)
2    Q = zeros(n,k+1)
3    H = zeros(k+1,k)
4    Q[:,1] .= b
5    normalize!(@view(Q[:,1]))
6    for j=1:k
7      y = A*Q[:,j]
8      for i=1:j
9        H[i,j] = Q[:,i]'*y # σ_i
10       y -= H[i,j]*Q[:,i]
11     end
12     H[j+1,j] = norm(y)    # y = z at this point
13     Q[:,j+1] = y / H[j+1,j]
14   end
15   return Q, H
16 end
```

A cautious check

TODO - We want to verify that nothing went wrong! And directly show that the Arnoldi process spans the Krylov subspace.

22.3 THE LANCZOS PROCESS

22.4 ALTERNATIVE POLYNOMIAL BASES

Exercise

Fix A and $\mathbf{b}$. What is the condition number of problem of minimizing the residual in the kth Krylov subspace?

Show that the product $R_{k+1} \begin{bmatrix} 0 \\ I_k \end{bmatrix} R_k^{-1}$ is upper triangular.

CONJUGATE GRADIENT

<div style="text-align: right">

23

</div>

The conjugate gradients (CG) method is one of the most celebrated algorithms for solving $A\mathbf{x} = \mathbf{b}$ when 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 24.)

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 A be $n \times n$, symmetric positive definite.

23.1   CONJUGATE GRADIENTS VIA THE LANC-ZOS PROCESS

Because A is symmetric, we can run the Lanczos process to iteratively compute a tridiagonal matrix T that approximates the matrix A. *For linear systems, we also begin the Lanczos process with the vector* $\mathbf{b}/\|\mathbf{b}\|$. After k steps, we have:

$$\underbrace{A}_{n \times n} \underbrace{V}_{n \times k} = \underbrace{V_{k+1}}_{n \times k+1} \underbrace{T_{k+1}}_{k+1 \times k}$$

where $V = \begin{bmatrix} \mathbf{v}_1, \ldots, \mathbf{v}_k \end{bmatrix}$ holds the first k vectors in the Lanczos process and V_{k+1} holds the first k vectors *and the $k + 1$st vector*. The matrix

$$T_{k+1} = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \ddots & & \\ & \ddots & \ddots & \beta_k & \\ & & \beta_k & \alpha_k & \\ & & & \beta_{k+1} & \end{bmatrix} = \begin{bmatrix} \bar{T} \\ \beta_{k+1}\mathbf{e}_k^T \end{bmatrix}$$

where $\bar{T}$ is the $k \times k$ tridiagonal matrix from the first k rows. When it's important, we'll write:

$$A V_k = V_{k+1} T_{k+1}$$

to denote the full sequence of matrices. Likewise, the Lanczos process gives rise to a sequence of tridiagonal matrices:

$$\bar{T}_1, \bar{T}_2, \ldots, \bar{T}_k, \bar{T}_{k+1}, \ldots$$

However, we'll often drop the index k when it applies to *any index*. For instance, in exact arithmetic, $V^T A V = \bar{T}$.

NOTE   The vector $\mathbf{v}_1 = \mathbf{b}/\|\mathbf{b}\|$, and also $V\mathbf{e}_1 = \mathbf{v}_1$ for all k.

QUIZ   Show that $\bar{T}$ is positive definite if A is positive definite.

23.1.1 *Lanczos and Linear Systems*

Let $\mathbf{y} = V\mathbf{z}$ 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

$$\mathbf{y} \in \mathcal{K}_k(A, \mathbf{b}).$$

For any vector $\mathbf{y} = V\mathbf{z}$:

$$\|\mathbf{b} - A\mathbf{y}\| = \|\mathbf{b} - AV\mathbf{z}\| = \|V_{k+1}\left(\|\mathbf{b}\|\mathbf{e}_1 - T_{k+1}\mathbf{z}\right)\| = \|\|\mathbf{b}\|\mathbf{e}_1 - T_{k+1}\mathbf{z}\|.$$

Thus, we want to pick $\mathbf{z}$ such that $\|\mathbf{b}\|\mathbf{e}_1 - T_{k+1}\mathbf{z}$ is small at each step.

In the conjugate gradients method, we choose $\mathbf{z}$ such that

$$\bar{T}\mathbf{z} = \|\mathbf{b}\|\mathbf{e}_1$$

so that

$$\|\|\mathbf{b}\|\mathbf{e}_1 - T_{k+1}\mathbf{z}\| = |\beta_{k+1}z_k|.$$

In contrast, in the MINRES method, we choose $\mathbf{z}$ to minimize $\|\|\mathbf{b}\|\mathbf{e}_1 - T_{k+1}\mathbf{z}\|$ at each step; and in the SYMMLQ method, we choose $\mathbf{y} = V_{k+1}\mathbf{z}$ and $\mathbf{z}$ is the minimum norm solution of $T_{k+1}^T\mathbf{z}$. We won't spend too much time studying these methods.

23.1.2 *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  T̄_k z_k = ‖b‖ e_1
    Compute x_k = V_k z_k
    If |β_{k+1} z_k| < tol, stop.
```

The essence of the method is that we replace solving $A\mathbf{x} = \mathbf{b}$ with solving $\bar{T}_k\mathbf{z} = \|\mathbf{b}\|\mathbf{e}_1$. Put another way, the idea is that the matrix $\bar{T}_k$ "approximates" A.

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 $\mathbf{V}_k$ and $\mathbf{T}_k$ 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 $\bar{T}$ is $O(k)$ work because it's a tridiagonal system. However, the problem is that computing $\mathbf{x}_k = \mathbf{V}_k \mathbf{z}$ 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.

23.1.3 *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 $\mathbf{V}_k$. 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 A is symmetric positive definite. Based on the quiz above, this means that $\bar{T}$ is also symmetric, positive definite. So it has a Cholesky factorization

$$\bar{T}_k = F_k F_k^T.$$

On the last homework, we worked out that:

$$F = \begin{bmatrix} \eta_1 & & & \\ \mu_2 & \eta_2 & & \\ & \ddots & \ddots & \\ & & \mu_k & \eta_k \end{bmatrix}$$

Moreover, we can compute μ_{k+1} and η_{k+1} from

$$\bar{T}_{k+1} = \begin{bmatrix} \bar{T}_k & \beta_{k+1}\mathbf{e}_k \\ \beta_{k+1}\mathbf{e}_k^T & \alpha_{k+1} \end{bmatrix} = \begin{bmatrix} F_k & \\ \mu_{k+1}\mathbf{e}_k^T & \eta_{k+1} \end{bmatrix} \begin{bmatrix} F_k^T & \mu_{k+1}\mathbf{e}_k \\ & \eta_{k+1} \end{bmatrix}.$$

By equating terms, we find that

$$\beta_{k+1}\mathbf{e}_k = \mu_{k+1}F_k\mathbf{e}_k = \mu_{k+1}\eta_k\mathbf{e}_k$$

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

$$\mathbf{x}_k = \|\mathbf{b}\| V_k \bar{T}^{-1} \mathbf{e}_1.$$

Let's substitute the Cholesky fatorization in here:

$$\mathbf{x}_k = \|\mathbf{b}\| V_k F^{-T} F^{-1} \mathbf{e}_1.$$

The "leap" is that we need to look at:

$$\mathbf{x}_k = \|\mathbf{b}\| \cdot \underbrace{C_k}_{=V_k F^{-T}} \cdot \underbrace{\mathbf{p}_k}_{=F^{-1}\mathbf{e}_1} .$$

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:

$$F_{k+1}\mathbf{p}_{k+1} = \begin{bmatrix} F & \\ \mu_{k+1}\mathbf{e}_k^T & \eta_{k+1} \end{bmatrix} \begin{bmatrix} \mathbf{p}_k \\ p_{k+1} \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1 \\ 0 \end{bmatrix}.$$

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 C_{k+1} from C_k. We'll write:

$$C_{k+1} F_{k+1}^T = V_{k+1}$$

or

$$\begin{bmatrix} C_k & \mathbf{c}_{k+1} \end{bmatrix} \begin{bmatrix} F_k^T & \mu_{k+1}\mathbf{e}_k \\ & \eta_{k+1} \end{bmatrix} = \begin{bmatrix} V_k & \mathbf{v}_{k+1} \end{bmatrix}.$$

Hence,

$$\mu_{k+1} C_k \mathbf{e}_k + \mathbf{c}_{k+1} \eta_{k+1} = \mu_{k+1}\mathbf{c}_k + \eta_{k+1}\mathbf{c}_{k+1} = \mathbf{v}_{k+1}.$$

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:

$$\mathbf{x}_{k+1} = C_{k+1}\mathbf{p}_{k+1} = \underbrace{C_k\mathbf{p}_k}_{=\mathbf{x}_k} + \mathbf{c}_{k+1} p_{k+1}.$$

And we have found an efficient expression for $\mathbf{x}_k$ in the CG method.

All together now, we have:

$\beta_1 = \|\mathbf{b}\|$
$\mathbf{v}_0 = 0 , \quad \mathbf{x}_0 = 0$
$\mathbf{v}_1 = \mathbf{b}/\beta_1$
```
for i=1, 2, ...
```
$\qquad \mathbf{w} = A\mathbf{v}_i - \beta_i \mathbf{v}_{i-1}$
$\qquad \alpha_i = \mathbf{v}_i^T \mathbf{w}$
$\qquad \mathbf{w} \leftarrow \mathbf{w} - \alpha_i \mathbf{v}_i$
$\qquad \beta_{i+1} = \|\mathbf{w}\|$
$\qquad \mathbf{v}_{i+1} = \mathbf{w}/\beta_{i+1}$
```
    if i = 1
```
$\qquad\qquad \mu_i = 0 , \quad \eta_i = \sqrt{\alpha_i} , \quad p_i = \beta_1/\eta_i , \quad \mathbf{c}_i = \mathbf{v}_1/\eta_1 .$
```
    else
```
$\qquad\qquad \mu_i = \beta_i/\eta_{i-1} , \quad \eta_i = \sqrt{\alpha_i - \mu_i^2} , \quad p_i = -\mu_i p_{i-1}/\eta_i , \quad \mathbf{c}_i = (\mathbf{v}_i - \mu_i \mathbf{c}_{i-1})/\eta_i$
$\qquad \mathbf{x}_i = \mathbf{x}_{i-1} + p_i \mathbf{c}_i$

In this iteration, you only need to keep the vector $\mathbf{v}_i$ and $\mathbf{c}_i$ to complete the iteration.

23.2 CONJUGATE GRADIENTS VIA OPTIMIZATION

In the second derivation of the CG method, we study the problem:

$$\min \phi(x) \text{ where } \phi(x) = \frac{1}{2}\mathbf{x}^T A \mathbf{x} - \mathbf{x}^T \mathbf{b}$$

This section is incomplete

where A is $n \times n$, symmetric positive definite. In this case, the solution is $\mathbf{x} = A^{-1}\mathbf{b}$, which we can derive by setting the gradient of $\phi(x)$ to zero,[1] that is,

[1] this condition suffices because the problem is strongly convex

$$\partial \phi/\partial \mathbf{x} = A\mathbf{x} - \mathbf{b} = 0.$$

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, $\mathbf{g}_k = A\mathbf{x}_k - \mathbf{b}$. 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]

[2] check the sign on this value

$$\alpha_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_k^T A \mathbf{g}_k}.$$

This simple method:

```
x^(1) = 0
for k=1, ...
    g^(k) = Ax^(k) − b
    if ‖g^(k)‖ < tol
        stop and return x_k
    α^(k) = g^(k)^T g^(k)/(g^(k)^T Ag^(k))
    x^(k+1) = x^(k) + α_k 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:

$$\alpha_k = \mathbf{p}^{(k)^T} \mathbf{r}^{(k)}/(\mathbf{p}^{(k)^T} A \mathbf{p}^{(k)})$$

as long as $\mathbf{p}^{(k)} \mathbf{r}^{(k)} \neq 0$ where $\mathbf{r}^{(k)}$ is the kth residual $\mathbf{b} - A\mathbf{x}^{(k)}$.

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 $\mathbf{p}^{(i)^T} A \mathbf{p}^{(j)} = 0$ if $i \neq j$. In this case, $\mathbf{x}^{(k)} = \sum_{i=1}^{n} \alpha_i \mathbf{p}^{(i)}$ or $\mathbf{x}^{(k)} \in \text{span}\{\mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(k)}\}$.

23.3   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

ORTHOGONAL POLYNOMIALS & MATRIX COMPUTATIONS

24

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."

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.

24.1 WHAT ARE ORTHOGONAL POLYNOMIALS?

For the purposes of this lecture, a polynomial is a univariate function:[1]

$$p(t) = \sum_{i=0}^{n} p_i t^i.$$

[1] Multivariate generalizations exist, but we won't need them.

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:

$$\mathbf{v}^T \mathbf{u} = \sum_{i=1}^{n} v_i u_i = 0.$$

EXAMPLE 24.1 The two polynomials p and q given above are orthogonal.

$$\int_{-1}^{1} p(t)q(t)\,dt = \int_{-1}^{1} \frac{1}{2}(t^2 - 1)\frac{1}{2}(5t^3 - 3t)\,dt$$

$$= \frac{1}{4}\int_{-1}^{1} 5t^5 - 5t^3 - 3t^3 + 3t\,dt$$

$$= \frac{1}{4}\int_{-1}^{1} 5t^5 - 8t^3 + 3t\,dt.$$

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 $\quad \int_{a}^{b} p(t)q(t)\,dt$

a weighted integral $\quad \int_{a}^{b} p(t)q(t)\,dw(t)$

a discrete weight $\quad \int_{a}^{b} p(t)q(t)\,dw(t) = \sum_{i=1}^{n} p(\lambda_i)q(\lambda_i)w_i$

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 $\int_{-1}^{1} dt$:

$$p_{-1}(t) = 0$$
$$p_0(t) = 1$$
$$p_1(t) = t$$
$$p_2(t) = \tfrac{1}{2}(3t^2 - 1)$$
$$p_3(t) = \tfrac{1}{2}(5t^3 - t)$$
$$p_4(t) = \tfrac{1}{8}(35t^4 - 30t^2 + 3)$$

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.

24.2   THE THREE TERM RECURRENCE &
TRIDIAGONAL MATRICES

The following fact, which we will not prove, is profound:

Theorem 24.2
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 24.3   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 $T(t)$ and solving:

$$T(t)\mathbf{p} = \mathbf{e}_1.$$

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:

$$J = \begin{bmatrix} \alpha_1 & 1 & & \\ \beta_1 & \alpha_2 & 1 & \\ 0 & \ddots & \ddots & \ddots & 0 \end{bmatrix}$$

but you'll need to read more about that in Gautschi's book.

24.3 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:

$$p(A) = \sum_{i=0}^{n} p_i A^i.$$

24.4 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 $\mathbb{K}_k(A, \mathbf{b}) = \text{span}\{\mathbf{b}, A\mathbf{b}, \ldots, A^{k-1}\mathbf{b}\}$. This means that there is some polynomial such that:[3]

$$\mathbf{x}_k = s_{k-1}(A)\mathbf{b}.$$

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:

$$\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k = \mathbf{b} - As_{k-1}(A)\mathbf{b} = \big(I - As_{k-1}(A)\big)\mathbf{b}. \quad \text{Thus} \quad \mathbf{r}_k = p_k(A)\mathbf{b}$$

where the polynomial $p_k(t) = 1 - ts_{k-1}(t)$ has degree k.

We call these two polynomials:

$$s_k(t) \quad \text{the solution polynomial}$$
$$p_k(t) \quad \text{the residual polynomial.}$$

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: $p_k(0)\mathbf{b} = \mathbf{b}$.

24.5 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:

$$\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k = p_k(A)\mathbf{b}.$$

We want:

$$\mathbf{r}_k^T \mathbf{r}_j = \mathbf{b}^T p_k(A)^T p_j(A)\mathbf{b} = 0.$$

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!

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:

$$(i) p_{k+1}(0) = 1$$
$$(ii) \mathbf{r}_{k+1} = p_{k+1}(A)\mathbf{b} = \mu_k \big(\mathbf{r}_k - \gamma_k A \mathbf{r}_k \big) + \eta_k \mathbf{r}_{k-1}$$
$$(iii) (\mathbf{r}_{k+1})^T \mathbf{r}_j = 0, j < k.$$

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):

$$\mathbf{r}_{k+1} = p_{k+1}(A)\mathbf{b} = \mu_k \big(\mathbf{r}_k - \gamma_k A \mathbf{r}_k \big) + (1 - \mu_k) \mathbf{r}_{k-1}.$$

If we apply constraint (iii) with $j = k$, we have:

$$0 = (\mathbf{r}_k)^T \mathbf{r}_{k+1} = \mu_k \big((\mathbf{r}_k)^T \mathbf{r}_k - \gamma_k (\mathbf{r}_k)^T A \mathbf{r}_k \big) + (1 - \mu_k)(\mathbf{r}_k)^T \mathbf{r}_{k-1}$$

or

$$\gamma_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_k^T A \mathbf{r}_k}.$$

Finally, using constraint (iii) with $j = k - 1$, we can solve for[5]

$$\mu_k = \frac{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1} + \gamma_k \mathbf{r}_{k-1}^T A \mathbf{r}_k}.$$

This lets us compute $\mathbf{r}_{k+1}$

Getting the solution polynomial

Thus far, we have the residual polynomial $\mathbf{r}_k = p_k(A)\mathbf{b}$. We do, however, need to recreate the solution too! To do so, we note that

$$s_k(t) = \frac{1 - p_{k+1}(t)}{t} = \frac{1 - \mu_k(p_k(t) - \gamma_k t p_k(t)) - (1 - \mu_k) p_{k-1}(t)}{t}.$$

[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 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.

[5] Check for an error here... and this simplifies more, see Saad

Now we add and subtract μ_k/t in order to rewrite this as:

$$s_k(t) = \mu_k \left[\frac{1 - p_k(t)}{t} - \gamma_k p_k(t) \right] - (1 - \mu_k)\frac{1 - p_{k-1}(t)}{t}$$
$$= \mu_k(s_{k-1}(t) - \gamma_k p_k(t)) - (1 - \mu_k)s_{k-2}(t).$$

Hence, we have:

$$\mathbf{x}_{k+1} = \mu_k(\mathbf{x}_k - \gamma_k \mathbf{r}_k) - (1 - \mu_k)\mathbf{x}_{k-1}.$$

24.6   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.

25 EFFICIENT GMRES

Recall the prototype-GMRES method.

```
Given A,b where we can only multiply by A.
for i=1 to maxiter
  Update the Arnoldi factorization Q_k, H_{k+1}.
  Solve for z_k by minimizing ‖H_{k+1}z_k − ‖b‖e_1‖,
    i.e. z_k = argmin‖H_{k+1}z_k − ‖b‖e_1‖
  Let x_k = Q_k z_k.
  Check ‖Ax_k − 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,b where we can only multiply by A.
for i=1 to maxiter
  Update the Arnoldi factorization Q_k, H_{k+1}.
  Compute ‖r_k‖ where
      r_k = Ax_k − b
      x_k = Q_k z_k
      z_k = argmin‖H_{k+1}z_k − ‖b‖e_1‖
    and stop once ‖r_k‖ is sufficiently small, i.e.
    Update ‖r_k‖ → ‖r_{k+1}‖ and stop if it's small enough.
Explicitly compute z_k = argmin‖H_{k+1}z_k − ‖b‖e_1‖
  and return x_k = Q_k z_k only at the end of the iteration
```

25.0.1 *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

$$\|\mathbf{r}_k\| = \|U_k R_k \mathbf{y}_k - \beta_0 \mathbf{e}_1\| = \|R_k \mathbf{y}_k - \beta_0 U_k^T \mathbf{e}_1\|.$$

Showing this after a few steps gives us the idea more clearly:

$$\|\mathbf{r}_k\| = \left\| \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{R_k} \mathbf{y}_4 - \beta_0 \underbrace{\begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_5 \end{bmatrix}}_{U_k^T \mathbf{e}_1} \right\|. = \beta_0 \gamma_5.$$

(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 $\|\mathbf{r}_k\|$.

25.0.2 *Taking it deeper*

We need to note a two things here to continue our optimization:

1. We only need Givens rotations to get $H_k \to R_k$.

2. We only need *one* rotation to update $R_k \to R_{k+1}$. (Woah!)

Let's review step 1 and see how that will help us with step 2.

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{J_1} \begin{bmatrix} * & * & * & * \\ \circ & * & * & * \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \quad \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{J_2} \begin{bmatrix} \times & \times & \times & \times \\ 0 & * & * & * \\ 0 & \circ & * & * \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{J_3} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & * & * \\ 0 & 0 & \circ & * \\ 0 & 0 & 0 & \times \end{bmatrix} \quad \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{J_4} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & \circ \end{bmatrix}$$

Finish and improve this figure, the point is we use $J_1, \ldots, J_4$ to do the Givens rotations. These give us $U_4^T = J_4 J_3 \ldots J_1$.

Now, suppose we have U_4 and R_4, how do we get U_5, R_5?

$$H_5 = \begin{bmatrix} H_4 & \mathbf{h} \\ 0 & h_{6,5} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} & \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} & h_{6,5} \end{bmatrix}$$

If we rotate by U_4^T, we get:

$$\begin{bmatrix} U_4^T & 0 \\ 0 & 1 \end{bmatrix} H_5 = \begin{bmatrix} U_4^T & 0 \\ 0 & 1 \end{bmatrix} H_5 = \begin{bmatrix} R_4 & U_4^T \mathbf{z}_4 \\ 0 & h_{6,5} \end{bmatrix}$$

So at this point, we just have the one Givens rotation: J_5 that we need to do to fixup the element $h_{6,5}$ and so $U_5^T = J_5 J_4 \cdots J_1$, which is just one update.

25.0.3 *Seeking gamma.*

Note that the elements of gamma are just the first column of U_k^T. Let $\mathbf{g}_k = U_k^T \mathbf{e}_1 = \begin{bmatrix} \gamma_1 & \gamma_2 & \ldots & \gamma_k \end{bmatrix}^T$. Then by our previous relationship:

$$\mathbf{g}_{k+1} = U_{k+1}^T \mathbf{e}_1 = J_{k+1} U_k^T \mathbf{e}_1 = J_{k+1} \mathbf{g}_k.$$

But this is weird, because J is a $k + 1 \times k + 1$ matrix and $\mathbf{g}_k$ is a length k vector. So what we really mean is

$$J_{k+1} \begin{bmatrix} \mathbf{g}_k \\ 0 \end{bmatrix}$$

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.

25.0.4 *The whole algorithm*

```
g = β₀e₁
for k=1 to ...
   Update  Qₖ,Hₖ
   Let  η_{k+1} = H_{k+1,k} .
   Let  zₖ = H_{1:k,k} .
   Apply  J₁...J_{k-1}  to  zₖ,  and update  H
   Create  Jₖ  to elminate  η_{k+1} .
   Determine  gₖ  from  Jₖg_{k-1}  growing by zeros as needed.
   If  gₖ(end) is small enough, then stop iterating.
At this point, H  has the factor  R, and (if we do keep g accurate), then
g is the right hand side, so we can just solve  Rₖyₖ = gₖ  and then
output  x = Qₖy.
```

25.1 GMRES VS. FOM

See notes.

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

$$\min \|A\mathbf{x} - \mathbf{b}\| \qquad \text{least squares}$$

$$A\mathbf{x} = \mathbf{b} \qquad \text{linear systems}$$

$$A\mathbf{x} = \lambda\mathbf{x} \qquad \text{eigenvalues}$$

$$A\mathbf{x} = \sigma\mathbf{y} \qquad \text{singular values}$$

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!

MULTIPLE RIGHT HAND SIDES

26

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.

26.1 MULTIPLE RIGHT HAND SIDES ALL KNOWN AHEAD OF TIME

The problem here is that we need to solve $A\mathbf{x} = \mathbf{b}$ for many vectors $\mathbf{b}_1, \ldots, \mathbf{b}_k$. In the simplest case, we will know $\mathbf{b}_1, \ldots, \mathbf{b}_k$ ahead of time. In this case, we really have the matrix problem

$$AX = B \qquad B = \begin{bmatrix} \mathbf{b}_1 & \cdots & \mathbf{b}_k \end{bmatrix}$$

where 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 A. [1] Then we would set $B = I$, and there are $k = n$ vectors $\mathbf{b}$ all known.

Another, more realistic scenario, arises in block Gaussian elimination. Suppose we are solving

$$A\mathbf{b} = \mathbf{b} \text{ where we have the partition } \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}.$$

Then note that if A_1 is non-singular, then $\mathbf{x}_1$ must satisfy $A_1\mathbf{x}_1 + A_2\mathbf{x}_1 = \mathbf{b}_1$ or

$$\mathbf{x}_1 = A_1^{-1}\mathbf{b}_1 + A_1^{-1}A_2\mathbf{x}_2.$$

The matrix $A_1^{-1}A_2$ 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 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 A and then apply this to all the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_k$ 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 $O(n^2 k + n^3)$, which is $O(n^3)$ if $k \leq O(n)$ and *more interseting* if $k \geq O(n)$.

As an example where the latter scenario arises, consider the partiton above where A_1 is 16×16 and n is 1024.

26.2 MULTIPLE RIGHT HAND SIDES DE-TERMINED SEQUENTIALLY

The second setting for multiple right hand sides is that we have

$$A\mathbf{x}_1 = \mathbf{b}_1$$

which determines $\mathbf{b}_2$, so $\mathbf{b}_2$ is unknown until we have solved $\mathbf{x}_1$. Then we must solve

$$A\mathbf{x}_2 = \mathbf{b}_2$$

$$A\mathbf{x}_3 = \mathbf{b}_3 = \text{function of } \mathbf{x}_2.$$

and so on...

The key is that the matrix 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

$$\mathbf{x}^{(0)} = \text{ arbitrary} \qquad \mathbf{x}^{(k+1)} = \rho_k A \mathbf{x}^{(k)} \quad \rho_k = \frac{1}{\|A\mathbf{x}^{(k)}\|}.$$

If the largest magnitude eigenvalue of A is unique, then $\mathbf{x}^{(k)}$ will converge towards the associated eigenvector. The inverse power method simply runs this iteration on A^{-1} instead (assuming A is non-singular). For instance, if A is symmetric positive definite, then the inverse power method will converge to the smallest eigenvalue of A. Here, we have exactly this type of setting where $\mathbf{b}^{(k+1)} = \rho_k \mathbf{x}^{(k)}$.[2]

[2] Again, note that the way to do this isn't to compute A^{-1} and use that instead of A! That's another good way to fail the class.

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 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 A that may not exist in the inverse 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,

```
1  F = lufact(A) # produces a factorization object F
2  F \ y # solves Ax = y using the LU factorizations without recomputing it.
```

So we could implement the inverse power method as follows

```
1  function invpower(A::Matrix)
2    x = normalize!(randn(size(A,1)))
3    F = lufact(A)
4    for iter = 1:maxiter
5      x = normalize!(F\x)
6    end
7    return x
8  end
```

PRECONDITIONING

<div style="text-align: right; font-size: 3em;">**27**</div>

Preconditioning is the process of taking a given linear system:

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

and turning it into a new linear system (with B non-singular):

$$B\mathbf{y} = \mathbf{c}$$

such that it's "easy" to find $\mathbf{x}$ from $\mathbf{y}$ and

$$\text{an iterative method for } B\mathbf{y} = \mathbf{c} \text{ is } \left\{ \begin{array}{c} \text{faster} \\ \text{more accurate} \\ \text{better behaved} \\ \text{convergent} \\ \text{easier,} \dots \end{array} \right\}.$$

THE STANDARD PRECONDITIONER. The standard goal with preconditioning is to make an iterative method for $A\mathbf{x} = \mathbf{b}$ go faster. Typically this is done by taking a non-singular matrix M and looking at the linear system: [1]

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

The standard idea is that $MA \approx I$, and we'll see how to make this idea precise shortly. *Also*, we need a *fast* way to *create M*, and to *multiply M* by a vector. While this seems like an easy task, *many* preconditioners involve solving a system, hence, $M = P^{-1}$ for some matrix P (which could also be called a preconditioner!). Thus, just multiplying by M can be expensive itself.

Quiz Why do we need M to be non-singular?

[1] So in this case $B = MA$, $\mathbf{x} = \mathbf{y}$ and $\mathbf{c} = M\mathbf{b}$.

QUESTION 1 (THE FUNDAMENTAL QUESTION IN PRECONDITIONING)
Thus, we arise at the fundamental question. Given $A\mathbf{x} = \mathbf{b}$, how do I pick M or 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 A. Recently, there has been some work on how to do this for symmetric, diagonally dominant linear systems;[2]

[2] This is the celebrated Spielman and Teng nearly-linear time solver for SDD systems. The current runtime is $O(\mathrm{nnz}\sqrt{\log n})$ in theory, which means that it's faster to solve $A\mathbf{x} = \mathbf{b}$ 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 MA-
TRIX. Suppose that our problem $A\mathbf{x} = \mathbf{b}$ arises from a physics-based
application or a complex engineered system. The problem that we want to
solve gives rise to some matrix A and some right hand side $\mathbf{b}$. While we
could *study* the matrix A and attempt to use a matrix-based preconditioner
on A, it is often a better strategy to attempt to decompose your problem
as:

A = approximation with analytical solution given a right hand-side+correction.

In which case, we really have:

$$A = \underbrace{S}_{\text{simple}} + \underbrace{C}_{\text{correction}}$$

and $M = S^{-1}$ is a good preconditioner because

$$S^{-1}A = I + S^{-1}C.$$

27.1 A MORE FORMAL TREATMENT.

The following theorem justifies why S^{-1} would be a good precondi-
tioner.

> **Theorem 27.1 (Golub and van Loan, 3rd edition, 10.2.5)**
> [3] If $A = I + B$ is an n-by-n symmetric postive definite matrix
> and rank$(B) = r$, then Krylov methods converge in at most
> $r + 1$ iterations.

[3] In the third edition, they split this into 11.3.1 and 11.3.2.

PROOF This is a standard proof strategy. We show that in at most $r + 1$
iterations, the Krylov space $\mathbb{K}_{r+1}(A, \mathbf{b})$ contains the solution $\mathbf{x}$. To do so,
note that:

$$\begin{aligned}
\mathbb{K}_k(A, \mathbf{b}) &= \text{span}(\mathbf{b}, A\mathbf{b}, \ldots, A^{k-1}\mathbf{b}) \\
&= \text{span}(\mathbf{b}, (I + B)\mathbf{b}, (I + B)^2 \ldots, (I + B)^{k-1}\mathbf{b}) \\
&= \text{span}(\mathbf{b}, B\mathbf{b}, B^2\mathbf{b}, \ldots, B^{k-1}\mathbf{b}).
\end{aligned}$$

Because B has rank r, we know that 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 optimal-
ity properties, any Krylov method will terminate in $r + 1$ steps in exact
arithmetic. ∎

[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 27.2 (Trefethen 38.5)**
> Let the CG iteration be applied to a symmetric positive definite linear system $A\mathbf{x} = \mathbf{b}$, where A has 2-norm condition number κ. Then there is a norm $\|\mathbf{z}\|_*$ where
> $$\left\|\mathbf{x} - \mathbf{x}^{(k)}\right\|_* \leq 2\left\|\mathbf{x} - \mathbf{x}^{(0)}\right\|_* \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k.$$

This gives rise to a *linear convergence theorem* that depends on the condition number of a matrix:

$$\left\|\mathbf{x} - \mathbf{x}^{(k)}\right\|_* = O(\rho^k)$$

where ρ depends on κ.

Quiz What is $\kappa(I)$?

Suppose $\kappa(A)$ is big (like one hundred million), then what happens? We get $\rho \approx 1$ (like 0.99999999).

Suppose $\kappa(A)$ is nearly 1 (like 16), then what happens? We get $\rho \approx 0$ (like 3/5).

So given any linear system, if we take $M = A^{-1}$, we will converge in one step. But, computing $M^{-1}\mathbf{x}$ is just as expensive as our original problem. So we want something cheaper.

27.2 DESIGNING A PRECONDITIONER

The above theorems motivate three different types of preconditioners:
1. Find a matrix P where P^{-1} is a fast operator and $P^{-1}A \approx I$, i.e. $\kappa(MA) \ll \kappa(A)$.
2. Find a matrix P where P^{-1} is a fast operator and $P^{-1}A = I +$ low-rank.

3. Find a matrix P where P^{-1} is a fast operator and $P^{-1}A$ = has few eigenvalues.

In all cases we need 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 $\dim(\mathbb{K}_k(A, \mathbf{b})) \leq 2$ when A is diagonalizable with two distinct eigenvalues.)

Some subtleties

Suppose we want to use conjugate gradient. Then we need A to be symmetric positive and definite. Suppose we have a matrix MA where M is fast operator and easy to find. Can we always use CG? No, because

$$MA \neq (MA)^T$$

in general.

27.3 TYPES OF PRECONDITIONERS

Thus, we consider four types of preconditioners:

Left	solve	$\underbrace{MA}_{B}\,\mathbf{x} = \underbrace{M\mathbf{b}}_{c}$
Right	solve	$\underbrace{AM}_{B}\,\underbrace{(M^{-1}\mathbf{x})}_{y} = \mathbf{b}$
Left &Right	solve	$\underbrace{M_1AM_2}_{B}\,\underbrace{(M_2^{-1}\mathbf{x})}_{y} = \underbrace{M_1\mathbf{b}}_{c}$
Symmetric	solve	$\underbrace{MAM^T}_{B}\,\underbrace{(M^{-T}\mathbf{x})}_{y} = \underbrace{M\mathbf{b}}_{c}$

For the CG case above, we want to use a symmetric preconditioner to preserve symmetry. Often, these are written with C:

$$\underbrace{C^{-1}AC^{-T}}_{B}\,\mathbf{y} = C^{-1}\mathbf{b} \qquad \mathbf{x} = C^{-1}\mathbf{y}.$$

With the hope that B has a small condition number, or clustered eigenvalues, ...

27.3.1 *Ensuring positive definiteness*

We also need $C^{-1}AC^{-T}$ to be positive definite when A is. We can insure this by taking CC^T as the Cholesky factorization of any positive definite matrix T.

27.3.2 *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.

27.4   EXAMPLES OF PRECONDITIONERS

27.4.1 *Diagonals*

The simplest case of preconditioning is to use the diagonal entries. Let $A = D + N$ (be a splitting into the diagonal and off-diagonal terms), then:

$$M = D^{-1}$$

is a preconditioner that makes

$$MA = I + D^{-1}N.$$

Quiz Is it always easy to use a diagonal precondition on a matrix?
Quiz How could you do symmetric diagonal preconditioning?

27.4.2 *Polynomials*

Recall the expansion of A^{-1} as it's Neumann series:[5]

$$(I - A)^{-1} = I + A + A^2 + A^3 + \dots.$$

Then we can use a finite truncation as the preconditioner to $A\mathbf{x} = \mathbf{b}$:

$$M \approx A^{-1} = I + (I - A) + (I - A)^2 + (I - A)^3.$$

27.4.3 *Incomplete factorizations*

Incomplete Cholesky and Incomplete LU are both factorizations:

$$A = CC^T - R \qquad A = LU^T - R$$

that are Cholesky-like and LU-like, but that have a new residual term. We call them incomplete if R has a zero-entry whenever 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.

[5] This is a matrix based on the geometric series: $1 + t + t^2 + \dots = \frac{1}{1-t}$

27.4.4 *Sparse approximate inverses*

Suppose we want the best tridiagonal preconditioner for a matrix A. To find this, we could consider the best approximation of the inverse:

$$\text{minimize} \quad \|I - AM\|$$
$$\text{subject to} \quad M \text{ is tridiagonal.}$$

The sparsity structure should be given, so the more general problem is, given sparsity structure matrix S:

$$\text{minimize} \quad \|I - AM\|$$
$$\text{subject to} \quad M \text{ has the same non-zeros as } S.$$

Consider the tridiagonal case. We can compute M a column at a time:

$$\text{Let } M\mathbf{e}_i = \mathbf{m}_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \alpha \\ \beta \\ \gamma \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{then} \quad \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \text{ solve } \text{minimize} \quad \left\| \mathbf{e}_i - \begin{bmatrix} A_{i-1} & A_i & A_{i+1} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \right\| .$$

27.4.5 *Multi-grid*

Recall how we thought about approximating the *problem* as a type of preconditioning. Suppose that $A\mathbf{x} = \mathbf{b}$ arises from a n-by-n discretization of Poisson's equation. This gives us an $n^2 \times n^2$ linear system: $A\mathbf{x} = \mathbf{b}$. 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 $A\mathbf{x} = \mathbf{b}$? 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 $A\mathbf{x} = \mathbf{b}$ 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

28

28.1 INVARIANT SUBSPACES

28.2 DEGENERATE EIGENVALUES

EIGENVALUE ALGORITHMS

<div style="text-align: right">**29**</div>

In these notes, we will frequently assume that 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.

29.1 USEFUL PROPERTIES OF EIGENVALUES.

Lemma 29.1
Let A be a symmetric matrix. If $p(A)$ is a polynomial of A, then the eigenvectors of A and $p(A)$ are the same, but the eigenvalues change.

PROOF Note that $p(A) = c_0 I + c_1 A + c_2 A^2 + \cdots + c_k A^k$ for some coefficients $c_0, \ldots, c_k$. Then $A = VDV^T$ is the eigenvalue decomposition and $A^j = VD^j V^T$. Hence $p(A) = Vp(D)V^T$. ∎

(The same proof holds for non-symmetric matrices with the Jordan canonical decomposition instead.)

Lemma 29.2 (Gerschgorin disks, Golub and van Loan Theorem 7.2.1)
Let A be any matrix. Suppose that $A = D + F$ where D is diagonal and F has zero diagonal. (So D are the diagonal entries of A and F are all the other entries. Then the eigenvalues of A are contained within the set:

$$\lambda(A) \subseteq \bigcup_{i=1}^{n} G_i$$

where G_i is the ith Gerschgorin disk:

$$G_i = \left\{ z \in \mathbb{C} : |z - D_{i,i}| \leq \sum_j |F_{i,j}| \right.$$

PROOF Let $\lambda \in \lambda(A)$ be any eigenvalue that is not equal to $D_{i,i}$ for any i. Then $A\mathbf{x} = \lambda\mathbf{x}$ yields:

$$(D + F - \lambda I)\mathbf{x} = 0 \Leftrightarrow \mathbf{x} = (D - \lambda I)^{-1} F\mathbf{x}.$$

From the latter fact, we have:

$$1 \le \left\| (\boldsymbol{D} - \lambda \boldsymbol{I})^{-1} \boldsymbol{F} \right\|$$

for any sub-multiplicative norm. If this is the ∞-norm, then

$$\left\| (\boldsymbol{D} - \lambda \boldsymbol{I})^{-1} \boldsymbol{F} \right\|_\infty = \sum_j \frac{|F_{k,j}|}{|D_{k,k} - \lambda|}$$

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 29.3 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 A of 8. ◆

29.2 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 τ
Iterate
  y = Ax
  λ = xᵀy
  x = y/ norm(y)
Until norm(Ax − λx) ≤ τ (your tolerance)
```

If the largest magnitude eigenvalue associated with A is unique, then this iteration converges to it.

— TODO – Insert picture of the eigenvalue convergence.

29.2.1 *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 29.4 Consider the matrix $A = \begin{bmatrix} 0 & \mathbf{e}^T \\ \mathbf{e} & 0 \end{bmatrix}$ 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 $|\lambda| \le |\rho(A)|$ because the spectral radius is the most extreme point. The following theorem gives an upper bound.

> **Theorem 29.5 (Trevisan 9.6, Lecture Notes on Graph Partitioning, Expanders and Spectral Methods, 2006**
> Let 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
>
> $$\mathbf{x}^{(k)^T} M \mathbf{x}^{(k)} / \mathbf{x}^{(k)^T} \mathbf{x}^{(k)} \ge \rho(A)(1 - \varepsilon) \frac{1}{1 + 4n(1 - \varepsilon)^{2k}}.$$
>
> with probability at least 3/16.

This can be then translated into a useful upper-bound on $\rho(A)$ and you can use the tightest value of ε to make the result you want.

29.2.2 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 $\theta \ge \rho(A)$ and run the power method on $\theta I - A$. 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.

29.2.3 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 A^{-1} itself, which corresponds to the changing one line of the iteration:

$$\mathbf{y} = A\mathbf{x} \quad \rightarrow \quad A\mathbf{y} = \mathbf{x}$$

so that we solve a linear system at each step. With this change, the power method will converge to the *smallest magnitude* eigenvalue of A instead. (That is, the one closest to 0.)

— TODO – Insert picture of the eigenvalue transformation $1/\lambda$

29.2.4 Targeting a specific eigenvalue.

If we wish to find an eigenvalue close to a value α, then we can use the iteration:

$$(A - \alpha I)\mathbf{y} = \mathbf{x}.$$

The eigenvalues of A that are near α will be close to zero in the matrix $(A - \alpha I)$, and so when we use inverse iteration on $(A - \alpha I)$ then we will find those as the solutions.

— TODO – Insert picture of the eigenvalue transformation $1/(\lambda - \alpha)$

29.2.5   *Using a folded spectrum to target an eigenvalue.*

If A is large and sparse, then we may not have a good way to solve linear systems with 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 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 $\theta^2 \leq A^2$, 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 $(A - \alpha I)^2$ to make eigenvalues nearby α close to zero. This approach is called the *folded spectrum* method.

29.2.6   *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 A. Then create a householder matrix H with $\mathbf{x}$ as associated vector such that $H\mathbf{x} = \mathbf{e}_1$. The matrix HAH^T has the following structure:

$$\begin{bmatrix} \lambda & 0 \\ 0 & B \end{bmatrix}$$

This is actually what we did way back when we created the SVD!

EXAMPLE 29.6   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])
```

29.3   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 τ
Iterate
  Y = AX
  Λ = diagonal elements of XᵀAX arranged as a diagonal matrix
  [X,R] = qr(Y)
Until norm(AX − XΛ) ≤ τ (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.

29.3.1   *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 τ
Set  X = I
Iterate
  Y = AX
  Λ = diagonal elements of XᵀAX arranged as a diagonal matrix
  [X,R] = qr(Y)
Until norm(AX − XΛ) ≤ τ (your tolerance)
```

EXAMPLE 29.7   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)]
```

29.3.2   *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 τ
X = I
Iterate
  Q,R = qr(A)
```

```
A = RQ.
Λ = diagonal elements of A arranged as a diagonal matrix
X = XQ
Until norm(AX − XΛ) ≤ τ (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 $A_{\text{next}} = Q^T A Q$. Essentially, we are multiplying by an orthogonal matrix on the left and the right.

EXAMPLE 29.8 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)]
```

29.3.3 *QR and Subspace Iteration are equivalent*

Here, we'll consider the following two methods and show that they are equivalent.

Subspace iteration   QR iteration

```
X₀ = I                                 A₁ = A
for i = 1, ...                         for i = 1, ...
  Xᵢ, Rᵢ = qr(AXᵢ₋₁)                    Qᵢ, Rᵢ = qr(Aᵢ)
                                        Aᵢ₊₁ = RᵢQᵢ
```

Lemma 29.9
$$X_i = Q_1 Q_2 \cdots Q_i$$

PROOF We'll prove this by induction. The key insight is that R_i is actually the same matrix. (Technical note, we need to assume that the signs of the diagonal elements of R_i are taken to be positive to get a unique Q factor in the QR decomposition.)

Base case. $X_1 = \text{qr}_Q(AX_{i-1}) = \text{qr}(A)$ and also $Q_1 = \text{qr}_Q(A)$ as well. Because $R_i = Q_1^T A = X_1^T A$, we get the same R_i factor here.

Inductive hypothesis. We assume that $X_i = Q_1 Q_2 \cdots Q_i$.

Important note. We saw this before, but $A_i = Q_i R_i$ implies that $R_i = Q_i^T A_i$. So $A_{i+1} = Q_i^T A_i Q_i$. If we iterate this, then note that

$$A_{i+1} = Q_i^T A_i Q_i = Q_i^T \cdots Q_1^T A Q_1 \cdots Q_i. \qquad \blacksquare$$

By our induction hypotheis $A_{i+1} = X_i^T A X_i$.

Note that $X_{i+1} R_{i+1} = \text{qr}(A X_i) = \text{qr}(A Q_1 \cdots Q_i)$.

So $X_{i+1}^T A Q_1 \cdots Q_i = R_{i+1}$.

29.3.4 *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 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 A to a matrix B with this property via orthogonal transformations $B = Q^T A Q$ or similarity transformations $B = X^{-1} A X$ (so we preserve eigenvalues)
2. computing a QR factorization of B is efficient
3. computing the product RQ is efficient
4. the matrix RQ has the same property as 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 B is tridiagonal, then 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

$$
B = \begin{bmatrix}
\bullet & \bullet & & & \\
\bullet & \bullet & \bullet & & \\
 & \bullet & \bullet & \bullet & \\
 & & \bullet & \bullet & \bullet \\
 & & & \bullet & \bullet
\end{bmatrix}
$$

Then in the first step of QR, we are going to "zero" out the 2, 1 entry via a Givens transform. This gives us:

$$
Q_1 = \begin{bmatrix}
\bullet & \bullet & & & \\
\bullet & \bullet & & & \\
 & & 1 & & \\
 & & & 1 & \\
 & & & & 1
\end{bmatrix}
\qquad
Q_1^T B = \begin{bmatrix}
\times & \times & \times & & \\
0 & \times & \times & & \\
 & \bullet & \bullet & \bullet & \\
 & & \bullet & \bullet & \bullet \\
 & & & \bullet & \bullet
\end{bmatrix}
$$

$$\mathbf{Q}_2 = \begin{bmatrix} 1 & & & & \\ & \bullet & \bullet & & \\ & \bullet & \bullet & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \qquad \mathbf{Q}_2^T \mathbf{Q}_1^T \mathbf{B} = \begin{bmatrix} \bullet & \bullet & \bullet & & \\ 0 & \times & \times & \times & \\ & 0 & \times & \times & \\ & & \bullet & \bullet & \bullet \\ & & & \bullet & \bullet \end{bmatrix}$$

Hence, at the end, we'll have

$$\mathbf{Q}_4^T \mathbf{Q}_3^T \mathbf{Q}_2^T \mathbf{Q}_1^T \mathbf{B} = \mathbf{R} = \begin{bmatrix} \bullet & \bullet & \bullet & & \\ 0 & \bullet & \bullet & \bullet & \\ & 0 & \bullet & \bullet & \bullet \\ & & 0 & \bullet & \bullet \\ & & & 0 & \bullet \end{bmatrix}$$

$$\mathbf{Q} = \mathbf{Q}_1 \mathbf{Q}_2 \mathbf{Q}_3 \mathbf{Q}_4 = \begin{bmatrix} \bullet & \bullet & & & \\ \bullet & \bullet & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & \\ & \bullet & \bullet & & \\ & \bullet & \bullet & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \cdots$$

So

$$\mathbf{RQ} = \underbrace{\begin{bmatrix} \bullet & \bullet & \bullet & & \\ 0 & \bullet & \bullet & \bullet & \\ & 0 & \bullet & \bullet & \bullet \\ & & 0 & \bullet & \bullet \\ & & & 0 & \bullet \end{bmatrix} \begin{bmatrix} \bullet & \bullet & & & \\ \bullet & \bullet & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix}}_{\text{linear combination of columns 1,2}} \mathbf{Q}_2 \mathbf{Q}_3 \mathbf{Q}_4 = \begin{bmatrix} \times & \times & \bullet & & \\ \times & \times & \bullet & \bullet & \\ & 0 & \bullet & \bullet & \bullet \\ & & 0 & \bullet & \bullet \\ & & & 0 & \bullet \end{bmatrix} \mathbf{Q}_2 \mathbf{Q}_3 \mathbf{Q}_4$$

$$= \begin{bmatrix} \bullet & \times & \times & & \\ \bullet & \times & \times & \bullet & \\ & \times & \times & \bullet & \bullet \\ & & 0 & \bullet & \bullet \\ & & & 0 & \bullet \end{bmatrix} \mathbf{Q}_3 \mathbf{Q}_4$$

$$= \begin{bmatrix} \bullet & \bullet & \times & \times & \\ \bullet & \bullet & \times & \times & \\ & \bullet & \times & \times & \bullet \\ & & \times & \times & \bullet \\ & & & 0 & \bullet \end{bmatrix} \mathbf{Q}_4$$

$$= \begin{bmatrix} \bullet & \bullet & \bullet & \times & \times \\ \bullet & \bullet & \bullet & \times & \times \\ & \bullet & \bullet & \times & \times \\ & & \bullet & \times & \times \\ & & & \times & \times \end{bmatrix}$$

We can keep going with this and we'll find that RQ is upper-Hessenberg.

But, $R = Q^T A$ so $RQ = Q^T Q Q$ is symmetric, upper-Hessenberg, i.e. tridiagonal!

29.4 REDUCTION TO TRIDIAGONAL FORM

EXAMPLE 29.10 A = randn(6)   ◆

29.5 QR WITH SHIFTS

29.6 EIGENVALUES AND EIGENVECTORS OF SPARSE MATRICES

Run the power method! Then we'll get

29.6.1 *Deflation*

29.6.2 *Subspace iteration*

29.6.3 *Lanczos*

29.6.4 *ARPACK*

EXAMPLE 29.11 This example shows how to use the eigs function in Matlab

```
A = sprandsym (10000 ,50/10000);
[V,D] = eigs(A);
```

ALGORITHMS FOR THE SVD

<div style="text-align:right">

30

</div>

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 A has more rows than columns.

VIEW 1 The singular values of the matrix A are the eigenvalues of $A^T A$

VIEW 2 . The singular values of the matrix A are the positive eigenvalues

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

The matrix B in view 2 is a specific instance of a bipartite matrix!

More generally, the theorem underlying **View 2** is

Theorem 30.1

Let $B = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$ and let $A = U\Sigma V^T$ be the SVD of A. Then the eigenvalues of B are $\pm\sigma_i$ along with $m + n - 2n$ additional zeros. Given an eigenvalue $+\sigma_i$ eigenvector $Bz = \sigma z$ if we partition $z = \begin{bmatrix} x \\ y \end{bmatrix}$. Then, $Ay = \sigma x$ is one of the singular vectors and value sets.

PROOF We have $B = \begin{bmatrix} 0 & U\Sigma V^T \\ V\Sigma^T U^T & 0 \end{bmatrix} = \begin{bmatrix} U & 0 \\ 0 & V \end{bmatrix}\begin{bmatrix} 0 & \Sigma \\ \Sigma^T & 0 \end{bmatrix}\begin{bmatrix} U & 0 \\ 0 & V \end{bmatrix}$.

— TODO – Show more of this matrix, including the number of zeros. Then note that there exists a permutation matrix P such that

$$P\begin{bmatrix} 0 & \Sigma \\ \Sigma^T & 0 \end{bmatrix}P^T = \begin{bmatrix} 0 & \sigma_1 & & & \\ \sigma_1 & 0 & & & \\ & & 0 & \sigma_2 & \\ & & \sigma_2 & 0 & \\ & & & & \ddots \end{bmatrix}$$

— 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 **B** instead of **A**. Of course, we do not actually form **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 **A** to a bidiagonal matrix **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 **B**.

```
EXAMPLE 30.2   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

DERIVATION OF THE HILBERT MATRIX

31

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:

$$E(\mathbf{c}) = \int_0^1 (f(x) - p(x))^2 \, dx = \int_0^1 (f(x) - \sum_{j=1}^k c_j x^{j-1})^2 \, dx$$

$$E(\mathbf{c}) = \sum_{j=1}^k \sum_{i=1}^k c_i c_j \int_0^1 x^{j-1} x^{i-1} \, dx - 2 \sum_{j=1}^k c_j \int_0^1 f(x) x^{j-1} \, dx + \int_0^1 f(x)^2 \, dx.$$

Hence, we have

$$E(\mathbf{c}) = \mathbf{c}^T A \mathbf{c} - 2\mathbf{c}^T \mathbf{b}$$

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 $A\mathbf{c} = \mathbf{b}$.

But the matrix 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 17).

PAGERANK

32

I love the PageRank problem and using it as an example throughout the book.

Much of my experience with these issues arises when trying to compute accurate solutions of the PageRank linear system of equations. Let A be the adjacency matrix of a directed graph. An entry of $A_{ij} = 1$ when there is a link from node i to node j and an entry has a value of 0 otherwise. For PageRank, we mathematically model a Markov process that behaves as follows. At node i

· with probability α, we follow an out-link 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.

Let X_t be the identity of the node at step t. The PageRank vector of the graph is defined as the amount of time that this process spends in each node as it runs forever:

$$x_i = \lim_{T \to \infty} \frac{1}{T} \sum_{k=0}^{T} \begin{cases} 1 & X_t = i \\ 0 & \text{otherwise} \end{cases}.$$

There are a variety of ways to get the following condition, but in the interest of time, let's just note that we can get there by simply assuming that x_i exists like we did with the hitting time analysis.[1] If there is such a vector **x**, we can convert this into a linear system of equations by noting that the value of x_i must depend on the other values x_j as in the next equation. Let d_j be the number of nodes that node j links to (so this is the out-degree of node j).

$$x_i = \alpha \sum_{j \text{ links to } i} \begin{cases} x_j/d_j & d_j > 0 \\ x_j/n & d_j = 0 \end{cases} + (1 - \alpha) \underbrace{\sum_{j=1}^{n} x_j/n}_{\text{random jumps}}$$

Let P be the matrix where

$$P_{i,j} = \begin{cases} 1/d_j & \text{node } j \text{ links to node } i, d_j > 0 \\ 0 & \text{otherwise.} \end{cases}$$

[1] Formally, this would require showing that the above limit and sum have a unique limit. Both are actually the case, but involve a study in stochastic processes that is outside the scope of this material. We will show a related derivation to justify existence.

If A is the adjacency matrix described above then

$$P = A^T D^{-1} \text{ where } D_{ij}^{-1} = \begin{cases} 1/d_i & i = j, d_j > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The matrix D^{-1} is really the inverse of the diagonal matrix of node degrees, where we simply skip over any node with degree 0.[2] Let $\mathbf{c}$ be the indicator vector where $c_i = 1$ for all nodes with $d_j = 0$ and $c_i = 0$ for all other nodes. Then

$$\mathbf{x} = \alpha P \mathbf{x} + \alpha/n \mathbf{e} \mathbf{c}^T \mathbf{x} + (1 - \alpha) \mathbf{e}/n.$$

In may cases, we assume the vector $\mathbf{c} = 0$. In which case
   Given the matrix A, the following algorithm

[2] This is often called the pseudoinverse.

CHESS RANKING

33

Discuss Vigna's early reference along with Richard Brent's paper.

COLOPHON   *VIII*

BIBLIOGRAPHY

33

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 36.

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 36.

L. Katz. *A new status index derived from sociometric analysis*. Psychometrika, 18 (1), pp. 39–43, 1953. `doi:10.1007/BF02289026`. Cited on page 36.

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 36.

BETTER NAMES IN MATRIX COMPUTATIONS

A

TOEPLITZ MATRIX	constant diagonal matrix
	simultaneous variable updates / solves / improvement / relaxation
JACOBI METHOD	parallel variable updates / solves / improvement / relaxation
GAUSS-SEIDEL METHOD	sequential variable updates / solves / improvement / relaxation
GAUSS-SEIDEL METHOD	sequential variable updates / solves / improvement / relaxation