C6.1 Numerical Linear Algebra

Yuji Nakatsukasa*

Welcome to numerical linear algebra (NLA)! NLA is a beautiful subject that combines mathematical rigor, amazing algorithms, and an extremely rich variety of applications.

What is NLA? In a sentence, it is a subject that deals with the numerical solution (i.e., using a computer) of linear systems $Ax = b$ (given $A \in \mathbb{R}^{n\times n}$ (i.e., a real $n \times n$ matrix) and $b \in \mathbb{R}^n$ (real n-vector), find $x \in \mathbb{R}^n$) and eigenvalue problems $Ax = \lambda x$ (given $A \in \mathbb{R}^{n\times n}$, find $\lambda \in \mathbb{C}$ and $x \in \mathbb{C}^n$), for problems that are too large to solve by hand ($n \geq 4$ is already large; we aim for n in the thousands or even millions). This can rightfully sound dull, and some mathematicians (those purely oriented?) tend to get turned off after hearing this — how could such a course be interesting compared with other courses offered by the Oxford Mathematical Institute? I hope and firmly believe that at the end of the course you will all agree that there is more to the subject than you imagined. The rapid rise of data science and machine learning has only meant that the importance of NLA is still growing, with a vast number of problems in these fields requiring NLA techniques and algorithms. It is perhaps worth noting also that these fields have had enormous impact on the direction of NLA, in particular the recent and very active field of randomised algorithm was born in light of needs arising from these extremely active fields.

In fact NLA is a truly exciting field that utilises a huge number of ideas from different branches of mathematics (e.g. matrix analysis, approximation theory, and probability) to solve problems that actually matter in real-world applications. Having said that, the number of prerequisites for taking the course is the bare minimum; essentially a basic understanding of the fundamentals of linear algebra would suffice (and the first lecture will briefly review the basic facts). If you've taken the Part A Numerical Analysis course you will find it helpful, but again, this is not necessary.

The field NLA has been blessed with many excellent books on the subject. These notes will try to be self-contained, but these references will definitely help. There is a lot to learn; literally as much as you want to.

- Trefethen-Bau (97) [36]: Numerical Linear Algebra
  - covers essentials, beautiful exposition
- Golub-Van Loan (12) [15]: Matrix Computations
  - classic, encyclopedic

*Please report any corrections or comments on these lecture notes to `nakatsukasa@maths.ox.ac.uk`

- Horn and Johnson (12) [22]: Matrix Analysis (& Topics in Matrix Analysis (86) [21])
    - excellent theoretical treatise, little numerical treatment
- J. Demmel (97) [9]: Applied Numerical Linear Algebra
    - impressive content
- N. J. Higham (02) [19]: Accuracy and Stability of Algorithms
    - bible for stability, conditioning
- H. C. Elman, D. J. Silvester, A. J. Wathen (14) [12]: Finite Elements and Fast Iterative Solvers
    - PDE applications of linear systems, Krylov methods and preconditioning

This course covers the fundamentals of NLA. We first discuss the singular value decomposition (SVD), which is a fundamental matrix decomposition whose importance is only growing. We then turn to linear systems and eigenvalue problems. Broadly, we will cover

- Direct methods ($n \lesssim 10{,}000$): Sections 5–10 (except 8)
- Iterative methods ($n \lesssim 1{,}000{,}000$, sometimes larger): Sections 11–13
- Randomised methods ($n \gtrsim 1{,}000{,}000$): Sections 14–16

in this order. Lectures 1–4 cover the fundamentals of matrix theory, in particular the SVD, its properties and applications.

This document consists of 16 sections. Very roughly speaking, one section corresponds to one lecture (though this will not be followed strictly at all).

Contents

Notation. For convenience below we list the notation that we use throughout the course.

- $\lambda(A)$: the set of eigenvalues of A. If a natural ordering exists (e.g. A is symmetric so λ is real), $\lambda_i(A)$ is the ith (largest) eigenvalue.
- $\sigma(A)$: the set of singular values of A. $\sigma_i(A)$ always denotes the ith largest singular value. We often just write σ_i.
- $\mathrm{diag}(A)$: the vector of diagonal entries of A.
- We use capital letters for matrices, lower-case for vectors and scalars. Unless otherwise specified, A is a given matrix, b is a given vector, and x is an unknown vector.
- $\|\cdot\|$ denotes a norm for a vector or matrix. $\|\cdot\|_2$ denotes the spectral (or 2-) norm, $\|\cdot\|_F$ the Frobenius norm. For vectors, to simplify notation we sometimes use $\|\cdot\|$ for the spectral norm (which for vectors is the familiar Euclidean norm).
- $\mathrm{Span}(A)$ denotes the span or range of the column space of A. This is the subspace consisting of vectors of the form Ax.

- We reserve Q for an orthonormal (or orthogonal) matrix. $L, (U)$ are often lower (upper) triangular.

- I always denotes the identity matrix. I_n is the $n \times n$ identity when the size needs to be specified.

- A^T is the transpose of the matrix; $(A^T)_{ij} = A_{ji}$. A^* is the (complex) conjugate transpose $(A^*)_{ij} = \bar{A}_{ji}$.

- $\succ, \succeq$ denote the positive (semi)definite ordering. That is, $A \succ (\succeq) 0$ means A is positive (semi)definite (abbreviated as PD, PSD), i.e., symmetric and with positive (nonnegative) eigenvalues. $A \succ B$ means $A - B \succ 0$.

We sometimes use the following shorthand: alg for algorithm, eigval for eigenvalue, eigvec for eigenvector, singval for singular value, and singvec for singular vector, iff for "if and only if".

0 Introduction, why $Ax = b$ and $Ax = \lambda x$?

As already stated, NLA is the study of numerical algorithms for problems involving matrices, and there are only two main problems(!):

1. Linear system
$$Ax = b.$$
Given a (often square $m = n$ but we will discuss $m > n$ extensively, and $m < n$ briefly at the end) matrix $A \in \mathbb{R}^{m\times n}$ and vector $b \in \mathbb{R}^m$, find $x \in \mathbb{R}^n$ such that $Ax = b$.

2. Eigenvalue problem
$$Ax = \lambda x.$$
Given a (always![1]) square matrix $A \in \mathbb{R}^{n\times n}$ find λ: eigenvalues (eigval), and $x \in \mathbb{R}^n$: eigenvectors (eigvec).

We'll see many variants of these problems; one worthy of particular mention is the SVD, which is related to eigenvalue problems but given its ubiquity has a life of its own. (So if there's a third problem we solve in NLA, it would definitely be the SVD.)

It is worth discussing *why* we care about linear systems and eigenvalue problems.

The primary reason is that many (in fact most) problems in scientific computing (and even machine learning) boil down to linear problems:

- Because that's often the only way to deal with the scale of problems we face today! (and in future)

[1]There are exciting recent developments involving eigenvalue problems for rectangular matrices, but these are outside the scope of this course.

- For linear problems, so much is understood and reliable algorithms are available[2].

A related important question is where and how these problems arise in real-world problems.

Let us mention a specific context that is relevant in data science: optimisation. Suppose one is interested in minimising a high-dimensional real-valued function $f(x) : \mathbb{R}^n \to \mathbb{R}$ where $n \gg 1$.

A successful approach is to try and find critical points, that is, points x_* where $\nabla f(x_*) = 0$. Mathematically, this is a non-linear high-dimensional root-finding problem of finding $x \in \mathbb{R}^n$ such that $\nabla f(x) =: F(x) = 0$ (the vector $0 \in \mathbb{R}^n$) where $F : \mathbb{R}^n \to \mathbb{R}^n$. One of the most commonly employed methods for this task is Newton's method (which some of you have seen in Prelims Constructive Mathematics). This boils down to

- Newton's method for $F(x) = 0$, $F : \mathbb{R}^n \to \mathbb{R}^n$ nonlinear:

  1. Start with initial guess $x^{(0)} \in \mathbb{R}^n$, set $i = 0$
  2. Find Jacobian matrix $J \in \mathbb{R}^{n\times n}$, $J_{ij} = \frac{\partial F_i(x)}{\partial x_j}|_{x=x^{(0)}}$
  3. Update $x^{(i+1)} := x^{(i)} - J^{-1}F(x^{(i)})$, $i \leftarrow i+1$, go to step 2 and repeat

  Note that the main computational task is to find the vector $y = J^{-1}F(x^{(i)})$, which is a linear system $Jy = F(x^{(i)})$ (which we solve for the vector y)

What about eigenvalue problems $Ax = \lambda x$? Google's pagerank is a famous application (we will cover this if we have time). Another example the Schrödinger equation of physics and chemistry. Sometimes a nonconvex optimisation problem can be solved by an eigenvalue problem.

Equally important is principal component analysis (PCA), which can be used for data compression. This is more tightly connected to the SVD.

Other sources of linear algebra problems include differential equations, optimisation, regression, data analysis, ...

1 Basic LA review

We start with a review of key LA facts that will be used in the course. Some will be trivial to you while others may not. You might also notice that some facts that you have learned in a core LA course will not be used in this course. For example we will never deal with finite fields, and determinants only play a passing role.

[2]A pertinent quote is Richard Feynman's "Linear systems are important because we can solve them". Because we can solve them, we do all sorts of tricks to reduce difficult problems to linear systems!

1.1 Warmup exercise

Let $A \in \mathbb{R}^{n\times n}$ ($n \times n$ square matrix). (or $\mathbb{C}^{n\times n}$; the difference hardly matters in most of this course[3]). Try to think of statements that are equivalent to A being nonsingular. Try to come up with as many conditions as possible, before turning the page.

[3]While there are a small number of cases where the distinction between real and complex matrices matters, in the majority of cases it does not, and the argument carries over to complex matrices by replacing $\cdot^T$ with $\cdot^*$. Therefore for the most part, we lose no generality in assuming the matrix is real (which slightly simplifies our mindset). Whenever necessary, we will highlight the subtleties that arise resulting from the difference between real and complex. (For the curious, these are the Schur form/decomposition, LDL^T factorisation and eigenvalue decomposition for (real) matrices with complex eigenvalues.)

Here is a list: The following are equivalent.

1. A is nonsingular.
2. A is invertible: A^{-1} exists.
3. The map $A : \mathbb{R}^n \to \mathbb{R}^n$ is a bijection.
4. All n eigenvalues of A are nonzero.
5. All n singular values of A are positive.
6. $\mathrm{rank}(A) = n$.
7. The rows of A are linearly independent.
8. The columns of A are linearly independent.
9. $Ax = b$ has a solution for every $b \in \mathbb{C}^n$.
10. A has no nonzero null vector.
11. A^T has no nonzero null vector.
12. A^*A is positive definite (not just semidefinite).
13. $\det(A) \neq 0$.
14. An $n \times n$ matrix A^{-1} exists such that $A^{-1}A = I_n$. (this, btw, implies (iff) $AA^{-1} = I_n$, a nontrivial fact)
15. ... (what did I miss?)

1.2 Structured matrices

We will be discussing lots of structured matrices. For square matrices,

- Symmetric: $A_{ij} = A_{ji}$ (Hermitian: $A_{ij} = \bar{A_{ji}}$)
  - The most important property of symmetric matrices is the symmetric eigenvalue decomposition $A = V\Lambda V^T$; V is orthogonal $V^TV = VV^T = I_n$, and Λ is a diagonal matrix of eigenvalues $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$.
  - symmetric positive (semi)definite $A \succ (\succeq) 0$: symmetric and all positive (nonnegative) eigenvalues.
- Orthogonal: $AA^T = A^TA = I$ (Unitary: $AA^* = A^*A = I$). Note that for square matrices, $A^TA = I$ implies $AA^T = I$.
- Skew-symmetric: $A_{ij} = -A_{ji}$ (skew-Hermitian: $A_{ij} = -\bar{A_{ji}}$).

- Normal: $A^TA = AA^T$. (Here it's better to discuss the complex case $A^*A = AA^*$: this is a necessary and sufficient condition for diagonalisability under a unitary transformation, i.e., $A = U\Lambda U^*$ where Λ is diagonal and U is unitary.)
- Tridiagonal: $A_{ij} = 0$ if $|i-j| > 1$.
- Upper triangular: $A_{ij} = 0$ if $i > j$.
- Lower triangular: $A_{ij} = 0$ if $i < j$.

For (possibly nonsquare) matrices $A \in \mathbb{C}^{m\times n}$, (usually $m \geq n$).

- (upper) Hessenberg: $A_{ij} = 0$ if $i > j+1$. (we will see this structure often.)
- "orthonormal": $A^TA = I_n$, and A is (tall) rectangular. (This isn't an established name—we could call it "matrix with orthonormal columns" every time it appears—but we use these matrices all the time in this course, so we need a consistent shorthand name for it.)
- sparse: most elements are zero. nnz(A) denotes the number of nonzero elements in A. Matrices that are not sparse are called *dense*.

Other structures: Hankel, Toeplitz, circulant, symplectic,... (we won't use these in this course)

1.3 Matrix eigenvalues: basics

$$Ax = \lambda x, \quad A \in \mathbb{R}^{n\times n}, (0 \neq) x \in \mathbb{R}^n$$

- Example: $\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = 4 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

  This matrix has an *eigenvalue* $\lambda = 4$, with corresponding *eigenvector* $x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$ (together they are an *eigenpair*).
  An $n \times n$ matrix always has n eigenvalues (not always n linearly independent eigenvectors); In the example above, $(\lambda, x) = \left(1, \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}\right), \left(1, \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}\right)$ are also eigenpairs.

- The eigenvalues are the roots of the characteristic polynomial $\det(\lambda I - A) = 0$: $\det(\lambda I - A) = \prod_{i=1}^n (\lambda - \lambda_i)$.
- According to Galois theory, eigenvalues cannot be computed exactly for matrices with $n \geq 5$. **But we still want to compute them!** In this course we will (among other things) explain how this is done in practice by the *QR algorithm*, one of the greatest hits of the field.

1.4 Computational complexity (operation counts) of matrix algorithms

Since NLA is a field that aspires to develop practical algorithms for solving matrix problems, it is important to be aware of the computational cost (often referred to as complexity) of the algorithms. We will discuss these as the algorithms are developed, but for now let's examine the costs for basic matrix-matrix multiplication. The cost is measured in terms of flops (floating-point operations), which counts the number of additions, subtractions, multiplications, and divisions (all treated equally) performed.

In NLA the constant in front of the leading term in the cost is (clearly) important. It is customary (for good reason) to only track the leading term of the cost. For example, $n^3 + 10n^2$ is abbreviated to n^3.

- Multiplying two $n \times n$ matrices AB costs $2n^3$ flops. More generally, if A is $m \times n$ and B is $n \times k$, then computing AB costs $2mnk$ flops.

- Multiplying a vector to an $m \times n$ matrix A costs $2mn$ flops.

Norms

We will need a tool (or metric) to measure how big a vector or matrix is. For example, when we approximate a matrix A with another one $\hat{A}$, we wish to be able to quantify how close they are by measuring how 'big' $A - \hat{A}$ is. Norms give us a means to achieve this. Surely you have already seen some norms (e.g. the vector Euclidean norm). We will discuss a number of norms for vectors and matrices that we will use in the upcoming lectures.

1.5 Vector norms

For vectors $x = [x_1, \dots, x_n]^T \in \mathbb{C}^n$

- p-norm $\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p} \quad (1 \le p \le \infty)$
  - Euclidean norm=2-norm $\|x\|_2 = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}$
  - 1-norm $\|x\|_1 = |x_1| + |x_2| + \cdots + |x_n|$
  - ∞-norm $\|x\|_\infty = \max_i |x_i|$

Of particular importance are the three cases $p = 1, 2, \infty$. In this course, we will see $p = 2$ the most often.

A norm needs to satisfy the following axioms:

- $\|\alpha x\| = |\alpha| \|x\|$ for any $\alpha \in \mathbb{C}$ (homogeneity),

- $\|x\| \ge 0$ and $\|x\| = 0 \Leftrightarrow x = 0$ (nonnegativity),

- $\|x+y\| \le \|x\| + \|y\|$ (triangle inequality).

The vector p-norm satisfies all these, for any p.

Here are some useful inequalities for vector norms. A proof is left for your exercise and is highly recommended. (Try to think when each equality is satisfied.) For $x \in \mathbb{C}^n$,

- $\frac{1}{\sqrt{n}}\|x\|_2 \le \|x\|_\infty \le \|x\|_2$
- $\frac{1}{\sqrt{n}}\|x\|_1 \le \|x\|_2 \le \|x\|_1$
- $\frac{1}{n}\|x\|_1 \le \|x\|_\infty \le \|x\|_1$

Note that with the 2-norm, $\|Ux\|_2 = \|x\|_2$ for any unitary U and any $x \in \mathbb{C}^n$. Norms with this property are called **unitarily invariant**.

The 2-norm is also induced by the inner product $\|x\|_2 = \sqrt{x^T x}$. An important property of inner products is the Cauchy-Schwarz inequality $|x^T y| \le \|x\|_2\|y\|_2$ (which can be directly proved but is perhaps best to prove in general setting)[4]. When we just say $\|x\|$ for a vector we mean the 2-norm.

1.6 Matrix norms

We now turn to norms of matrices. As you will see, many (but not the Frobenius and trace norms) are defined via the vector norms (these are called *induced* norms).

- p-norm $\|A\|_p = \max_x \frac{\|Ax\|_p}{\|x\|_p}$
  - 2-norm=spectral norm(=Euclidean norm) $\|A\|_2 = \sigma_{\max}(A)$ (largest singular value; see Section 2)
  - 1-norm $\|A\|_1 = \max_i \sum_{j=1}^m |A_{ji}|$
  - ∞-norm $\|A\|_\infty = \max_i \sum_{j=1}^n |A_{ij}|$
- Frobenius norm $\|A\|_F = \sqrt{\sum_i \sum_j |A_{ij}|^2}$
  (2-norm of vectorisation)
- trace norm=nuclear norm $\|A\|_* = \sum_{i=1}^{\min(m,n)} \sigma_i(A)$. (this is the maximum trace of $Q^T A$ where Q is orthonormal, hence the name)

Colored in red are **unitarily invariant** norms $\|A\|_* = \|UAV\|_*$, $\|A\|_F = \|UAV\|_F$, $\|A\|_2 = \|UAV\|_2$ for any unitary/orthogonal U, V.

Norm axioms hold for each of these. Useful inequalities include: For $A \in \mathbb{C}^{m\times n}$, (exercise; it is instructive to study the cases where each of these equalities holds)

[4] Just in case, here's a proof: for any scalar c, $\|x - cy\|^2 = \|x\|^2 - 2cx^T y + c^2\|y\|^2$. This is minimised wrt c at $c = \frac{x^T y}{\|y\|^2}$ with minimiser $\|x\|^2 - \frac{(x^T y)^2}{\|y\|^2}$. Since this must be ≥ 0, the CS inequality follows.

- $\frac{1}{\sqrt{n}}\|A\|_\infty \le \|A\|_2 \le \sqrt{m}\|A\|_\infty$
- $\frac{1}{\sqrt{m}}\|A\|_1 \le \|A\|_2 \le \sqrt{n}\|A\|_1$
- $\|A\|_2 \le \|A\|_F \le \sqrt{\min(m,n)}\|A\|_2$

A useful property of p-norms is that they are subordinate, i.e., $\|AB\|_p \le \|A\|_p\|B\|_p$ (problem sheet). Note that not all norms satisfy this, e.g. with the max norm $\|A\|_{\max} = \max_{i,j}|A_{ij}|$, with $A = [1,1]$ and $B = \begin{bmatrix}1\\1\end{bmatrix}$ one has $\|AB\|_{\max} = 2$ but $\|A\|_{\max} = \|B\|_{\max} = 1$.

1.7 Subspaces and orthonormal matrices

A key notion that we will keep using throughout is a **subspace** $\mathcal{S}$. In this course we will almost exclusively confine ourselves to subspaces of $\mathbb{R}^n$, even though they generalize to more abstract vector spaces. A subspace is the set of vectors that can be written as a linear combination **basis vectors** $v_1, \dots, v_d$, which are assumed to be linearly independent (otherwise there is a basis with fewer vectors). That is, $x \in \mathcal{S}$ iff $\sum_{i=1}^d c_i v_i$ (where c_i are scalars, $\mathbb{R}$ or $\mathbb{C}$). The integer d is called the *dimension* of the subspace. We also say the subspace is *spanned* by the vectors $v_1, \dots, v_d$, or that $v_1, \dots, v_d$ spans the subpsace.

How does one represent a subspace? An obvious answer is to use the basis vectors $v_1, \dots, v_d$. This sometimes becomes cumbersome, and a common and convenient way to represent the subspace is to use a (tall-skinny) rectangular matrix $V \in \mathbb{R}^{n\times d} = [v_1, v_2, \dots, v_d]$, as $\mathcal{S} = \text{span}(V)$ (or sometimes just "subspace V") which means the subspace of vectors that can be written as Vc, where c is a 'coefficient' vector $c \in \mathbb{R}^d$.

It will be (not necessary but) convenient to represent subspaces using an orthonormal matrix $Q \in \mathbb{R}^{n\times d}$. (once we cover the QR factorisation, you'll see that there is no loss of generality in doing so).

An important fact about subspaces of $\mathbb{R}^n$ is the following:

Lemma 1.1 *Let $V_1 \in \mathbb{R}^{n\times d_1}$ and $V_2 \in \mathbb{R}^{n\times d_2}$ each have linearly independent column vectors. If $d_1 + d_2 > n$, then there is a nonzero* intersection *between two subspaces $\mathcal{S}_1 = span(V_1)$ and $\mathcal{S}_2 = span(V_2)$, that is, there is a nonzero vector $x \in \mathbb{R}^n$ such that $x = V_1c_1 = V_2c_2$ for some vectors c_1, c_2.*

This is straightforward but important enough to warrant a proof.

Proof: Consider the matrix $M := [V_1, V_2]$, which is of size $n \times (d_1 + d_2)$. Since $d_1 + d_2 > n$ by assumption, this matrix has a right null vector[5] $c \ne 0$ such that $Mc = 0$. Splitting $c = \begin{bmatrix}c_1\\-c_2\end{bmatrix}$ we have the required result. $\square$

Let us conclude this review with a list of useful results that will be helpful. Proofs (or counterexamples) should be straightforward.

[5]If this argument isn't convincing to you now, probably the easiest way to see this is via the SVD; so stay tuned and we'll resolve this in footnote 9, once you've seen the SVD!

- $(AB)^T = B^T A^T$
- If A, B invertible, $(AB)^{-1} = B^{-1}A^{-1}$
- If A, B square and $AB = I$, then $BA = I$
- $\begin{bmatrix} I_m & X \\ 0 & I_n \end{bmatrix}^{-1} = \begin{bmatrix} I_m & -X \\ 0 & I_n \end{bmatrix}$
- Neumann series: if $\|X\| < 1$ in any subordinate norm,
$$(I - X)^{-1} = I + X + X^2 + X^3 + \cdots$$
- For a square $n \times n$ matrix A, the trace is $\text{Trace}(A) = \sum_{i=1}^n A_{i,i}$ (sum of diagonals). For any X, Y such that XY is square, $\text{Trace}(XY) = \text{Trace}(YX)$ (quite useful). For $B \in \mathbb{R}^{m\times n}$, we have $\|B\|_F^2 = \sum_i \sum_j |B_{ij}|^2 = \text{Trace}(B^T B)$.
- Triangular structure (upper or lower) is invariant under addition, multiplication, and inversion. That is, triangular matrices form a ring (in abstract algebra; don't worry if this is foreign to you).
- Symmetry is invariant under addition and inversion, *but not multiplication*; AB is usually not symmetric even if A, B are.

2 SVD: the most important matrix decomposition

We now start the discussion of the most important topic of the course: the singular value decomposition (SVD). The SVD exists for any matrix, square or rectangular, real or complex.

We will prove its existence and discuss its properties and applications in particular in low-rank approximation, which can immediately be used for compressing the matrix, and therefore data.

The SVD has many intimate connections to symmetric eigenvalue problems. Let's start with a review.

- **Symmetric eigenvalue decomposition**: Any symmetric matrix $A \in \mathbb{R}^{n\times n}$ has the decomposition
$$A = V\Lambda V^T \tag{1}$$
where V is orthogonal, $V^T V = I_n = VV^T$, and $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$ is a diagonal matrix of eigenvalues.

λ_i are the eigenvalues, and V is the matrix of eigenvectors (its columns are the eigenvectors).

The decomposition (1) makes two remarkable claims: the eigenvectors can be taken to be orthogonal (which is true more generally of *normal* matrices s.t. $A^*A = AA^*$), and the eigenvalues are real.

It is worth reminding you that eigenvectors are not uniquely determined: (assuming they are normalised s.t. the 2-norm is 1) their signs can always be flipped. (And actually more, when there are eigenvalues that are multiple $\lambda_i = \lambda_j$; in this case the eigenvectors span a subspace whose dimension matches the multiplicity. For example, any vector is an eigenvector of the identity matrix I).

Now here is the protagonist of this course: Be sure to spend dozens (if not hundreds) of hours thinking about it!

Theorem 2.1 (Singular Value Decomposition (SVD)) *Any matrix $A \in \mathbb{R}^{m\times n}$ has the decomposition :*

$$A = U\Sigma V^T. \tag{2}$$

Here $U^TU = V^TV = I_n$ (assuming $m \geq n$ for definiteness), $\Sigma = diag(\sigma_1, \dots, \sigma_n)$, $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0$.

σ_i (always nonnegative) are called the *singular values* of A. The *rank* of A is the number of positive singular values. The columns of U are called the *left singular vectors*, and the columns of V are the *right singular vectors.*

Writing $U = [u_1, \dots, u_n]$ and $V = [v_1, \dots, v_n]$, we have an important alternative expression for A: $A = \sum_{i=1}^n \sigma_i u_i v_i^T$. We will use this expression repeatedly in what follows. Also note that we always order the singular values in nonincreasing order, so σ_1 is always the largest singular value.

The SVD tells us that any (tall) matrix can be written as orthonormal-diagonal-orthogonal. Roughly, ortho-normal/gonal matrices can be thought of as rotations or reflection, so the SVD says the action of a matrix can be thought of as a rotation/reflection followed by magnification (or shrinkage), followed by another rotation/reflection.

Proof: For SVD[6] ($m \geq n$ and assume full-rank $\sigma_n > 0$ for simplicity): Take Gram matrix A^TA (symmetric) and its eigendecomposition $A^TA = V\Lambda V^T$ with V orthogonal. Λ is nonnegative, and $(AV)^T(AV) =: \Sigma^2$ is diagonal, so $AV\Sigma^{-1} =: U$ is orthonormal. Right-multiply by ΣV^T to get $A = U\Sigma V^T$. $\square$

It is also worth mentioning the "full" SVD: $A = [U, U_\perp]\begin{bmatrix}\Sigma \\ 0\end{bmatrix} V^T$ where $[U, U_\perp] \in \mathbb{R}^{m\times m}$ is square and orthogonal. $U_\perp \in \mathbb{R}^{m\times m-n}$ (pronounced 'U-perp', for perpendicular) is an orthonormal matrix such that $U^TU_\perp = 0$ (whose existence and construction can be established via the Householder QR factorsation in Section 6). Essentially the full SVD follows from the (thin) SVD (2) by filling in U in (2) with its orthogonal complement $U_\perp$.

[6] I like to think this is the shortest proof out there.

Example 2.1 *Let's find the SVD of* $A = \begin{bmatrix} -1 & -2 \\ 2 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$. *We can follow the proof and compute the Gram matrix* $A^TA = \begin{bmatrix} 6 & 4 \\ 4 & 6 \end{bmatrix}$. *Its eigenvalues are* 10 *and* 2 *(via the characteristic polynomial* $(\lambda - 6)(\lambda - 6) - 4^2 = 0$*—please note though that this isn't how they should be computed on a computer (the QR algorithm), which we'll cover later.) An eigenvector matrix can be found as* $V = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$ *(degree of freedom in the signs of the columns), computed e.g. via the null vector of* $(A - \lambda I)$. *Hence* $A^TA = V\Sigma^2V^T$ *where* $\Sigma^2 = \begin{bmatrix} 10 & \\ & 2 \end{bmatrix}$. *Let* $U = AV\Sigma^{-1} = \begin{bmatrix} -3/\sqrt{20} & -1/2 \\ 3/\sqrt{20} & -1/2 \\ 1/\sqrt{20} & -1/2 \\ 1/\sqrt{20} & 1/2 \end{bmatrix}$, *which is orthonormal. We thus have* $A = U\Sigma V^T$.

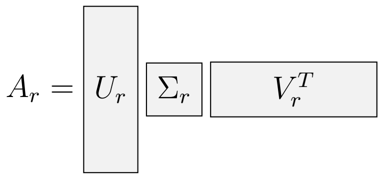
2.1 (Some of the many) applications and consequences of the SVD: rank, column/row space, etc

From the SVD one can immediately read off a number of important properties of the matrix. For example:

- The *rank* r of $A \in \mathbb{R}^{m\times n}$, often denoted rank$(A)$: this is the number of nonzero (positive) singular values $\sigma_i(A)$. rank(A) is also equal to the number of linearly independent rows or the number of independent columns, as you probably learnt in your first course on linear algebra (exercise).

  – We can always write $A = \sum_{i=1}^{\text{rank}(A)} \sigma_i u_i v_i^T$.

  Important: An $m \times n$ matrix A that is of rank r can be written as an (outer) product of $m \times r$ and $r \times n$ matrices:



  To see this, note from the SVD that $A = \sum_{i=1}^{n} \sigma_i u_i v_i^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T$ (since $\sigma_{r+1} = 0$), and so $A = U_r\Sigma_r V_r^T$ where $U_r = [u_1, \dots, u_r]$, $V_r = [v_1, \dots, v_r]$, and Σ_r is the leading $r \times r$ submatrix of Σ.

- Column space (Span(A), linear subspace spanned by vectors Ax): span of $U = [u_1, \dots, u_r]$, often denoted Span(U).

- Row space (Span(A^T)): row span of $v_1^T, \dots, v_r^T$. Span(V).

- Null space Null(A): span of $v_{r+1}, \dots, v_n$; as $Av = 0$ for these vectors. Null(A) is empty (or just the 0 vector) if $m \geq n$ and $r = n$. (When $m < n$ the full SVD is needed to describe Null(A).)

- We have $Av_i = \sigma_i u_i$, and $u_i^T A = \sigma_i v_i$. If σ_i, u_i, v_i satisfy this pair of equations, they are called a (the ith) singular triplet of A.

Aside from these and other applications, the SVD is also a versatile theoretical tool. Very often, a good place to start in proving a fact about matrices is to first consider its SVD; you will see this many times in this course. For example, the SVD can give solutions immediately for linear systems and least-squares problems, though there are more efficient ways to solve these problems.

2.2 SVD and symmetric eigenvalue decomposition

As mentioned above, the SVD and the eigenvalue decomposition for symmetric matrices are closely connected. Here are some results that highlight the connections between $A = U\Sigma V^T$ and symmetric eigenvalue decomposition. (We assume $m \geq n$ for definiteness)

- V is an eigvector matrix of A^TA. (To verify, see proof of SVD)

- U is an eigvector matrix (for nonzero eigvals) of AA^T (be careful with sign flips; see below)

- $\sigma_i = \sqrt{\lambda_i(A^TA)}$ for $i = 1, \dots, n$

- If A is symmetric, its singular values $\sigma_i(A)$ are the absolute values of its eigenvalues $\lambda_i(A)$, i.e., $\sigma_i(A) = |\lambda_i(A)|$.

  Exercise: What if A is unitary, skew-symmetric, normal matrices, triangular? (problem sheet)

- Jordan-Wieldant matrix $\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$: This matrix has eigenvalues $\pm\sigma_i(A)$, and $m-n$ copies of eigenvalues at 0. Its eigenvector matrix is $\begin{bmatrix} U & U & U_\perp \\ V & -V & 0 \end{bmatrix}$, where $A^T U_\perp = 0$ ($U_\perp$ is empty when $m = n$). This matrix, along with the Gram matrix A^TA, is a very useful tool when one tries to extend a result on symmetric eigenvalues to an analogue in terms of the SVD (or vice versa).

2.3 Uniqueness etc

We have established the existence of the SVD $A = U\Sigma V^T$. A natural question is: is it unique? In other words, are the factors U, Σ, V uniquely determined by A?

It is straightforward to see that the singular vectors are not uniquely determined. Most obviously, the singular vectors can be flipped in signs, just like eigenvectors. However, note that the signs of u_i, v_i are not entirely arbitrary: if we replace u_i by $-u_i$, the same needs to be done for v_i in order to satisfy $A = \sum_{i=1}^n \sigma_i u_i v_i^T$. Essentially, once the sign (or rotation) of u_i (or v_i) is fixed, v_i (or u_i) is determined uniquely.

More generally, in the presence of multiple singular values (i.e., $\sigma_i = \sigma_{i+1}$ for some i), there is a higher degree of freedom in the SVD. Again this is very much analogous to eigenvalues (recall the discussion on eigenvectors of the identity). Here think of what the SVD is for an orthogonal matrix: there is an enormous amount of degrees of freedom in the choice of U and V. The singular values σ_i, on the other hand, are always unique, as they are the eigenvalues of the Gram matrix.

3 Low-rank approximation via truncated SVD

While the SVD has a huge number of applications, undoubtedly the biggest reason that makes it so important in computational mathematics is its optimality for low-rank approximation.

To discuss this topic we need some preparation. We will make heavy use of the spectral norm of matrices. We start with an important characterisation of $\|A\|_2$ in terms of the singular value(s), which we previously stated but did not prove.

Proposition 3.1

$$\|A\|_2 = \max_x \frac{\|Ax\|_2}{\|x\|_2} = \max_{\|x\|_2=1} \|Ax\|_2 = \sigma_1(A).$$

Proof: Use the SVD: For any x with unit norm $\|x\|_2 = 1$,

$$\begin{aligned}
\|Ax\|_2 &= \|U\Sigma V^T x\|_2 \\
&= \|\Sigma V^T x\|_2 \quad \text{by unitary invariance} \\
&= \|\Sigma y\|_2 \quad \text{with } \|y\|_2 = 1 \\
&= \sqrt{\sum_{i=1}^n \sigma_i^2 y_i^2} \\
&\le \sqrt{\sum_{i=1}^n \sigma_1^2 y_i^2} = \sigma_1 \|y\|_2^2 = \sigma_1.
\end{aligned}$$

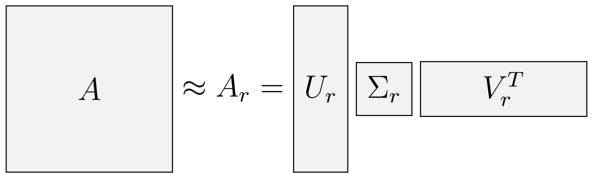
Finally, note that taking $x = v_1$ (the leading right singular vector), we have $\|Av_1\|_2 = \sigma_1$. $\square$

Similarly, the Frobenius norm can be expressed as $\|A\|_F = \sqrt{\sum_i \sum_j |A_{ij}|^2} = \sqrt{\sum_{i=1}^{n}(\sigma_i(A))^2}$, and the trace norm is (by definition) $\|A\|_* = \sum_{i=1}^{\min(m,n)} \sigma_i(A)$. (exercise) In general, norms that are *unitarily invariant* can be characterised by the singular values [22].

Now to the main problem: Given $A \in \mathbb{R}^{m\times n}$, consider the problem of finding a *rank-r matrix* (remember; these are matrices with r nonzero singular values) $A_r \in \mathbb{R}^{m\times n}$ that best approximates A. That is, find the minimiser A_r for

$$\operatorname{argmin}_{\operatorname{rank}(A_r)\le r} \|A - A_r\|_2. \tag{3}$$

It is definitely worth visualizing the situation:



We immediately see that a low rank approximation (when possible) is beneficial in terms of the storage cost when $r \ll m, n$. Instead of storing mn entries for A, we can store entries for U_r, Σ_r, V_r ($(m+n+1)r$ entries) to keep the low-rank factorisation without losing information.

Low-rank approximation can also bring computational benefits. For example, in order to compute Ax for a vector x, by noting that $A_r x = U_r(\Sigma_r(V_r^T x))$, one needs only $O((m+n)r)$ operations[7] instead of $O(mn)$. The utility and prevalence of low-rank matrices in data science is remarkable.

Here is the solution for (3): Truncated SVD, defined via $A_r = \sum_{i=1}^{r} \sigma_i u_i v_i^T (= U_r \Sigma_r V_r^T)$. This is the matrix obtained by truncating (removing) the trailing terms in the expression $A = \sum_{i=1}^{n} \sigma_i u_i v_i^T$. Pictorially,

$$A = \underbrace{\begin{bmatrix} * \\ * \\ \vdots \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & \cdots & * & * \end{bmatrix}}_{\sigma_1 u_1 v_1} + \underbrace{\begin{bmatrix} * \\ * \\ \vdots \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & \cdots & * & * \end{bmatrix}}_{\sigma_2 u_2 v_2} + \cdots + \underbrace{\begin{bmatrix} * \\ * \\ \vdots \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & \cdots & * & * \end{bmatrix}}_{\sigma_n u_n v_n},$$

$$A_r = \underbrace{\begin{bmatrix} * \\ * \\ \vdots \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & \cdots & * & * \end{bmatrix}}_{\sigma_1 u_1 v_1} + \cdots + \underbrace{\begin{bmatrix} * \\ * \\ \vdots \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & \cdots & * & * \end{bmatrix}}_{\sigma_r u_r v_r}.$$

In particular, we have

[7] I presume you've seen the big-Oh notation before—$f(n) = O(n^k)$ means there exists a constant C s.t. $f(n)/n^k \le C$ for all sufficiently large n. In NLA it is a convenient way to roughly measure the operation count.

Theorem 3.1 *For any $A \in \mathbb{R}^{m\times n}$ with singular values $\sigma_1(A) \geq \sigma_2(A) \geq \cdots \geq \sigma_n(A) \geq 0$, and any nonnegative integer*[8] $r < \min(m,n)$,

$$\|A - A_r\|_2 = \sigma_{r+1}(A) = \min_{\mathit{rank}(B)\leq r} \|A - B\|_2. \tag{4}$$

Before proving this result, let us make some observations.

- Good approximation $A \approx A_r$ is obtained iff $\sigma_{r+1} \ll \sigma_1$.
- Optimality holds for any unitarily invariant norm: that is, the norms in (3) can be replaced by e.g. the Frobenius norm. This is surprising, as the low-rank approximation problem $\min_{\text{rank}(B)\leq r} \|A - B\|$ does depend on the choice of the norm (and for many problems, including the least-squares problem in Section 6.5, the norm choice has a significant effect on the solution). The proof for this fact is nonexaminable, but if curious see [22] for a complete proof.
- A prominent application of low-rank approximation is PCA (principal component analysis) in statistics and data science.
- Many matrices have explicit or hidden low-rank structure (nonexaminable, but see e.g. [38]).

Proof: of Theorem 3.1:

1. Since $\text{rank}(B) \leq r$, we can write $B = B_1 B_2^T$ where B_1, B_2 have r columns.
2. It follows that B_2^T (and hence B) has a null space of dimension at least $n - r$. That is, there exists an orthonormal matrix $W \in \mathbb{C}^{n\times(n-r)}$ s.t. $BW = 0$. Then $\|A - B\|_2 \geq \|(A - B)W\|_2 = \|AW\|_2 = \|U\Sigma(V^T W)\|_2$. (Why does the first inequality hold?)
3. Now since W is $(n - r)$-dimensional, by Lemma 1.1 there is an intersection between $\text{Span}(W)$ and $\text{Span}([v_1, \ldots, v_{r+1}])$, the $(r + 1)$-dimensional subspace spanned by the leading $r + 1$ right singular vectors.

   We will use this type of argument again, so to be more precise: the matrix $[W, v_1, \ldots, v_{r+1}]$ is "fat" rectangular, so must have a null vector. That is, $[W, v_1, \ldots, v_{r+1}]\left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right] = 0$ has a nonzero solution x_1, x_2; then Wx_1 is such a vector[9]. We scale it so that it has unit norm $\|Wx_1\|_2 = \|[v_1, \ldots, v_{r+1}]x_2\|_2 = 1$, that is, $\|x_1\|_2 = \|x_2\|_2 = 1$.

[8]If $r \geq \min(m,n)$ then we can simply take $A_r = A$.

[9]Let us now resolve the cliffhanger in footnote 5. The claim is that any "fat" $m \times n$ ($m < n$) matrix M has a right null vector $y \neq 0$ such that $My = 0$. To prove this, use the full SVD $M = U\Sigma V^T$ to see that $Mv_n = 0$.

4. Note that $U\Sigma V^T W x_1 = U\Sigma V^T [v_1, \ldots, v_{r+1}] x_2$ and $V^T[v_1, \ldots, v_{r+1}] = \begin{bmatrix} I_{r+1} \\ 0 \end{bmatrix}$, so $\|U\Sigma V^T W x_1\|_2 = \|U \begin{bmatrix} \Sigma_{r+1} \\ 0 \end{bmatrix} x_2\|_2$, where Σ_{r+1} is the leading (top-left) $(r+1) \times (r+1)$ part of Σ. As U is orthogonal this is equal to $\|\Sigma_{r+1} y_1\|_2$, and $\|\Sigma_{r+1} y_1\|_2 \geq \sigma_{r+1}$ can be verified by direct calculation.

Finally, for the reverse direction, take $B = A_r$. $\square$

3.1 Low-rank approximation: image compression

A classical and visually pleasing illustration of low-rank approximation by the truncated SVD is image compression. (Admittedly this is slightly outdated—it is not the state-of-the-art way of compressing images.)

The idea is that greyscale images can be represented by a matrix where each entry indicates the intensity of a pixel. The matrix can then be compressed by finding a low-rank approximation[10], resulting in image compression. (Of course there are generalisations for color images.)

Below in Figure 1 we take the Oxford logo, represent it as a matrix and find its rank-r approximation, for varying r. (The matrix being a mere 500×500, its SVD is easy to compute in a fraction of a second; see Section 10.2 for how this is done.) We then reconstruct the image from the low-rank approximations to visualise them.

We see that as the rank is increased the image becomes finer and finer. At rank 50 it is fair to say the image looks almost identical to the original. The original matrix is 500×500, so we still achieve a significant amount of data compression in the matrix with $r = 50$.

4 Courant-Fischer minmax theorem

Continuing on SVD-related topics, we now discuss a very important and useful result with far-reaching ramifications: the Courant-Fischer (C-F) minimax characterisation.

Theorem 4.1 *The ith largest[11] eigenvalue λ_i of a symmetric matrix $A \in \mathbb{R}^{n\times n}$ is (below $x \neq 0$)*

$$\lambda_i(A) = \max_{\dim \mathcal{S}=i} \min_{x\in\mathcal{S}} \frac{x^T A x}{x^T x} \quad \left(= \min_{\dim \mathcal{S}=n-i+1} \max_{x\in\mathcal{S}} \frac{x^T A x}{x^T x} \right) \tag{5}$$

Analogously, for any rectangular $A \in \mathbb{C}^{m\times n} (m \geq n)$, we have

$$\sigma_i(A) = \max_{\dim \mathcal{S}=i} \min_{x\in\mathcal{S}} \frac{\|Ax\|_2}{\|x\|_2} \quad \left(= \min_{\dim \mathcal{S}=n-i+1} \max_{x\in\mathcal{S}} \frac{\|Ax\|_2}{\|x\|_2} \right). \tag{6}$$

[10] It is somewhat surprising that images are approximable by low-rank matrices. See `https://www.youtube.com/watch?v=9BYsNpTCZGg` for a nice explanation.

[11] exact analogues hold for the ith *smallest* eigenvalue and singular values.

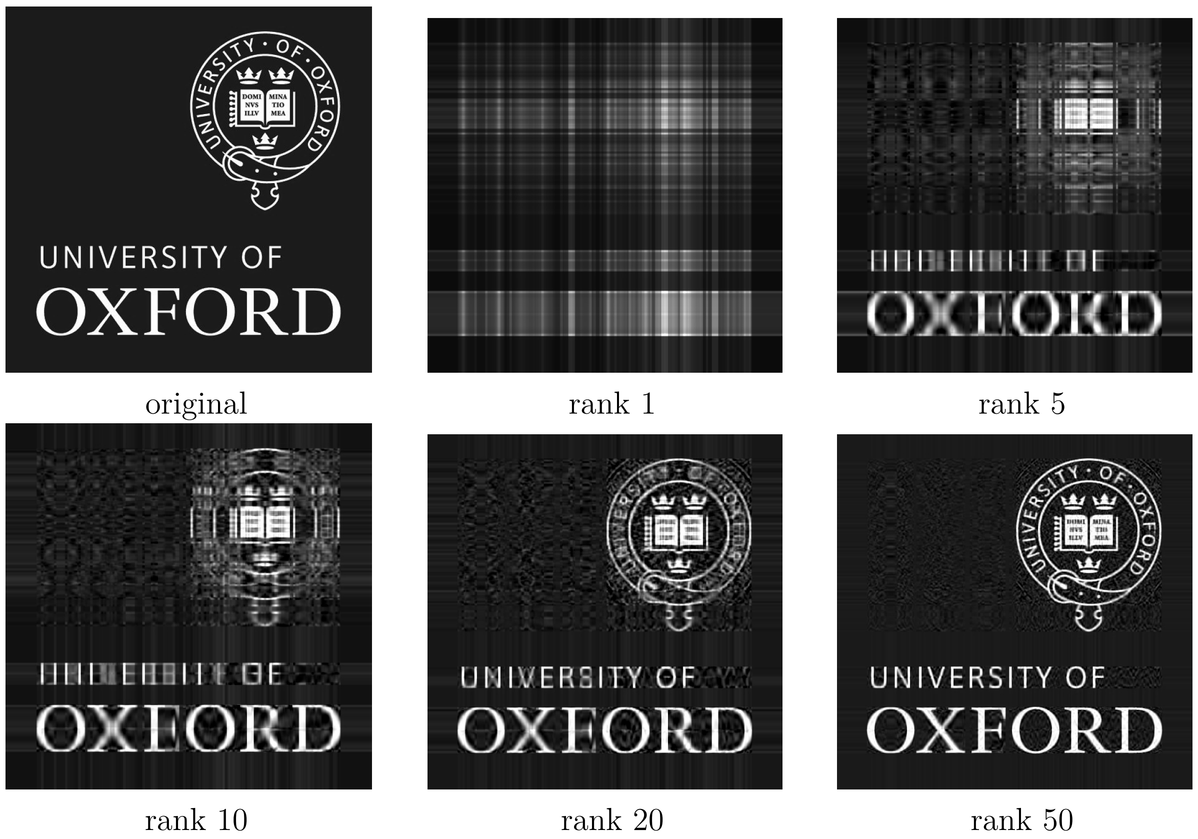


Figure 1: Image compression by low-rank approximation via the truncated SVD.

It would take some time to get a hang of what the statements mean. One helpful way to look at it is perhaps to note that inside the maximum in (6) the expression is $\min_{x\in\mathcal{S},\|x\|_2=1}\|Ax\|_2 = \min_{Q^TQ=I_i,\|y\|_2=1}\|AQy\|_2 = \sigma_{\min}(AQ) = \sigma_i(AQ)$, where $\text{span}(Q)=\mathcal{S}$. The C-F theorem says $\sigma_i(A)$ is equal to the maximum possible value of this over all subspaces $\mathcal{S}$ of dimension i.

Proof: We will prove (6). A proof for (5) is analogous and a recommended exercise.

1. Fix S and let $V_i = [v_i, \dots, v_n]$. We have $\dim(\mathcal{S})+\dim(\text{span}(V_i)) = i+(n-i+1) = n+1$, so $\exists$ intersection $w \in S \cap V_i$, $\|w\|_2 = 1$.
2. For this w, we have $\|Aw\|_2 = \|\text{diag}(\sigma_i,\dots,\sigma_n)(V_i^Tw)\|_2 \le \sigma_i$; thus $\sigma_i \ge \min_{x\in\mathcal{S}} \frac{\|Ax\|_2}{\|x\|_2}$.
3. For the reverse inequality, take $S = [v_1, \dots, v_i]$, for which $w = v_i$.

$\square$

Let's consider some special cases. When $i=1$, (6) gives $\sigma_1(A) = \max_{\dim\mathcal{S}=1}\min_{x\in\mathcal{S}}\frac{\|Ax\|_2}{\|x\|_2}$, but $\mathcal{S}=1$ means the minimisation is essentially over a single vector x; hence the expression reduces to $\sigma_1(A) = \max_x \frac{\|Ax\|_2}{\|x\|_2}$, as we've seen previously.

When $i=n$, (6) gives $\sigma_n(A) = \sigma_{\min}(A) = \max_{\dim\mathcal{S}=n}\min_{x\in\mathcal{S}}\frac{\|Ax\|_2}{\|x\|_2}$, but now the maximisation is with respect to $\dim\mathcal{S}=n$, which is always the whole space $\mathbb{R}^n$—so the expression reduces to $\sigma_{\min}(A) = \min_{x\in\mathcal{S}}\frac{\|Ax\|_2}{\|x\|_2} = \min_{\|x\|_2=1}\|Ax\|_2$. For other values of i, the expression (6) cannot be simplified.

4.1 Weyl's inequality

As an example of the many significant ramifications of the C-F theorem, we present *Weyl's theorem*[12] (or Weyl's inequality), an important perturbation result for singular values and eigenvalues of symmetric matrices.

Theorem 4.2 *Weyl's inequality*

- *For the singular values of any matrix A,*
  - $\sigma_i(A+E) \in \sigma_i(A) + [-\|E\|_2, \|E\|_2]$ *for all i.*
  - *Special case:* $\|A\|_2 - \|E\|_2 \le \|A+E\|_2 \le \|A\|_2 + \|E\|_2$
- *For eigenvalues of a symmetric matrix A, $\lambda_i(A+E) \in \lambda_i(A) + [-\|E\|_2, \|E\|_2]$ for all i.*

(Proof: exercise; almost a direct consequence of C-F.)

The upshot is that singular values and eigenvalues of symmetric matrices are insensitive to perturbation; a property known as being *well conditioned.*

This is important because this means a backward stable algorithm (see Section 7) computes these quantities with essentially full precision.

[12]Hermann Weyl was one of the prominent mathematicians of the 20th centry.

4.1.1 Eigenvalues of nonsymmetric matrices are sensitive to perturbation

It is worth remarking that eigenvalues of nonsymmetric matrices can be far from well conditioned! Consider for example the Jordan block $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. By perturbing this to $\begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix}$ one gets eigenvalues that are perturbed by $\sqrt{\epsilon}$, a magnification factor of $1/\sqrt{\epsilon} \gg 1$. More generally, consider the eigenvalues of a Jordan block and its perturbation

$$J = \begin{bmatrix} 1 & 1 & & \\ & 1 & \ddots & \\ & & \ddots & 1 \\ & & & 1 \end{bmatrix} \in \mathbb{R}^{n\times n}, \quad J + E = \begin{bmatrix} 1 & 1 & & \\ & 1 & \ddots & \\ & & \ddots & 1 \\ \epsilon & & & 1 \end{bmatrix}$$

$\lambda(J) = 1$ (n copies), but we have $|\lambda(J + E) - 1| \approx \epsilon^{1/n}$. For example when $n = 100$, an 10^{-100} perturbation in J would result in a 0.1 perturbation in all the eigenvalues!

(nonexaminable) This is pretty much the worst-case situation. In the generic case where the matrix is diagonalizable, $A = X\Lambda X^{-1}$, with an ϵ-perturbation the eigenvalues get perturbed by $O(c\epsilon)$, where the constant c depends on the so-called *condition number* $\kappa_2(X)$ of the eigenvector matrix X (see Section 7.2, and [9]).

4.2 More applications of C-F

(Somewhat optional) Let's explore more applications of C-F. A lot more can be proved; see [21] for many more results and examples along these lines.

Example 4.1

- $\sigma_i\left(\begin{bmatrix} A_1 \\ A_2 \end{bmatrix}\right) \geq \max(\sigma_i(A_1), \sigma_i(A_2))$

  Proof (sketch): LHS $= \max_{\dim \mathcal{S}=i} \min_{x\in\mathcal{S}, \|x\|_2=1} \left\| \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} x \right\|_2$, *and for any* x, $\left\| \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} x \right\|_2 \geq \max(\|A_1x\|_2, \|A_2x\|_2)$.

- $\sigma_i(\begin{bmatrix} A_1 & A_2 \end{bmatrix}) \geq \max(\sigma_i(A_1), \sigma_i(A_2))$

  Proof: LHS $= \max_{\dim \mathcal{S}=i} \min_{\left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right]\in\mathcal{S}, \left\|\left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right]\right\|_2=1} \left\| \begin{bmatrix} A_1 & A_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right\|_2$, *while*

  $\sigma_i(A_1) = \max_{\dim \mathcal{S}=i, \mathit{range}(\mathcal{S})\in \mathit{range}(\left[\begin{smallmatrix} I_n \\ 0 \end{smallmatrix}\right])} \min_{\left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right]\in\mathcal{S}, \left\|\left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right]\right\|_2=1} \left\| \begin{bmatrix} A_1 & A_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right\|_2$. *Since the latter imposes restrictions on* $\mathcal{S}$ *to take the maximum over, the former is at least as big.*

4.3 (Taking stock) Matrix decompositions you should know

Let us now take stock to review the matrix decompositions that we have covered, along with those that we will discuss next.

- SVD $A = U\Sigma V^T$
- Eigenvalue decomposition $A = X\Lambda X^{-1}$
  - Normal: X unitary $X^*X = I$
  - Symmetric: X unitary and Λ real
- Jordan decomposition: $A = XJX^{-1}$, $J = \text{diag}(\begin{bmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{bmatrix})$
- Schur decomposition $A = QTQ^*$: T upper triangular
- QR: Q orthonormal, U upper triangular
- LU: L lower triangular, U upper triangular

Red: Orthogonal decompositions, stable computation available

The Jordan decomposition is mathematically the ultimate form that any matrix can be reduced to by a similarity transformation, but numerically it is not very useful—one of the problems is that Jordan decompositions are very difficult to compute, as an arbitrarily small perturbation can change the eigenvalues and block sizes by a large amount (recall the discussion in Section 4.1.1).

5 Linear systems $Ax = b$

We now (finally) start our discussion on direct algorithms in NLA. The fundamental idea is to *factorisa* a matrix into a product of two (or more) simpler matrices. This foundational idea has been named one of the top 10 algorithms of the 20th centry [10].

The sophistication of the state-of-the-art implementation of direct methods is simply astonishing. For instance, a 100×100 dense linear system or eigenvalue problem can be solved in less than a milisecond on a standard laptop. Imagine solving it by hand!

We start with solving linear systems, unquestionably the most important problem in NLA (for applications). The classical way to solve it is via the (pivoted) LU decomposition. In some sense we needn't spend too much time here, as you must have seen much of the material (e.g. Gaussian elimination) before. However, the description of the LU decomposition given below will likely be different from the one that you have seen before. We have chosen a nonstandard description as it reveals its connection to low-rank approximation.

Let $A \in \mathbb{R}^{n\times n}$. Suppose we can decompose (or factorise) A into (here and below, $*$ denotes entries that are possibly nonzero).

$$A = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix} = \begin{bmatrix} {\color{red}*} & & & & \\ {\color{red}*} & {\color{red}*} & & & \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & & \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & {\color{red}*} & \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & {\color{red}*} & {\color{red}*} \end{bmatrix} \begin{bmatrix} {\color{blue}*} & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} \\ & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} \\ & & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} \\ & & & {\color{blue}*} & {\color{blue}*} \\ & & & & {\color{blue}*} \end{bmatrix} = {\color{red}L}{\color{blue}U}$$

Here L is lower triangular, and U is upper triangular. How can we find L, U?

To get started, consider rewriting A as

$$\begin{aligned} A &= \begin{bmatrix} * \\ * \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & * & * & * \end{bmatrix} + \begin{bmatrix} & & & & \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} * \\ * \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & * & * & * \end{bmatrix}}_{L_1U_1} + \underbrace{\begin{bmatrix} 0 \\ * \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} 0 & * & * & * & * \end{bmatrix}}_{L_2U_2} + \begin{bmatrix} & & & & \\ & & & & \\ & & * & * & * \\ & & * & * & * \\ & & * & * & * \end{bmatrix} = \cdots \end{aligned}$$

Namely, the first step finds L_1, U_1 such that:

$$A = \underbrace{\begin{bmatrix} * \\ * \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & * & * & * \end{bmatrix}}_{L_1U_1} + \begin{bmatrix} & & & & \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix}.$$

Specifying the elements, the algorithm can be described as (taking $a = A_{11}$)

$$\begin{aligned} \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & & & & \\ A_{31} & & & & \\ A_{41} & & & & \\ A_{51} & & & & \end{bmatrix} &= \begin{bmatrix} L_{11} & & & & \\ L_{21} & & & & \\ L_{31} & & & & \\ L_{41} & & & & \\ L_{51} & & & & \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} & U_{15} \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix} + \begin{bmatrix} & & & & \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 1 & & & & \\ A_{21}/a & & & & \\ A_{31}/a & & & & \\ A_{41}/a & & & & \\ A_{51}/a & & & & \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix}}_{=L_1U_1} + \begin{bmatrix} & & & & \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix} \end{aligned}$$

Here we've assumed $a \neq 0$; we'll discuss the case $a = 0$ later. Repeating the process gives

$$\begin{aligned} A &= \begin{bmatrix} * \\ * \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & * & * & * \end{bmatrix} + \begin{bmatrix} 0 \\ * \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} 0 & * & * & * & * \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} 0 & 0 & * & * & * \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ * \\ * \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & * & * \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ * \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & * \end{bmatrix} \\ &= \quad L_1U_1 \quad + \quad L_2U_2 \quad + \quad L_3U_3 \quad + \quad L_4U_4 \quad + \quad L_5U_5 \\ &= [L_1, L_2, \ldots, L_5] \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_5 \end{bmatrix} = \begin{bmatrix} {\color{red}*} & & & & \\ {\color{red}*} & {\color{red}*} & & & \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & & \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & {\color{red}*} & \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & {\color{red}*} & {\color{red}*} \end{bmatrix} \begin{bmatrix} {\color{blue}*} & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} \\ & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} \\ & & {\color{blue}*} & {\color{blue}*} & {\color{blue}*} \\ & & & {\color{blue}*} & {\color{blue}*} \\ & & & & {\color{blue}*} \end{bmatrix}, \end{aligned}$$

an LU decomposition as required.

Note the above expression for A; clearly the L_iU_i factors are rank-1 matrices; the LU decomposition can be thought of as writing A as a sum of (structured) rank-1 matrices.

5.1 Solving $Ax = b$ via LU

Having found an LU decomposition $A = LU$, one can efficiently solve an $n \times n$ linear system $Ax = b$: *First solve* $Ly = b$, *then* $Ux = y$. Then $b = Ly = LUx = Ax$.

- These are *triangular* linear systems, which are easy to solve and can be done in $O(n^2)$ flops.
- Triangular solve is always backward stable: e.g. $(L + \Delta L)\hat{y} = b$ (see Higham's book)

The computational cost is

- For LU: $\frac{2}{3}n^3$ flops (floating-point operations).
- Triangular solve is $O(n^2)$.

Note that once we have an LU factorisation we can solve another linear system with respect to the same matrix with only $O(n^2)$ additional operations.

5.2 Pivoting

Above we've assumed the diagonal element (pivot) $a \neq 0$—when $a = 0$ we are in trouble! In fact not every matrix has an LU decomposition. For example, there is no LU for $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. We need a remedy. In practice, a remedy is needed whenever a is small.

The idea is to *permute* the rows, so that the largest element of the (first active) column is brought to the pivot. This process is called *pivoting* (sometimes *partial pivoting*, to emphasize the difference from *complete pivoting* wherein both rows and columns are permuted[13]). This results in $PA = LU$, where P is a *permutation matrix*: orthogonal matrices with only 1 and 0s (every row/column has exactly one 1); applying P would reorder the rows (with PA) or columns (AP)).

Thus solving $Ax_i = b_i$ for $i = 1, \ldots, k$ requires $\frac{2}{3}n^3 + O(kn^2)$ operations instead of $O(kn^3)$.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & & & & \\ A_{31} & & & & \\ A_{41} & & & & \\ A_{51} & & & & \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ A_{21}/a & & & & \\ A_{31}/a & & & & \\ A_{41}/a & & & & \\ A_{51}/a & & & & \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix} + \begin{bmatrix} & & & & \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix}$$

When $a = 0$, remedy: pivot, permute rows such that the largest entry of first (active) column is at the top. $\Rightarrow PA = LU$, P: permutation matrix

- for $Ax = b$, solve $PAx = Pb \Leftrightarrow LUx = Pb$

[13]While we won't discuss complete pivoting further, you might be interested to know that when LU with complete pivoting is applied to a low-rank matrix (with rapidly decaying singular values), one tends to find a good low-rank approximation, almost as good as truncated SVD.

- cost still $\frac{2}{3}n^3 + O(n^2)$

In fact, one can show that any nonsingular matrix A has a pivoted LU decomposition. (proof: exercise). This means that any linear system that is computationally feasible can be solved by pivoted LU decomposition.

- Even with pivoting, unstable examples exist (Section 7), but almost always stable in practice and used everywhere.
- Stability here means $\hat{L}\hat{U} = PA + \Delta A$ with small $\|\Delta A\|$; see Section 7.

5.3 Cholesky factorisation for $A \succ 0$

If $A \succ 0$ (symmetric positive definite[14] (S)PD$\Leftrightarrow\lambda_i(A) > 0$ for all i), two simplifications happen in LU:

- We can take $U_i = L_i^T =: R_i$ by symmetry
- No pivot needed as long as A is PD

$$A = \underbrace{\begin{bmatrix} * \\ * \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} * & * & * & * & * \end{bmatrix}}_{R_1 R_1^T} + \underbrace{\begin{bmatrix} & & & & \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix}}_{\text{also PD}} \tag{7}$$

Notes:

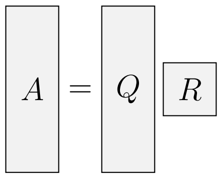
- $\frac{1}{3}n^3$ flops, half as many as LU
- diag(R) no longer 1's (clearly)
- A can be written as $A = R^T R$ for some $R \in \mathbb{R}^{n\times n}$ iff $A \succeq 0$ ($\lambda_i(A) \geq 0$)
- Indefinite case: when $A = A^*$ but A not PSD (i.e., negative eigenvalues are present), $\exists\ A = LDL^*$ where D diagonal (when $A \in \mathbb{R}^{n\times n}$, D can have 2×2 diagonal blocks), L has 1's on diagonal. This is often called the "LDLT factorisation".
- It's not easy at this point to see why the second (red) term above is also PD; one way to see this is via the uniqueness of the Cholesky factorisation; see next section.

Therefore, roughly speaking, symmetric linear systems can be solved with half the effort of a general non-symmetric system.

[14]Positive definite matrices are so important a class of matrices; also full of interesting mathematical properties, enough for a nice book to be written about it [5].

6 QR factorisation and least-squares problems

We've seen that the LU decomposition is a key step towards solving $Ax = b$. For an overdetemined problem (least-squares problems, the subject of Section 6.5), we will need the *QR factorisation.* For any $A \in \mathbb{R}^{m\times n}$, there exists a factorisation



where $Q \in \mathbb{R}^{m\times n}$ is orthonormal $Q^TQ = I_n$, and $R \in \mathbb{R}^{n\times n}$ is upper triangular.

- Many algorithms available: Gram-Schmidt, Householder QR, CholeskyQR, ...
- various applications: least-squares, orthogonalisation, computing SVD, manifold retraction...
- With Householder, pivoting $A = QRP$ not needed for numerical stability.
  - but pivoting gives rank-revealing QR (nonexaminable).

6.1 QR via Gram-Schmidt

No doubt you have seen the Gram-Schmidt (G-S) process. What you might not know is that when applied to the columns of a matrix A, it gives you a QR factorisation $A = QR$.

Gram-Schmidt: Given $A = [a_1, a_2, \dots, a_n] \in \mathbb{R}^{m\times n}$ (assume full-rank rank$(A) = n$), find orthonormal $[q_1, \dots, q_n]$ s.t. $\text{span}(q_1, \dots, q_n) = \text{span}(a_1, \dots, a_n)$

More precisely, the algorithm performs the following: $q_1 = \frac{a_1}{\|a_1\|}$, then $\tilde{q}_2 = a_2 - q_1q_1^Ta_2$, $q_2 = \frac{\tilde{q}_2}{\|\tilde{q}_2\|}$, (orthogonalise and normalise)
repeat for $j = 3, \dots, n$: $\tilde{q}_j = a_j - \sum_{i=1}^{j-1} q_iq_i^Ta_j$, $q_j = \frac{\tilde{q}_j}{\|\tilde{q}_j\|}$.

This gives a QR factorisation! To see this, let $r_{ij} = q_i^Ta_j$ $(i \neq j)$ and $r_{jj} = \|a_j - \sum_{i=1}^{j-1} r_{ij}q_i\|$,

$$q_1 = \frac{a_1}{r_{11}}$$
$$q_2 = \frac{a_2 - r_{12}q_1}{r_{22}}$$
$$q_j = \frac{a_j - \sum_{i=1}^{j-1} r_{ij}q_i}{r_{jj}}$$

which can be written equivalently as

$$\begin{aligned} a_1 &= r_{11}q_1 \\ a_2 &= r_{12}q_1 + r_{22}q_2 \\ a_j &= r_{1j}q_1 + r_{2j}q_2 + \cdots + r_{jj}q_j. \end{aligned}$$

This in turn is $\boxed{A} = \boxed{Q}\boxed{R}$, where $Q^TQ = I_n$, and R upper triangular.

- But this isn't the recommended way to compute the QR factorisation, as it's numerically unstable; see Section 7.7.1 and [19, Ch. 19,20].

6.2 Towards a stable QR factorisation: Householder reflectors

There is a beautiful alternative algorithm for computing the QR factorisation: *Householder QR factorisation*. In order to describe it, let us first introduce Householder reflectors. These are the class of matrices H that are symmetric, orthogonal and can be written as a rank one update of the identity

$$H = I - 2vv^T, \qquad \|v\| = 1$$

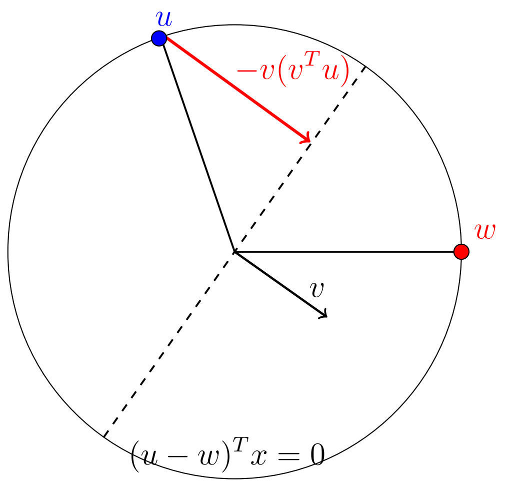
- H is orthogonal and symmetric: $H^TH = H^2 = I$. Its eigenvalues are 1 ($n-1$ copies) and -1 (1 copy).

- For any given $u, w \in \mathbb{R}^n$ s.t. $\|u\| = \|w\|$ and $u \neq w$, $H = I - 2vv^T$ with $v = \frac{w-u}{\|w-u\|}$ gives $Hu = w$ ($\Leftrightarrow u = Hw$, thus 'reflector')

- For a formal proof: Setting $v = \frac{u-w}{\|u-w\|_2}$ we have $Hu = (I - 2vv^T)u = u - 2v(v^Tu)$. This can be written as $\alpha u + \beta w$. Now

$$ww^Tu = (u-w)\frac{(u-w)^Tu}{\|u-w\|_2^2} = (u-w)\frac{1-w^Tu}{2-2u^Tw} = \frac{1}{2}(u-w),$$

so $Hu = u - 2v(v^Tu) = w$.



It follows that by choosing the vector v appropriately, one can perform a variety of operations to a given vector x. The primary example is the particular Householder reflector

$$H = I - 2vv^T, \qquad v = \frac{x - \|x\|e}{\|x - \|x\|e\|}, \qquad e = [1, 0, \ldots, 0]^T,$$

which satisfies $Hx = [\|x\|, 0, \dots, 0]^T$. That is, an arbitrary vector x is mapped by H to a multiple of $e = [1, 0, \dots, 0]^T$.

(I hope the picture above is helpful—the reflector reflects vectors about the hyperplane described by $v^T x = 0$.)

In summary, we have the useful result

Lemma 6.1 *For any $x \in \mathbb{R}^n$ not in the form $[\pm\|x\|, 0, \dots, 0]^T$, there exists a Householder reflector $H = I - 2vv^T$ where $\|v\| = 1$ such that $Hx = [\|x\|, 0, \dots, 0]^T$.*

6.3 Householder QR factorisation

Now we describe how to use the Householder reflectors in order to compute a QR factorisation of a given matrix $A \in \mathbb{R}^{m\times n}$.

The first step to obtain QR is to find H_1 s.t. $H_1 a_1 = \begin{bmatrix} \|a_1\| \\ 0 \\ \vdots \\ 0 \end{bmatrix}$,

and repeat to get $H_n \cdots H_2 H_1 A = R$ upper triangular, then $A = (H_1 \cdots H_{n-1} H_n) R = QR$

Here is a pictorial illustration: start with

$$A = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}.$$

We apply a sequence of Householder reflectors

$$H_1 A = (I - 2v_1 v_1^T)A = \begin{bmatrix} * & * & * & * \\ & * & * & * \\ & * & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix}, \qquad H_2 H_1 A = (I - 2v_2 v_2^T) H_1 A = \begin{bmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & * & * \\ & & * & * \end{bmatrix},$$

$$H_3 H_2 H_1 A = \begin{bmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & & * \\ & & & * \end{bmatrix}, \qquad H_n \cdots H_3 H_2 H_1 A = \begin{bmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & & * \\ & & & \end{bmatrix}.$$

Note the zero pattern $v_k = [\underbrace{0, 0, \dots, 0}_{k-1 \text{ 0's}}, *, *, \dots, *]^T$. We have

$$H_n \cdots H_2 H_1 A = \begin{bmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & & * \\ & & & \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

To obtain a QR factorisation of A we simply invert the Householder reflectors, noting that they are both orthogonal and symmetric; therefore the inverse is itself. This yields

$\Leftrightarrow A = (H_1^T \cdots H_{n-1}^T H_n^T) \begin{bmatrix} R \\ 0 \end{bmatrix} =: Q_F \begin{bmatrix} R \\ 0 \end{bmatrix}$; which is a **full** QR, wherein Q_F is square orthogonal.

Moreover, writing $Q_F = [Q\ Q_\perp]$ where $Q \in \mathbb{R}^{m\times n}$ is orthonormal, we also have $A = QR$ (**'thin'** QR or just QR); this is more economical especially when A is tall-skinny. In the majority of cases in computational mathematics, this is the object that we wish to compute).

We note some properties of Householder QR.

- Cost $\frac{4}{3}n^3$ flops with Householder-QR (twice that of LU when $m = n$; if $m > n$, $2mn^2 - \frac{2}{3}n^3$). See Golub-Van Loan Ch. 5 for details; it is important to note that for a vector $a \in \mathbb{R}^m$, $Ha = (I - 2vv^T)a = a - 2v(v^Ta)$ can be computed in $O(m)$ operations, not $O(m^2)$.
- Unconditionally backward stable: the computed version satisfies $\hat{Q}\hat{R} = A + \Delta A$, $\|\hat{Q}^T\hat{Q} - I\|_2 = \epsilon$ (Section 7).
- The algorithm gives a constructive proof for the existence of a full QR $A = QR$. It also gives, for example, a proof of the existence of the orthogonal complement of the column space of an orthonormal matrix U.
- To solve $Ax = b$, solve $Rx = Q^Tb$ via triangle solve.
  $\to$ Excellent method, but twice slower than LU (so it is rarely used)
- The process is aptly called orthogonal triangularisation. (By contrast, Gram-Schmidt and CholeskyQR[15] are triangular orthogonalisation).

6.4 Givens rotations

Householder QR is an excellent method for computing the QR factorisation of a general matrix, and it is widely used in practice. However, each Householder reflector acts globally– it affects all the entries of the (active part of) the matrix. For structured matrices—such as sparse matrices—sometimes there is a better tool to reduce the matrix to triangular form (and other forms) by working more locally. Givens rotations give a convenient tool for this. They are matrices of the form

$$G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \quad c^2 + s^2 = 1.$$

Designed to 'zero' one element at a time. For example to compute the QR factorisation for an upper Hessenberg matrix, one can perform

$$A = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \end{bmatrix}, \quad G_1A = \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \end{bmatrix}, \quad G_2G_1A = \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & * & * & * \\ & & & * & * \end{bmatrix},$$

[15]This algorithm does the following: $A^TA = R^TR$ (Cholesky), then $Q = AR^{-1}$. As stated it's a very fast but unstable algorithm.

$$G_3G_2G_1A = \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \\ & & & * & * \end{bmatrix}, \; G_4G_3G_2G_1A = \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \\ & & & & * \end{bmatrix} =: R.$$

This means $A = G_1^TG_2^TG_3^TG_4^TR$ is the QR factorisation. (note that Givens rotations are orthogonal but not symmetric—so its inverse is G^T, not G).

- G acts locally on two rows (when left-multiplied; two columns if right-multiplied)
- Non-neighboring rows/cols allowed. For example, a rotation acting on the i, jth columns would have c, s values in the $(i, i), (i, j), (j, i), (j, j)$ entries. Visually,

$$G_{i,j} = \begin{bmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & \cos(\theta) & & & & \sin(\theta) & & & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & 1 & & & & \\ & & & -\sin(\theta) & & & & \cos(\theta) & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{bmatrix}$$

6.5 Least-squares problems via QR

So far we have discussed linear systems wherein the coefficient matrix is square. However, in many situations in data science and beyond, there is a good reason to over-sample to obtain a robust solution (for instance, in the presence of noise in measurements). For example, when there is massive data and we would like to fit the data with a simple model, we will have many more equations than the degrees of freedom. This leads to the so-called *least-squares problem* which we will discuss here.

Given $A \in \mathbb{R}^{m\times n}, m \geq n$ and $b \in \mathbb{R}^m$, a least-squares problem seeks to find $x \in \mathbb{R}^n$ such that the residual is minimised:

$$\min_x \left\| \boxed{A}\,\boxed{x} - \boxed{b} \right\|_2 \tag{8}$$

- 'Overdetermined' linear system; attaining equality $Ax = b$ is usually impossible
- Thus the goal is to try minimise the *residual* $\|Ax-b\|$; usually $\|Ax-b\|_2$ but sometimes e.g. $\|Ax-b\|_1$ is of interest. Here we focus on $\|Ax-b\|_2$.

- Throughout we assume full rank condition rank$(A) = n$; this makes the solution unique and is generically satisfied. If not, the problem will have infinitely many minimisers (and a standard practice is to look for the minimum-norm solution).

Here is how we solve the least-squares problem (8).

Theorem 6.1 *Let $A \in \mathbb{R}^{m\times n}, m > n$ and $b \in \mathbb{R}^m$, with rank$(A) = n$. The least-squares problem $\min_x \|Ax - b\|_2$ has solution given by $x = R^{-1}Q^T b$, where $A = QR$ is the (thin) QR factorisation.*

Proof: Let $A = [Q\ Q_\perp]\left[\begin{smallmatrix} R \\ 0 \end{smallmatrix}\right] = Q_F\left[\begin{smallmatrix} R \\ 0 \end{smallmatrix}\right]$ be 'full' QR factorisation. Then

$$\|Ax - b\|_2 = \|Q_F^T(Ax - b)\|_2 = \left\| \begin{bmatrix} R \\ 0 \end{bmatrix} x - \begin{bmatrix} Q^T b \\ Q_\perp^T b \end{bmatrix} \right\|_2$$

so $x = R^{-1}Q^T b$ is the solution, which is seen to be unique as R is nonsingular (as A is full rank). $\square$

This also gives an algorithm (which is essentially the workhorse algorithm used in practice):

1. Compute **thin** QR factorisation $A = QR$ (using Householder QR)
2. Solve linear system $Rx = Q^T b$.

- This process is backward stable. That is, the computed $\hat{x}$ solution for $\min_x \|(A + \Delta A)x + (b + \Delta b)\|_2$ (see Higham's book Ch.20)
- Unlike square system $Ax = b$, one really needs QR: LU won't do the job at all.

One might wonder why we chose the 2-norm in the least-squares formulation (8). Unlike for low-rank approximation (where the truncated SVD is a solution for any unitarily invariant norm[16]) the choice of the norm does matter and affects the properties of the solution x significantly. For example, an increasingly popular choice of norm is the 1-norm, which tends to promote sparsity in the quantity to be minimised. In particular, if we simply replace the 2-norm with the 1-norm, the solution tends to give a residual that is sparse. (nonexaminable)

[16]Just to be clear, if one uses a norm that is not unitarily invariant (e.g. 1-norm), the truncated SVD may cease to be a solution for the low-rank approximation problem.

6.6 QR-based algorithm for linear systems

It is straightforward to see that the exact same algorithm can be applied for solving square linear systems. Is this algorithm good? Absolutely! It turns out that it is even better than the LU-based method in that backward stability can be guaranteed (which isn't the case with pivoted LU; backward stability is discussed in the next section). However, it is unfortunately twice expensive; which is the reason LU is used in the vast majority of cases for solving linear systems.

Another very stable algorithm is to compute the SVD $A = U\Sigma V^T$ and take $x = V\Sigma^{-1}U^T b$. This is even more expensive than via QR (by $\approx$ x10).

6.7 Solution of least-squares via normal equation

There is another way to solve the least-squares problem, by the so-called normal equation. We've seen that

$$\min_x \|Ax - b\|_2, \qquad A \in \mathbb{R}^{m\times n}, m \geq n$$

$x = R^{-1}Q^T b$ is the solution $\Leftrightarrow$ x solution for $n \times n$ **normal equation**

$$(A^T A)x = A^T b$$

- $A^T A \succeq 0$ (always) and $A^T A \succ 0$ if $\text{rank}(A) = n$; then PD linear system; use Cholesky to solve.
- This is fast! but NOT backward stable; $\kappa_2(A^T A) = (\kappa_2(A))^2$ where $\kappa_2(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$ **condition number** (next topic)

In fact, more generally, given a linear least-squares approximation problem $\min_{p\in\mathcal{P}} \|f - p\|$ in an inner-product space (of which (8) is a particular example; other examples include polynomial approximation of a function with the inner product e.g. $\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)dx$) the solution is characterised by the property that the residual is orthogonal to the subspace from which a solution is sought, that is, $\langle q, f - p\rangle = 0$ for all $q \in \mathcal{P}$. To see this, consider the problem of approximating f with p in a subspace $\mathcal{P}$. Let $p_* \in \mathcal{P}$ be such that the residual $f - p_+$ is orthogonal to any element in $\mathcal{P}$. Then for any $q \in \mathcal{P}$, we have $\|f - (p_* + q)\|^2 = \|f - p_*\|^2 - 2\langle f - p_*, q\rangle + \|q\|^2 = \|f - p_*\|^2 + \|q\|^2 \geq \|f - p_*\|^2$, proving p_* is a minimiser (it is actually unique).

Since we mentioned Cholesky, let us now revisit (7) and show why the second term there must be PSD. A PD matrix has an eigenvalue decomposition $A = VD^2V^T = (VDV^T)^2 = (VDV^T)^T(VDV^T)$. Now let $VDV^T = QR$ be the QR factorisation. Then $(VDV^T)^T(VDV^T) = R^T R$ (this establishes the existence of Cholesky). But now the 0-structure in (7) means the first term must be rr^T where r^T is the first row of R, and hence the second term must be $R_2^T R_2$, which is PSD. Here $R^T = [r\ R_2^T]$.

6.8 Application of least-squares: regression/function approximation

To illustrate the usefulness of least-squares problems as compared with linear systems here let's consider a function approximation problem.

Given function $f : [-1, 1] \to \mathbb{R}$,

Consider approximating via polynomial $f(x) \approx p(x) = \sum_{i=0} c_i x^i$.

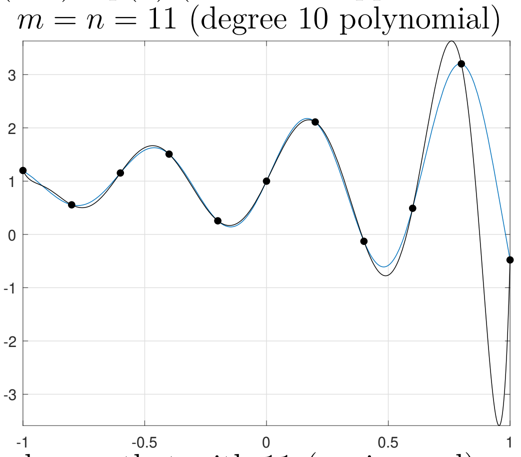
Very common technique: **Regression**: this is a very widely applicable problem in statistics and data science.

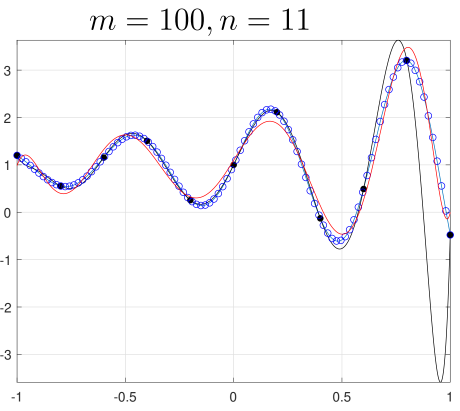
1. Sample f at points $\{z_i\}_{i=1}^m$, and
2. Find coefficients c defined by Vandermonde system $Ac \approx f$,

$$\begin{bmatrix} 1 & z_1 & \cdots & z_1^n \\ 1 & z_2 & \cdots & z_2^n \\ \vdots & \vdots & & \vdots \\ 1 & z_m & \cdots & z_m^n \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix} \approx \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_m) \end{bmatrix}.$$

- Numerous applications, e.g. in statistics, numerical analysis, approximation theory, data analysis!

Illustration We illustrate this with an example where we approximate the function $f(x) = 1 + \sin(10x)\exp(x)$ (which we suppose we don't know but we can sample it).





We observe that with 11 (equispaced) sample points, the degree-10 polynomial is deviating from the 'true function' quite a bit. With many more sample points the situation significantly improves. This is not a cherry-picked example but a phenomenon that can be mathematically proved; look for "Lebesgue constant" if interested (nonexaminable).

7 Numerical stability

An important aspect that is very often overlooked in numerical computing is *numerical stability*. Very roughly, it concerns the quality of a solution obtained by a numerical algorithm, given that computation on computers is done not exactly but with rounding errors. So far we have mentioned stability here and there in passing, but in this section it will be our focus.

Let us first look at an example where roundoff errors play a visible role to affect the computed solution of a linear system.

The situation is complicated. For example, let $A = U\Sigma V^T$, where $U = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, $\Sigma = \begin{bmatrix} 1 & \\ & 10^{-15} \end{bmatrix}$, $V = I$, and let $b = A\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. That is, we are solving a linear system whose solution is $x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

If we solve this in MATLAB using $\texttt{x} = \texttt{A}\backslash\texttt{b}$, the output is $\begin{bmatrix} 1.0000 \\ 0.94206 \end{bmatrix}$. Quite different from the exact solution! Did something go wrong? Did MATLAB or the algorithm fail? The answer is NO, MATLAB and the algorithm (LU) performed just fine. This is a ramification of ill-conditioning, not instability. Make sure that after covering this section, you will be able to explain what happened in the above example.

7.1 Floating-point arithmetic

The IEEE (double precision) floating point arithmetic is by far the most commonly used model of computation adopted in computation, and we will assume its use here. We will not get into its details as it becomes too computer scientific, rather than mathematical. Just to give a very sketchy introduction

- Computers store number in base 2 with finite/fixed memory (bits)
- Numbers not exactly representable with finite bits in base 2, including irrational numbers, are stored inexactly (rounded), e.g. $1/3 \approx 0.333...$ The unit with which rounding takes place is the *machine precision*, often denoted by ϵ (or u for unit roundoff). In the most standard setting of IEEE double-precision arithmetic, $u \approx 10^{-16}$.
- Whenever calculations (addition, subtraction, multiplication, division) are performed, the result is rounded to the nearest floating-point number (rounding error); this is where numerical errors really creep in
- Thus the accuracy of the final error is nontrivial; in pathological cases, it is rubbish!

To get an idea of how things can go wrong and how serious the final error can be, here are two examples with MATLAB:

- $((\texttt{sqrt}(2))^2 - 2) * \texttt{1e15} = \texttt{0.4441}$ (should be 0..)
- $\sum_{n=1}^{\infty} \frac{1}{n} \approx 30$ (should be ∞..)

For matrices, there are much more nontrivial and surprising phenomena than these. An important (but not main) part of numerical analysis/NLA is to study the effect of rounding errors. This topic can easily span a whole course. By far the best reference on this topic is Higham's book [19].

In this section we denote by $fl(X)$ a computed version of X (fl stands for floating point). For basic operations such as addition and multiplication, one has $fl(x+y) = x+y+c\epsilon$ where $|c| \leq \max(|x|,|y|)$ and $fl(xy) = xy + c\epsilon$ where $|c| \leq \max(|xy|)$.

7.2 Conditioning and stability

It is important to solidify the definition of **stability** (which is a property of an algorithm) and **conditioning** (which concerns the sensitivity of a problem and has nothing to do with the algorithm used to solve it).

- Conditioning is the sensitivity of a problem (e.g. of finding $y = f(x)$ given x) to perturbation in inputs, i.e., how large $\kappa := \sup_{\delta x} \|f(x+\delta x) - f(x)\|/\|\delta x\|$ is in the limit $\delta x \to 0$. (Very informally, one can think of conditioning as the largest directional derivative).

  (this is the *absolute* condition number; equally important is the *relative* condition number $\kappa_r := \sup_{\delta x} \frac{\|f(x+\delta x)-f(x)\|}{\|f(x)\|} \Big/ \frac{\|\delta x\|}{\|x\|}$)

- (Backward) Stability is a property of an algorithm, which describes if the computed solution $\hat{y}$ is a 'good' solution, in that it is an exact solution of a nearby input, that is, $\hat{y} = f(x+\Delta x)$ for a small Δx: if $\|\Delta x\|$ can be shown to be small, $\hat{y}$ is a backward stable solution. If an algorithm is guarantee to output a backward stable solution, that algorithm is called backward stable. Throughout this section, Δ denotes a quantity that is small relative to the quantity to follow: $\|\Delta X\|/\|X\| = O(\epsilon)$, where ϵ is the machine precision.

To repeat, conditioning is intrinsic in the problem. Stability is a property of an algorithm. Thus we will never say "this problem is backward stable" or "this algorithm is ill-conditioned". We can say "this problem is ill/well-conditioned", or "this algorithm is/isn't (backward) stable". If a problem is ill-conditioned $\kappa \gg 1$, and the computed solution is no very accurate, then one should blame the problem, not the algorithm. In such cases, a backward stable solution (see below) is usually still considered a good solution.

Notation/convention in this section: $\hat{x}$ denotes a computed approximation to x (e.g. of $x = A^{-1}b$). ϵ denotes a small term $O(u)$, on the order of unit roundoff/working precision; so we write e.g. $u, 10u, (m+n)u, mnu$ all as ϵ. (In other words, here we assume $m, n \ll u^{-1}$.)

- Consequently (in this lecture/discussion) the norm choice does not matter for the discussion.

7.3 Numerical stability; backward stability

Let us dwell more on (backward) stability, because it is really at the heart of the discussion on numerical stability. The word *backward* is key here and it probably differs from the natural notion of stability that you might first think of.

For a computational task $Y = f(X)$ (given input X, compute Y) and computed approximant $\hat{Y}$,

- Ideally, error $\|Y - \hat{Y}\|/\|Y\| = \epsilon$: but this is seldom true, and often impossible!
  (u: unit roundoff, $\approx 10^{-16}$ in standard double precision)

- Good alg has Backward stability $\hat{Y} = f(X + \Delta X)$, $\frac{\|\Delta X\|}{\|X\|} = \epsilon$ "exact solution of a slightly wrong input".

- Justification: The input (matrix) is usually inexact anyway, as storing it on a computer as a floating-point object already incurs rounding errors! Consequently, $f(X + \Delta X)$ is just as good at $f(X)$ at approximating $f(X_*)$ where $\|\Delta X\| = O(\|X - X_*\|)$.

  We shall 'settle with' such solution, though it may not mean $\hat{Y} - Y$ is small.

- Forward stability[17] $\|Y - \hat{Y}\|/\|Y\| = O(\kappa(f)u)$ "error is as small as backward stable alg".

- Another important notion: mixed forward-backward stability: "The computed output is a slightly perturbed solution for a slightly perturbed problem".

7.4 Matrix condition number

The best way to illustrate conditioning is to look at the conditioning of linear systems. In fact it leads to the following definition, which arises so frequently in NLA that it merits its own name: the *condition number* of a matrix.

Definition 7.1 *The matrix condition number is defined by*

$$\kappa_2(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} (\geq 1).$$

That is, $\kappa_2(A) = \frac{\sigma_1(A)}{\sigma_n(A)}$ *for* $A \in \mathbb{R}^{m\times n}, \quad m \geq n.$

[17]The definition here follows Higham's book [19]. The phrase is sometimes used to mean small error; However, as hopefully you will be convinced after this section, it is very often impossible to get a solution of full accuracy if the original problem was ill-conditioned. The notion of backward stability that we employ here (and in much of numerical analysis) is therefore much more realistic.

Let's see how this arises:

Theorem 7.1 *Consider a backward stable solution for $Ax = b$, s.t. $(A + \Delta A)\hat{x} = b$ with $\|\Delta A\| \leq \epsilon \|A\|$ and $\kappa_2(A) \ll \epsilon^{-1}$ (so $\|A^{-1}\Delta A\| \ll 1$). Then we have*

$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \epsilon \kappa_2(A) + O(\epsilon^2).$$

Proof: By Neumann series

$$(A + \Delta A)^{-1} = (A(I + A^{-1}\Delta A))^{-1} = (I - A^{-1}\Delta A + O(\|A^{-1}\Delta A\|^2))A^{-1}$$

So $\hat{x} = (A+\Delta A)^{-1}b = A^{-1}b - A^{-1}\Delta A A^{-1}b + O(\|A^{-1}\Delta A\|^2) = x - A^{-1}\Delta A x + O(\|A^{-1}\Delta A\|^2)$, Hence

$$\|x - \hat{x}\| \lesssim \|A^{-1}\Delta A x\| \leq \|A^{-1}\| \|\Delta A\| \|x\| \leq \epsilon \|A\| \|A^{-1}\| \|x\| = \epsilon \kappa_2(A) \|x\|.$$

□

In other words, even with a backward stable solution, one would only have $O(\kappa_2(A)\epsilon)$ relative accuracy in the solution. If $\kappa_2(A)\epsilon > 1$, the solution may be rubbish! But the NLA view is that's not the fault of the algorithm, the blame is on the problem being so ill-conditioned.

7.4.1 Backward stable+well conditioned=accurate solution

We've seen that backward stability does not necessarily imply the solution is accurate. There is a happier side of this argument and useful rule of thumb that can be used to estimate the accuracy of a computed solution using backward stability and conditioning.

Suppose

- $Y = f(X)$ is computed backward stably i.e., $\hat{Y} = f(X + \Delta X)$, $\|\Delta X\| = \epsilon$.
- Conditioning $\|f(X) - f(X + \Delta X)\| \lesssim \kappa \|\Delta X\|$.

Then (this is the absolute version, relative version possible)

$$\|\hat{Y} - Y\| \lesssim \kappa \epsilon.$$

'proof':

$$\|\hat{Y} - Y\| = \|f(X + \Delta X) - f(X)\| \lesssim \kappa \|\Delta X\| = \kappa \epsilon.$$

Here is how to interpret the result: If the problem is well-conditioned $\kappa = O(1)$, this immediately implies good accuracy of the solution! But otherwise the solution might have poor accuracy—but it is still the exact solution of a nearby problem. This is often as good as one can possibly hope for.

The reason this is only a rule of thumb and not exactly rigorous is that conditioning only examines the asymptotic behavior, where the perturbation is infinitesimally small. Nonetheless it often gives an indicative estimate for the error and sometimes we can get rigorous bounds if we know more about the problem. Important examples include the following:

- Well-conditioned linear system $Ax = b$, $\kappa_2(A) \approx 1$.
- Eigenvalues of symmetric matrices (via Weyl's bound $\lambda_i(A+E) \in \lambda_i(A)+[-\|E\|_2, \|E\|_2]$).
- Singular values of any matrix $\sigma_i(A+E) \in \sigma_i(A) + [-\|E\|_2, \|E\|_2]$.

Indeed, these problems are well-conditioned, so can be solved with extremely high accuracy, essentially to working precision $O(u)$ (times a small factor, usually bounded by something like $\sqrt{n}$).

Note: eigvecs/singvecs can be highly ill-conditioned even for the simplest problems. Again, think of the identity. (Those curious are invited to look up the Davis-Kahan $\sin\theta$ theorem [8].)

7.5 Stability of triangular systems

7.5.1 Backward stability of triangular systems

While we will not be able to discuss in detail which NLA algorithm is backward stable etc, we will pick a few important examples.

One fact is worth knowing (a proof is omitted; see [36] or [19]): triangular linear systems can be solved in a backward stable manner. This fact is important as these arise naturally in the solution of linear systems: $Ax = b$ is solved via $Ly = b$, $Ux = y$ (triangular systems), as we've seen in Section 5.

Let R denote a matrix that is (upper or lower) triangular. The computed solution $\hat{x}$ for a (upper/lower) triangular linear system $Rx = b$ solved via back/forward substitution is backward stable, i.e., it satisfies

$$(R + \Delta R)\hat{x} = b, \qquad \|\Delta R\| = O(\epsilon\|R\|).$$

Proof: Trefethen-Bau or Higham (nonexaminable but interesting).

Notes:

- The backward error can be bounded componentwise.
- Using the previous rule-of-thumb, this means $\|\hat{x} - x\|/\|x\| \lesssim \epsilon\kappa_2(R)$.
  - (unavoidably) poor worst-case (and attainable) bound when ill-conditioned
  - often better with triangular systems; the behavior of triangular linear systems keep surprising experts!

7.5.2 (In)stability of $Ax = b$ via LU with pivots

We have discussed how to solve a linear system using the LU decomposition. An obvious question given the context is: is it backward stable? This question has a fascinating answer, and transpires to touch on one of the biggest open problems in the field.

Fact (proof nonexaminable): Computed $\hat{L}\hat{U}$ satisfies $\frac{\|\hat{L}\hat{U}-A\|}{\|L\|\|U\|} = \epsilon$.

(note: not $\frac{\|\hat{L}\hat{U}-A\|}{\|A\|} = \epsilon$)

- By stability of triangular systems $(L+\Delta L)(U+\Delta U)\hat{x} = b$. Now if $\|L\|\|U\| = O(\|A\|)$, then it follows that $\Rightarrow \hat{x}$ backward stable solution (exercise).

Question: Does $LU = A + \Delta A$ or $LU = PA + \Delta A$ with $\|\Delta A\| = \epsilon\|A\|$ (i.e., $\|L\|\|U\| = O(\|A\|)$) hold?

Without pivot ($P = I$): $\|L\|\|U\| \gg \|A\|$ unboundedly (e.g. $\left[\begin{smallmatrix} \epsilon & 1 \\ 1 & 1 \end{smallmatrix}\right]$) unstable.
With pivots:

- Worst-case: $\|L\|\|U\| \gg \|A\|$ grows exponentially with n, unstable.
  - growth governed by that of $\|L\|\|U\|/\|A\| \Rightarrow \|U\|/\|A\|$.
- In practice (average case): perfectly stable.
  - Hence this is how $Ax = b$ is solved, despite alternatives with guaranteed stability exist (but slower; e.g. via SVD, or QR (next)).

Resolution/explanation: among biggest open problems in numerical linear algebra!

7.5.3 Backward stability of Cholesky for $A \succ 0$

We've seen that symmetric positive definite matrices can be solved with half the effort using the Cholesky factorisation. As a bonus, in this case it turns out that stability can be guaranteed.

The key fact is that the Cholesky factorisation $A = R^T R$ for $A \succ 0$

- succeeds without pivot (the active matrix is always positive definite).
- R never contains entries $> \sqrt{\|A\|}$.

It follows that computing the Cholesky factorisation (assuming $A \succ 0$) is backward stable! Hence positive definite linear system $Ax = b$ can always be solved in a backward stable manner via Cholesky.

7.6 Matrix multiplication is not backward stable

Here is perhaps a shock—matrix matrix multiplication, one of the most basic operations, is in general not backward stable.

Let's start with the basics.

- Vector-vector multiplication is backward stable: $fl(y^T x) = (y + \Delta y)(x + \Delta x)$; in fact $fl(y^T x) = (y + \Delta y)x$. (direct proof is possible)
- It immediately follows that matrix-vector multiplication is also backward stable: $fl(Ax) = (A + \Delta A)x$.

- But it is not true to say matrix-matrix multiplication is backward stable; which would require $fl(AB)$ to be equal to $(A+\Delta A)(B+\Delta B)$. This may not be satisfied!

In general, what we can prove is a bound for the forward error $\|fl(AB)-AB\| \leq \epsilon\|A\|\|B\|$, so $\|fl(AB)-AB\|/\|AB\| \leq \epsilon\min(\kappa_2(A),\kappa_2(B))$ (proof: problem sheet).

This is great news when A or B is orthogonal (or more generally square and well-conditioned): say if $A=Q$ is orthogonal, then we have

$$\|fl(QB)-QB\| \leq \epsilon\|B\|,$$

so it follows that $fl(QB)=QB+\epsilon\|B\|$, hence defining $\Delta B = Q^T\epsilon\|B\|$ we have $fl(QB) = Q(B+\Delta B)$, that is, **orthogonal multiplication is backward stable** (this argument proves this for left multiplication; orthogonal right-multiplication is entirely analogous).

One of the reasons backward stability of matrix-matrix multiplication fails to hold is that there are not enough parameters in the backward error to account for the computational error incurred in the matrix multiplication. Each matrix-vector product is backward stable; but we cannot concentrate the backward errors into a single term without potentially increasing the backward error's norm.

On the other hand, we have seen that a linear system can be solved in a backward stable fashion (by QR if necessary; in practice LU with pivoting suffices). This means the inverse can be applied to a vector in a backward stable fashion: The computed $\hat{x}$ satisfies $(A+\Delta A)^{-1}\hat{x}=b$.

One might wonder, can we not solve n linear systems in order to get a backward stable inverse? However, if one solves n linear systems with $b=e_1,e_2,\dots,e_n$, the solutions will satisfy $(A+\Delta_i A)^{-1}\hat{x}_i = e_i$, where $\Delta_i A$ differs for each i. There is no global ΔA that is $O(\epsilon)$ such that $(A+\Delta A)^{-1}[\hat{x}_1,\dots,\hat{x}_n]=I$.

Orthogonality matters for stability As mentioned above, a happy and important exception is with orthogonal matrices Q (or more generally with well-conditioned matrices):

$$\frac{\|fl(QA)-QA\|}{\|QA\|} \leq \epsilon, \qquad \frac{\|fl(AQ)-AQ\|}{\|AQ\|} \leq \epsilon.$$

They are also backward stable:

$$\begin{aligned} fl(QA) = QA+\epsilon \quad &\Leftrightarrow \quad fl(QA) = Q(A+\Delta A).\\ fl(AQ) = AQ+\epsilon \quad &\Leftrightarrow \quad fl(AQ) = (A+\Delta A)Q.\end{aligned}$$

Hence algorithms involving ill-conditioned matrices are unstable (e.g. eigenvalue decomposition of non-normal matrices, Jordan form, etc), whereas those based on orthogonal matrices are stable. These include

- Householder QR factorisation, QR-based linear system (next subsection).
- **QR algorithm** for $Ax=\lambda x$.
- **Golub-Kahan** algorithm for $A=U\Sigma V^T$.

- **QZ algorithm** for $Ax = \lambda Bx$.

Section 8 onwards treats our second big topic, eigenvalue problems. This includes discussing algorithms shown above in boldface.

7.7 Stability of Householder QR

Householder QR has excellent numerical stability, basically because it's based on orthogonal transformations. With Householder QR, the computed $\hat{Q}, \hat{R}$ satisfy

$$\|\hat{Q}^T\hat{Q} - I\| = O(\epsilon), \quad \|A - \hat{Q}\hat{R}\| = O(\epsilon\|A\|),$$

and (of course) R is upper triangular.

Rough proof: Essentially the key idea is that multiplying by an orthogonal matrix is very stable and Householder QR is based on a sequence of multiplications by orthogonal matrices. To give a bit more detail,

- Each reflector satisfies $fl(H_iA) = H_iA + \epsilon_i\|A\|$.
- Hence $(\hat{R} =)fl(H_n \cdots H_1A) = H_n \cdots H_1A + \epsilon\|A\|$.
- $fl(H_n \cdots H_1) =: \hat{Q}^T = H_n \cdots H_1 + \epsilon$.
- Thus $\hat{Q}\hat{R} = A + \epsilon\|A\|$.

Notes:

- This doesn't mean $\|\hat{Q}-Q\|, \|\hat{R}-R\|$ are small at all! Indeed Q, R are as ill-conditioned as A [19, Ch. 20].
- Solving $Ax = b$ and least-squares problems via the QR factorisation is stable.

7.7.1 (In)stability of Gram-Schmidt

(Nonexaminable) A somewhat surprising fact is that the Gram-Schmidt algorithm, when used for computing the QR factorisation, is not backward stable. Namely, orthogonality of the computed $\hat{Q}$ matrix is not guaranteed.

- Gram-Schmidt is subtle:
  - plain (classical) version: $\|\hat{Q}^T\hat{Q} - I\| \leq \epsilon(\kappa_2(A))^2$.
  - modified Gram-Schmidt (orthogonalise 'one vector at a time'): $\|\hat{Q}^T\hat{Q} - I\| \leq \epsilon\kappa_2(A)$.
  - Gram-Schmidt twice (G-S again on computed $\hat{Q}$) is excellent: $\|\hat{Q}^T\hat{Q} - I\| \leq \epsilon$.

8 Eigenvalue problems

We now turn to to the other problem in NLA: eigenvalue problems. First of all, recall that $Ax = \lambda x$ has no explicit solution (neither λ nor x); this is a huge difference from $Ax = b$ for which $x = A^{-1}b$.

From a mathematical viewpoint this marks an enormous point of departure: something that is is explicitly written vs. something that has no closed-from solution.

From a practical viewpoint the gulf is much smaller, because we have an extremely reliable algorithm for eigenvalue problems, namely the QR algorithm; so robust that it is essentially bulletproof provably backward stable.

Before we start describing the QR algorithm let's discuss a few interesting properties of eigenvalues.

- Eigenvalues are the roots of characteristic polynomial $\det(\lambda I - A)$ (Abel's theorem).
- For any polynomial p, there exist (infinitely many) matrices whose eigenvalues are roots of p.
- In particular, here is a nice and useful fact[18]: Let $p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, $a_i \in \mathbb{C}$. Then
  $p(\lambda) = 0 \Leftrightarrow \lambda$ eigenvalue of

$$C = \begin{bmatrix} -a_{n-1} & -a_{n-2} & \dots & -a_1 & -a_0 \\ 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & 0 \end{bmatrix} \in \mathbb{C}^{n \times n}.$$

  Proof: check that $x = [\lambda^{n-1}, \lambda^{n-2}, \dots, \lambda, 1]^T$ satisfies $Cx = \lambda x$.
- So no finite-step algorithm exists for $Ax = \lambda x$.

It follows that eigenvalue algorithms are necessarily iterative and approximate.

- Same for SVD, as $\sigma_i(A) = \sqrt{\lambda_i(A^T A)}$.
- But this doesn't mean they're inaccurate!

Usual goal: compute the Schur decomposition $A = UTU^*$: U unitary, T upper triangular.

[18] A great application of this is univariate optimisation: to minimise a polynomial $p(x)$, one can find the critical points by solving for $p'(x) = 0$, via the companion eigenvalues.

8.1 Schur decomposition

Theorem 8.1 *Let $A \in \mathbb{C}^{n\times n}$ (arbitrary square matrix). Then there exists a unitary matrix $U \in \mathbb{C}^{n\times n}$ s.t.*

$$A = UTU^*, \tag{9}$$

with T upper triangular. The decomposition (9) *is called the Schur decomposition.*

Note that

- $\text{eig}(A) = \text{eig}(T) = \text{diag}(T)$.
- T diagonal iff A is normal, i.e., $A^*A = AA^*$.

Proof: First, every matrix has eigenvalues, as they are roots of the characteristic polynomial. In particular, there exists v, λ_1 such that $Av = \lambda_1 v$. Now find $U_1 = [v_1, V_\perp]$ unitary. Then $AU_1 = U_1 \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix} \Leftrightarrow U_1^* A U_1 = \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix}$. Repeat on the lower-right $(n-1)\times(n-1)$ part to get
$U_{n-1}^* U_{n-2}^* \dots U_1^* A U_1 U_2 \dots U_{n-1} = T$. □

The reason we usually take the Schur form to be the goal is that its computation can be done in a backward stable manner. Essentially it boils down to the fact that the decomposition is based on orthogonal transformations.

- For normal matrices $A^*A = AA^*$, T must be diagonal and the Schur form is automatically diagonalised.
- For nonnormal A, if diagonalisation $A = X\Lambda X^{-1}$ is really necessary, this is done via starting with the Schur decomposition and further reducing T by solving what are called Sylvester equations; but this process involves nonorthogonal transformations and is not backward stable (nonexaminable).
- The Schur decomposition is among the few examples in NLA where the difference between $\mathbb{C}$ and $\mathbb{R}$ matters a bit. When working only in $\mathbb{R}$ one cannot always get a triangular T; it will have 2×2 blocks in the diagonal (these blocks have complex eigenvalues). This is essentially because real matrices can have complex eigenvalues. This is still not a major issue as eigenvalues of 2×2 matrices can be computed easily.

(This marks the end of "the first half" of the course (i.e., up until the basic facts about eigenvalues, but not its computation). This information is relevant only to MMSC students.)

8.2 The power method for finding the dominant eigenpair $Ax = \lambda x$

We now start describing the algorithms for solving eigenvalue problems. The first algorithm that we introduce is surprisingly simple and is based on the idea of keep multiplying the matrix A to an arbitrary vector.

This algorithm by construction is designed to compute only a single eigenvalue and its corresponding eigenvector, namely the dominant one. It is not able, at least as presented, to compute all the eigenvalues.

Despite its limitation and simplicity it is an extremely important algorithm and the underlying idea is in fact a basis for the ultimate QR algorithm that we use in order to compute all eigenvalues of a matrix. So here is the algorithm, called the *power method*.

Algorithm 8.1 The power method for computing the dominant eigenvalue and eigenvector of $A \in \mathbb{R}^{n\times n}$.

```
1: Let $x \in \mathbb{R}^n$ :=random initial vector (unless specified)
2: Repeat: $x = Ax$, $x = \frac{x}{\|x\|_2}$.
3: $\hat{\lambda} = x^TAx$ gives an estimate for the eigenvalue.
```

The method derives its name from the fact that after k iterations, the approximate eigenvector is parallel to $x = A^kx_0$, a large power of A times the initial (random) vector.

- Convergence analysis: suppose A is diagonalisable (generic assumption). We can write $x_0 = \sum_{i=1}^n c_iv_i$, $Av_i = \lambda_iv_i$ with $|\lambda_1| > |\lambda_2| > \cdots$. Then after k iterations,

$$x = C\sum_{i=1}^{n}\left(\frac{\lambda_i}{\lambda_1}\right)^k c_iv_i \to Cc_1v_1 \quad \text{as } k \to \infty \text{ for some scalar } C$$

- Converges geometrically $(\lambda, x) \to (\lambda_1, v_1)$ with **linear rate** $\frac{|\lambda_2|}{|\lambda_1|}$

- What does this imply about $A^k = QR$ as $k \to \infty$? First column of $Q \to v_1$ under mild assumptions.

Notes:

- Google pagerank & Markov chain are linked to the power method.

- As we'll see, the power method is basis for refined algorithms (QR algorithm, Krylov methods (Lanczos, Arnoldi,...))

8.2.1 Digression (optional): Why compute eigenvalues? Google PageRank

Let us briefly digress and talk about a famous application that at least used to solve an eigenvalue problem in order to achieve a familiar task: Google web search.

Once Google receives a user's inquiry with keywords, it needs to rank the relevant webpages, to output the most important pages first.

Here, the 'importance' of websites is determined by the dominant eigenvector of column-stochastic matrix (i.e., column sums are all 1)

$$A = \alpha P + \frac{(1-\alpha)}{n}\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}$$

P: scaled adjacency matrix ($P_{ij} = 1$ if i, j connected by an edge, 0 otherwise, and then scaled s.t. column sums are 1), $\alpha \in (0,1)$

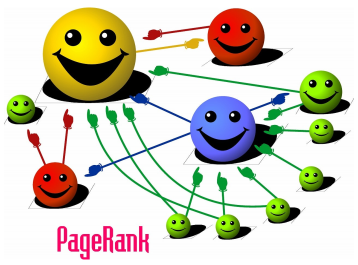


image from wikipedia

To solve this approximately (note that high accuracy isn't crucial here—getting the ordering wrong isn't the end of the world), Google does (at least in the old days) a few steps of power method: with initial guess x_0, $k = 0, 1, \ldots$

1. $x_{k+1} = Ax_k$
2. $x_{k+1} = x_{k+1}/\|x_{k+1}\|_2, \quad k \leftarrow k+1$, repeat.

- $x_k \to$ PageRank vector $v_1 : Av_1 = \lambda_1 v_1$. As A is a nonnegative matrix $A_{ij} \geq 0$, the dominant eigenvector v_1 can be taken to be nonnegative (by the Perron-Frobenius theorem; nonexaminable).

For more on pagerank etc, see Gleich [14] if interested (nonexaminable).

8.2.2 Shifted inverse power method

We saw that the convergence of the power method is governed by $|\lambda_1/\lambda_2|$, the ratio between the absolute value of dominant and the next dominant eigenvalue.

If this ratio is close to 1 the power method would converge very slowly. A natural question arises: can one speed up the convergence in such cases?

This leads to the *inverse power method*, or more generally shifted inverse power method, which can compute eigenpairs with much improved speed provided that the parameters (shifts) are chosen appropriately.

Algorithm 8.2 Shifted inverse power method for computing the eigenvalue and eigenvector of $A \in \mathbb{R}^{n\times n}$ closest to a prescribed value $\mu \in \mathbb{C}$.

```
1: Let $x \in \mathbb{R}^n$ :=random initial vector, unless specified.
2: Repeat: $x := (A - \mu I)^{-1}x$, $x = x/\|x\|$.
3: $\hat{\lambda} = x^T Ax$ gives an estimate for the eigenvalue.
```

Note that the eigenvalues of the matrix $(A - \mu I)^{-1}$ are $(\lambda(A) - \mu)^{-1}$.

- By the same analysis as above, shifted-inverse power method converges with **improved linear rate** $\frac{|\lambda_{\sigma(1)}-\mu|}{|\lambda_{\sigma(2)}-\mu|}$ to the eigenpair closest to μ. Here σ denotes a permutation of $\{1,\dots,n\}$ such that $|\lambda_{\sigma(1)}-\mu|$ minimises $|\lambda_i-\mu|$ over i.
- μ can change adaptively with the iterations. The choice $\mu := x^TAx$ gives the *Rayleigh quotient iteration*, with **quadratic** convergence $\|Ax^{(k+1)}-\lambda^{(k+1)}x^{(k+1)}\| = O(\|Ax^{(k)}-\lambda^{(k)}x^{(k)}\|^2)$; this is further improved to cubic convergence if A is symmetric (nonexaminable, but fun exercise).

It is worth emphasising that the improved convergence comes at a cost: one step of the shifted inverse power method requires a solution of a linear system, which is clearly much more expensive than the power method, $O(n^3)$ vs. $O(n^2)$ with a standard method.

9 The QR algorithm

We'll now describe an algorithm called the QR algorithm that is used universally for solving eigenvalue problems of moderate size (e.g. $n \lesssim 10000$), e.g. by MATLAB's `eig`. Given $A\in\mathbb{R}^{n\times n}$, the algorithm

- Finds all eigenvalues (approximately but reliably) in $O(n^3)$ flops,
- Is backward stable.

Sister problem: Given $A\in\mathbb{R}^{m\times n}$ compute SVD $A=U\Sigma V^T$: clearly very important!

- 'ok' algorithm: eig(A^TA) to find V, then normalise AV
- there's a better (but still closely related) algorithm: Golub-Kahan bidiagonalisation, discussed later in Section 10.2.

9.1 QR algorithm for $Ax=\lambda x$

As the name suggests, the QR algorithm is based on the QR factorisation of a matrix. Another key fact is that the eigenvalues of a product of two matrices remain the same when the order of the product is reversed (eig(AB)=eig(BA), problem sheet). The QR algorithm is essentially a simple combination of these two ideas, and the vanilla version is deceptively simple: basically take the QR factorisation, swap the order, take the QR and repeat the process. Namely,

Algorithm 9.1 The QR algorithm for finding the Schur decomposition of a square matrix A.

```
1: Set $A_1 = A$.
2: Repeat: $A_1 = Q_1R_1$,   $A_2 = R_1Q_1$,   $A_2 = Q_2R_2$,   $A_3 = R_2Q_2$,   $\dots$
```

Notes:

- A_k are all similar: $A_{k+1} = Q_k^T A_k Q_k$ (proved below)
- We shall 'show' that $A \to$ **triangular** (diagonal if A normal), under weak assumptions.
- Basically: QR(factorise)$\to$ RQ(swap)$\to$ $QR \to RQ \to \cdots$
- Fundamental work by Francis (61,62) and Kublanovskaya (63)
- This is a truly magical algorithm!
  - The algorithm is backward stable, as based on orthogonal transforms: essentially, $RQ = fl(Q^T(QR)Q) = Q^T(QR + \Delta(QR))Q$.
  - always converges (with shifts) in practice, but a global proof was unavailable until rather recently [3].
  - uses 'shifted inverse power method' (rational functions) without inversions

Again, the QR algorithm performs: $A_k = Q_k R_k$, $A_{k+1} = R_k Q_k$, repeat.
Let's look at its properties.

Theorem 9.1 *For $k \geq 1$,*

$$A_{k+1} = (Q^{(k)})^T A Q^{(k)}, \qquad A^k = (Q_1 \cdots Q_k)(R_k \cdots R_1) =: Q^{(k)} R^{(k)}.$$

Proof: recall $A_{k+1} = Q_k^T A_k Q_k$, repeat.

Proof by induction: $k = 1$ trivial.
Suppose $A^{k-1} = Q^{(k-1)} R^{(k-1)}$. We have

$$A_k = (Q^{(k-1)})^T A Q^{(k-1)} = Q_k R_k.$$

Then $AQ^{(k-1)} = Q^{(k-1)} Q_k R_k$, and so

$$A^k = AQ^{(k-1)} R^{(k-1)} = Q^{(k-1)} Q_k R_k R^{(k-1)} = Q^{(k)} R^{(k)} \square$$

QR algorithm and power method We can deduce that the QR algorithm is closely related to the power method. Let's try explain the connection. By Theorem 9.1, $Q^{(k)}R^{(k)}$ is the QR factorisation of A^k: as we saw in the analysis of the power method, the columns of A^k are 'dominated by the leading eigenvector' x_1, where $Ax_1 = \lambda_1 x_1$.
In particular, consider $A^k[1, 0, \ldots, 0]^{=} A^k e_1$:

- $A^k e_1 = R^{(k)}(1,1) Q^{(k)}(:,1)$, which is parallel to the first column of $Q^{(k)}$. (MATLAB notation is used here; e.g. $X(:,k)$ is the kth column of X).
- By the analysis of the power method, this implies $Q^{(k)}(:,1) \to x_1$

- Hence by $A_{k+1} = (Q^{(k)})^T A Q^{(k)}$, $A_k(:,1) \to [\lambda_1, 0, \dots, 0]^T$.

This tells us why the QR algorithm would eventually compute the eigenvalues—at least the dominant ones. One can even go further and argue that once the dominant eigenvalue has converged, we can expect the next dominant eigenvalue to start converging too—as due to the nature of the QR algorithm based on orthogonal transformations, we are then working in the orthogonal complement; and so on and so forth until the matrix A becomes upper triangular (and in the normal case this becomes diagonal), completing the solution of the eigenvalue problem. But there is much better news.

QR algorithm and inverse power method We have seen that the QR algorithm is related to the power method. We have also seen that the power method can be improved by a shift-and-invert technique. A natural question arises: can we introduce a similar technique in the QR algorithm? This question has an elegant and remarkable answer: not only is this possible but it is possible without ever inverting a matrix or solving a linear system (isn't that incredible!?).

Let's try and explain this. We start with the same expression for the QR iterates:

$$A^k = (Q_1 \cdots Q_k)(R_k \cdots R_1) =: Q^{(k)} R^{(k)}, \qquad A_{k+1} = (Q^{(k)})^T A Q^{(k)}.$$

Now take inverse: $A^{-k} = (R^{(k)})^{-1} (Q^{(k)})^T$.

Then the transpose: $(A^{-k})^T = Q^{(k)} (R^{(k)})^{-T}$

This is the QR factorisation of matrix $(A^{-k})^T$ with eigvals $r(\lambda_i) = \lambda_i^{-k}$.

$\Rightarrow$ Connection also with the (unshifted) inverse power method. Note that no matrix inverse is performed in the algorithm.

- This means the final column of $Q^{(k)}$ converges to minimum left eigenvector v_n with rate $\frac{|\lambda_n|}{|\lambda_{n-1}|}$, hence $A_k(n,:) \to [0, \dots, 0, \lambda_n]$.
- (Very) fast convergence if $|\lambda_n| \ll |\lambda_{n-1}|$.
- Can we achieve this situation? **Yes by shifts**.

QR algorithm with shifts and shifted inverse power method We are now ready to reveal the connection between the shift-and-invert power method and the QR algorithm with shifts.

First, here is the QR algorithm with shifts.

Algorithm 9.2 The QR algorithm with shifts for finding the Schur decomposition of a square matrix A.

```
1: Set $A_1 = A$, $k = 1$.
2: $A_k - s_k I = Q_k R_k$ (QR factorisation), typically shift $s_k = A_{nn}$
3: $A_{k+1} = R_k Q_k + s_k I$,    $k \leftarrow k + 1$, repeat.
```

Roughly, if $s_k \approx \lambda_n$, then $A_{k+1} \approx \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ & & & & \lambda_n \end{bmatrix}$ by the argument just made.

Theorem 9.2

$$\prod_{i=1}^{k}(A - s_i I) = Q^{(k)}R^{(k)}\,(= (Q_1 \cdots Q_k)(R_k \cdots R_1))\,.$$

Proof: Suppose true for $k-1$. Then the QR alg computes $(Q^{(k-1)})^T(A-s_kI)Q^{(k-1)} = Q_kR_k$, so $(A - s_kI)Q^{(k-1)} = Q^{(k-1)}Q_kR_k$, hence

$$\prod_{i=1}^{k}(A - s_iI) = (A - s_kI)Q^{(k-1)}R^{(k-1)} = Q^{(k-1)}Q_kR_kR^{(k-1)} = Q^{(k)}R^{(k)}.$$

Let's now take the inverse conjugate transpose[19] : $\prod_{i=1}^{k}(A - s_iI)^{-*} = \bar{Q}^{(k)}(R^{(k)})^{-*}$. $\square$

- This means the algorithm (implicitly) finds the QR factorisation of a matrix with eigvals $r(\lambda_j) = \prod_{i=1}^{k} \frac{1}{\bar{\lambda}_j - s_i}$.
- The rate is now the ratio of the smallest $\prod_{i=1}^{k}(\bar{\lambda}_j - s_i)$ to the runner-up; very fast if $s_i \approx \lambda_j$
- This reveals the intimate connection between shifted QR and shifted inverse power method, hence rational approximation.
- Ideally, we would like to choose the shift $s_k \approx \lambda_n$ to accelerate convergence. This is done by choosing s_k to be the bottom-right corner entry[20] of A_k; which is sensible given that with the QR algorithm, it tends to converge to an eigenvalue rapidly (with a few steps of the unshifted QR, it tends to converge to the smallest eigenvalue).

9.2 QR algorithm preprocessing: reduction to Hessenberg form

We've seen the QR iterations drives colored entries to 0 (esp. red ones)

$$A = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

- Hence $A_{n,n} \to \lambda_n$, so choosing $s_k = A_{n,n}$ is sensible.

[19]We are taking the inverse *complex conjugate* transpose because even if A is real, we may need/wish to take the shifts s_i to be nonreal.

[20]In production code a so-called Wilkinson shift is often used, which computes the eigenvalue of the 2×2 bottom-right submatrix of A_k, and takes s_k to be the one closer to the bottom-right entry of A_k.

- This reduces #QR iterations to $O(n)$; this is an empirical but reliable estimate.
- But each iteration of the QR algorithm is $O(n^3)$ for QR, overall $O(n^4)$. We next discuss a preprocessing technique to reduce the overall cost to $O(n^3)$.

The idea is to initially apply a series of deterministic orthogonal transformations that reduces the matrix to a form that is closer to upper triangular. This is done before starting the QR iterates, hence called a preprocessing step.

More precisely, to improve the cost of QR factorisation, we first reduce the matrix via orthogonal Householder transformations as follows:

$$A = \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \end{bmatrix}, \quad H_1 A = \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \end{bmatrix}, \quad H_1 = I - 2v_1 v_1^T, \ v_1 = \begin{bmatrix} 0 \\ \ast \\ \ast \\ \ast \\ \ast \end{bmatrix}$$

Then $H_1 A H_1 = \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \end{bmatrix}$. Repeat with $H_2 = I - 2v_2 v_2^T, v_2 = [0, 0, \ast, \ast, \ast]^T$, ...:

$$H_2 H_1 A H_1 H_2 = \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & & \ast & \ast & \ast \\ & & \ast & \ast & \ast \end{bmatrix}, \qquad H_3 H_2 H_1 A H_1 H_2 H_3 = \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & & \ast & \ast & \ast \\ & & & \ast & \ast \end{bmatrix},$$

$$A = \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \end{bmatrix} \overset{H_1}{\rightarrow} \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \end{bmatrix} \overset{H_2}{\rightarrow} \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & & \ast & \ast & \ast \\ & & \ast & \ast & \ast \end{bmatrix} \overset{H_3}{\rightarrow} \cdots \overset{H_{n-2}}{\rightarrow} \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & \ast & \ast & \ast \\ & \ast & \ast & \ast & \ast \\ & & \ast & \ast & \ast \\ & & & \ast & \ast \end{bmatrix}.$$

We have thus transformed the matrix A into upper Hessenberg form, i.e., $H_{ij} = 0$ if $i > j + 1$).

- Crucially, the Hessenberg structure is preserved throughout the QR algorithm: if $A_1 = QR$ Hessenberg, then so is $A_2 = RQ$. (Check!)
- Using Givens rotations, each QR iteration is $O(n^2)$, not $O(n^3)$.

We are thus going to first reduce A to Hessenberg form via a sequence of orthogonal transformations (thus preserving eigenvalues), and then apply the QR algorithm with shifts to the Hessenberg matrix. The remaining task is thus to drive the subdiagonal elements $\ast$ to 0. A is thus reduced to a triangular form, revealing the Schur decomposition of A (if one traces back all the orthogonal transformations employed).

Once the bottom-right is converged $|{\color{red}\ast}| < \epsilon$, we take

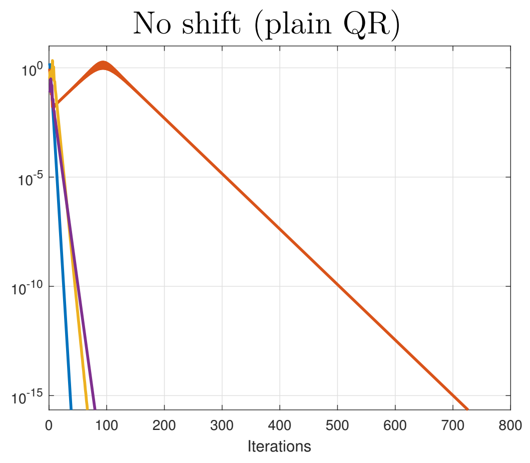
$$\begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ {\color{red}\ast} & \ast & \ast & \ast & \ast \\ & {\color{red}\ast} & \ast & \ast & \ast \\ & & {\color{red}\ast} & \ast & \ast \\ & & & {\color{red}\ast} & \ast \end{bmatrix} \approx \begin{bmatrix} \ast & \ast & \ast & \ast & \ast \\ {\color{red}\ast} & \ast & \ast & \ast & \ast \\ & {\color{red}\ast} & \ast & \ast & \ast \\ & & {\color{red}\ast} & \ast & \ast \\ & & & & \ast \end{bmatrix}$$

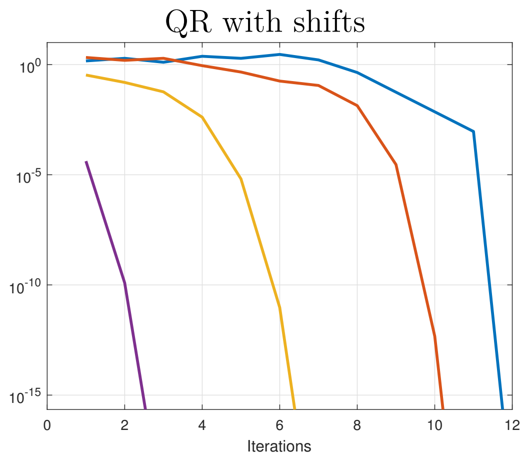
and continue with shifted QR on $(n-1) \times (n-1)$ block, repeat. This process is called **deflation**.

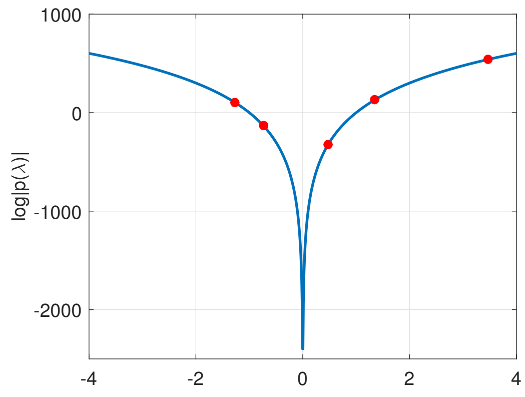
- Empirically, the shifted QR algorithm needs $2-4$ iterations for convergence per eigenvalue. Overall, the cost is $O(n^3), \approx 25n^3$ flops.

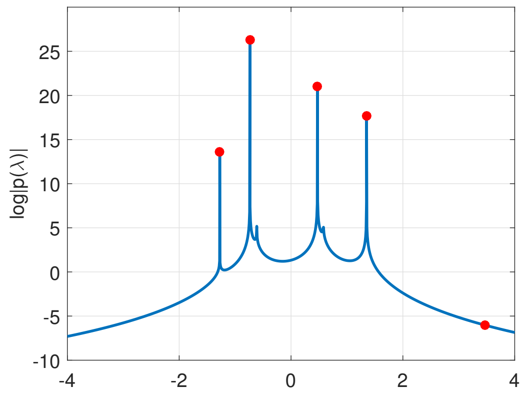
9.2.1 The (shifted) QR algorithm in action

Let's see how the QR algorithm works with a small, 5×5 example. The plots show the convergence of the subdiagonal entries $|A_{i+1,i}|$ (note that their convergence signifies the convergence of the QR algorithm as we initially reduce the matrix to Hessenberg form).









In light of the connection to rational functions as discussed above, here we plot the underlying functions (red dots: eigvals). The idea here is that we want the functions to take large values at the target eigenvalue (at the current iterate) in order to accelerate convergence.

9.2.2 (Optional) QR algorithm: other improvement techniques

We have completed the description of the main ingredients of the QR algorithm. Nonetheless, there are a few more bells and whistles that go into to a production code. We will not get into too much detail but here is a list of the key players.

- Double-shift strategy for $A \in \mathbb{R}^{n\times n}$
  - $(A - sI)(A - \bar{s}I) = QR$ using only real arithmetic
- Aggressive early deflation [Braman-Byers-Mathias 2002 [6]]
  - Examine lower-right (say 100×100) block instead of $(n, n-1)$ element
  - dramatic speedup ($\approx \times 10$)
- Balancing $A \leftarrow DAD^{-1}$, D: diagonal
  - Aims at reducing $\|DAD^{-1}\|$: often yields better-conditioned eigenvalues.

10 QR algorithm continued

10.1 QR algorithm for symmetric A

so far we have not assumed anything about the matrix besides that it is square. This is for good reason—because the QR algorithm is applicable to solving any eigenvalue problem. However, in many situations the eigenvalue problem is *structured*. In particular the case where the matrix is symmetric arises very frequently in practice and it comes with significant simplification of the QR algorithm, so it deserves special attention.

- Most importantly, symmetry immediately implies that the initial reduction to Hessenberg form reduces A to tridiagonal:
$$A = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix} \xrightarrow{Q_1} \begin{bmatrix} * & * & & & \\ * & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \end{bmatrix} \xrightarrow{Q_2} \begin{bmatrix} * & * & & & \\ * & * & * & & \\ & * & * & * & * \\ & & * & * & * \\ & & * & * & * \end{bmatrix} \xrightarrow{Q_3} \begin{bmatrix} * & * & & & \\ * & * & * & & \\ & * & * & * & \\ & & * & * & * \\ & & & * & * \end{bmatrix}$$
- QR steps for tridiagonal: requires $O(n)$ flops instead of $O(n^2)$ per step.
- Powerful alternatives are available for tridiagonal eigenvalue problems (divide-conquer [Gu-Eisenstat 95], HODLR [Kressner-Susnjara 19],...)

- Cost: $\frac{4}{3}n^3$ flops for eigvals, $\approx 10n^3$ for eigvecs (store Givens rotations to compute eigvecs).

- Another approach (nonexaminable): spectral divide-and-conquer (Nakatsukasa-Freund, Higham); which is all about using a carefully chosen rational approximation to find orthogonal matrices V_i such that

$$A = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix} \xrightarrow{V_1} \begin{bmatrix} * & * & * & & \\ * & * & * & & \\ * & * & * & & \\ & & & * & * \\ & & & * & * \end{bmatrix} \xrightarrow{V_2} \begin{bmatrix} * & & & & \\ & * & * & & \\ & * & * & & \\ & & & * & \\ & & & & * \end{bmatrix} \xrightarrow{V_3} \begin{bmatrix} * & & & & \\ & * & & & \\ & & * & & \\ & & & * & \\ & & & & * \end{bmatrix} = \Lambda.$$

10.2 Computing the SVD: Golub-Kahan's bidiagonalisation algorithm

The key ideas of the QR algorithm turn out to be applicable for the computation of the SVD. Clearly this is a major step—given the importance of the SVD. In particular the SVD algorithm has strong connections to the symmetric QR algorithm. This is perhaps not surprising given the strong connection between the SVD and symmetric eigenvalue problems, as seen e.g. in the proof of the SVD (Theorem 2.1). Indeed, one could compute the SVD via the Gram matrix A^TA and its eigenvalue decomposition $A^TA = V\Sigma^2V^T$. However, this results in inaccurately computed singular values (at least for the small ones). There is a better approach.

A noteworthy difference is that in the SVD $A = U\Sigma V^T$ the two matrices U and V are allowed to be different. The algorithm respects this, and instead of initially reducing the matrix to tridiagonal form, it reduces it to a so-called bidiagonal form.

Here is how it works. Apply Householder reflectors from left and right (different ones) to **bidiagonalise**

$$A \to B = H_{L,n} \cdots H_{L,1} A H_{R,1} H_{R,2} \cdots H_{R,n-2}$$

$$A \xrightarrow{H_{L,1}} \begin{bmatrix} \star & \star & \star & \star \\ & \star & \star & \star \\ & \star & \star & \star \\ & \star & \star & \star \\ & \star & \star & \star \end{bmatrix} \xrightarrow{H_{R,1}} \begin{bmatrix} \star & \star & & \\ & \star & \star & \star \\ & \star & \star & \star \\ & \star & \star & \star \\ & \star & \star & \star \end{bmatrix} \xrightarrow{H_{L,2}} \begin{bmatrix} \star & \star & & \\ & \star & \star & \star \\ & & \star & \star \\ & & \star & \star \\ & & \star & \star \end{bmatrix} \xrightarrow{H_{R,2}} \begin{bmatrix} \star & \star & & \\ & \star & \star & \\ & & \star & \star \\ & & \star & \star \\ & & \star & \star \end{bmatrix} \xrightarrow{H_{L,3}} \begin{bmatrix} \star & \star & & \\ & \star & \star & \\ & & \star & \star \\ & & & \star \\ & & & \star \end{bmatrix} \xrightarrow{H_{L,4}} \begin{bmatrix} \star & \star & & \\ & \star & \star & \\ & & \star & \star \\ & & & \star \\ & & & \end{bmatrix} = B,$$

- Since the transformations are all orthogonal multiplications, singular values are preserved $\sigma_i(A) = \sigma_i(B)$. In addition, these are backward stable operations.

- Once bidiagonalised, one can complete the SVD as follows:
  - Mathematically, run the QR algorithm on B^TB (symmetric tridiagonal)
  - More elegant: divide-and-conquer [Gu-Eisenstat 1995] or dqds algorithm [Fernando-Parlett 1994] (nonexaminable)

- Cost: $\approx 4mn^2$ flops for singular values Σ, $\approx 20mn^2$ flops to also compute the singular vectors U, V.

10.3 (Optional but important) QZ algorithm for generalised eigenvalue problems

An increasingly important class of eigenvalue problems is the so-called generalised eigenvalue problems involving two matrices. You have probably not seen them before, but the number of applications that boil down to a generalised eigenvalue problem has been increasing rapidly.

A generalised eigenvalue problem is of the form

$$Ax = \lambda Bx, \qquad A, B \in \mathbb{C}^{n\times n}, \quad x \neq 0, \lambda \in \mathbb{C}.$$

- The matrices A, B are given. The goal is to find the eigenvalues λ and their corresponding eigenvectors x.
- There are usually (incl. when B is nonsingular) n eigenvalues, which are the roots of $\det(A - \lambda B)$.
- When B is invertible, one can reduce the problem to $B^{-1}Ax = \lambda x$.
- Important case: A, B symmetric, B positive definite: in this case λ are all real.

How do se solve generalised eigenvalue problems? There is a variant of the QR algorithm called the QZ algorithm: it looks for unitary Q, Z s.t. QAZ, QBZ both upper triangular.

- Then $\mathrm{diag}(QAZ)/\mathrm{diag}(QBZ)$ are the eigenvalues.
- Algorithm: first reduce A, B to Hessenberg-triangular form.
- Then implicitly do QR to $B^{-1}A$ (without inverting B).
- Cost: $\approx 50n^3$.
- See [15] for details.

10.4 (Optional) Tractable eigenvalue problems

Beyond generalised eigenvalue problems there are more exotic generalisations of eigenvalue problems and reliable algorithms have been proposed for solving them.

Thanks to the remarkable developments in NLA research, the following problems are 'tractable' in that reliable algorithms exist for solving them, at least when the matrix size n is modest (say in the thousands).

- Standard eigenvalue problems $Ax = \lambda x$
  - symmetric ($4/3n^3$ flops for eigvals, $+9n^3$ for eigvecs)
  - nonsymmetric ($10n^3$ flops for eigvals, $+15n^3$ for eigvecs)

- SVD $A = U\Sigma V^T$ for $A \in \mathbb{R}^{m\times n}$: ($\frac{8}{3}mn^2$ flops for singvals, $+20mn^2$ for singvecs)

- Generalised eigenvalue problems $Ax = \lambda Bx$, $A, B \in \mathbb{C}^{n\times n}$

- Polynomial eigenvalue problems, e.g. $P(\lambda)x = (\lambda^2 A + \lambda B + C)x = 0$, $A, B, C \in \mathbb{C}^{n\times n}$ (degree $k = 2$, quadratic eigenvalue problem):$\approx 20(nk)^3$

- Nonlinear problems, e.g. $N(\lambda)x = (A\exp(\lambda) + B)x = 0$
  - often solved via approximating by polynomial $N(\lambda) \approx P(\lambda)$
  - more difficult: $A(x)x = \lambda x$: eigenvector nonlinearity

Further speedup is often possible when structure present (e.g. sparse, low-rank)

11 Iterative methods: introduction

This section marks a point of departure from previous sections. So far we've been discussing *direct* methods. Namely, we've covered the LU-based linear system solution, QR-based solution for least-squares, and the QR algorithm for eigenvalue problems[21].

Direct methods are

- Incredibly reliable, and backward stable
- Works like magic if $n \lesssim 10000$
- But not if n is larger!

A 'big' matrix problem is one for which direct methods aren't practical. Historically, as computers become increasingly faster, roughly

- 1950: $n \geq 20$
- 1965: $n \geq 200$
- 1980: $n \geq 2000$
- 1995: $n \geq 20000$
- 2010: $n \geq 100000$
- 2020: $n \geq 500000$ ($n \geq 50000$ on a standard desktop computer)

[21]Note that the QR algorithm is iterative so it is not exactly correct to call it direct; however the nature of its consistency and robustness together with the fact that the cost is more less predictable has rightfully earned it the classification as a direct method.

has been considered 'too large for direct methods'. While it's clearly good news that our ability to solve problems with direct methods has been improving, the scale of problems we face in data science has been growing at the same (or even faster) pace! For such problems, we need to turn to alternative algorithms: we'll cover **iterative** and **randomised** methods. We first discuss iterative methods, with a focus on *Krylov subspace methods.*

Direct vs. iterative methods Broadly speaking, the idea of iterative methods is to:

- Gradually refine the solution iteratively. That is, we get solutions $x_1, x_2, \ldots$, with the hope that $x_k \to x_*$, the exact solution, as k increases.
- Each iteration should be (a lot) cheaper than direct methods, usually $O(n^2)$ or less.
- Iterative methods can be (but not always) much faster than direct methods.
- Tends to be (slightly) less robust, nontrivial/problem-dependent analysis.
- Often, after $O(n^3)$ work it still gets the exact solution (ignoring roundoff errors). But one would hope to get a (acceptably) good solution long before that!

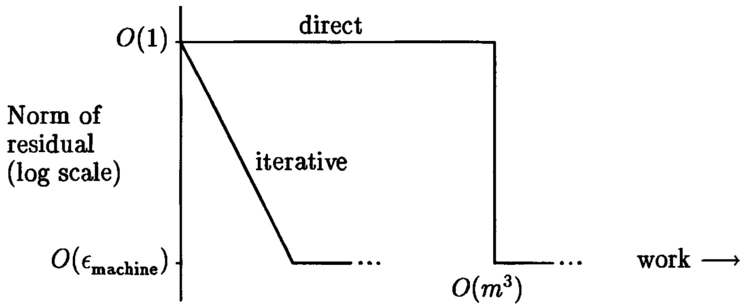


Figure 2: Rough comparison of direct vs. iterative methods, image from Trefethen-Bau [36]

Each iteration of most iterative methods is based on multiplying A to a vector; clearly cheaper than an $O(n^3)$ direct algorithm. We'll focus on **Krylov subspace methods**. (Other iterative methods we won't get into details include the Gauss-Seidel, SOR and Chebyshev semi-iterative methods.)

11.1 Polynomial approximation: basic idea of Krylov

The big idea behind Krylov subspace methods is to approximate the solution in terms of a polynomial of the matrix times a vector. Namely, in Krylov subspace methods, we look for an (approximate) solution $\hat{x}$ (for $Ax = b$ or $Ax = \lambda x$) of the form (after kth iteration)

$$\hat{x} = p_{k-1}(A)v \ ,$$

where p_{k-1} is a polynomial of degree (at most) $k-1$, that is, $p_{k-1}(z) = \sum_{i=0}^{k-1} c_i z^i$ for some coefficients $c_i \in \mathbb{C}$, and $v \in \mathbb{R}^n$ is the *initial vector* (usually $v = b$ for linear systems, for eigenproblems v is usually a random vector). That is, $p_{k-1}(A) = \sum_{i=0}^{k-1} c_i A^i$, and $\hat{x} = \sum_{i=0}^{k-1} c_i A^i v$.

Natural questions:

- Why would this be a good idea?
  - Clearly, 'easy' to compute, in that only matrix-vector products with A are needed to form $\hat{x}$.
  - One example: recall power method $\hat{x} = A^{k-1}v = \tilde{p}_{k-1}(A)v$, for $\tilde{p}_{k-1}(z) = z^{k-1}$ Krylov finds a "better/optimal" polynomial $p_{k-1}(A)$. When the eigenvalues of A are well-behaved, we'll see that $O(1)$ iterations suffices for convergence.
  - We'll see more cases where Krylov is powerful.
- How to turn this idea into an algorithm?
  - Find an orthonormal basis: Arnoldi (next), Lanczos.

11.2 Orthonormal basis for $\mathcal{K}_k(A, b)$

Goal: Find approximate solution $\hat{x} = p_{k-1}(A)b$, i.e. in Krylov subspace

$$\mathcal{K}_k(A, b) := \text{span}([b, Ab, A^2b, \dots, A^{k-1}b])$$

You will want to convince yourself that any vector in the Krylov subspace can be written as a polynomial of A times the vector b. (Problem sheet)

An important and non-trivial step towards finding a good solution is to form an orthonormal basis for the Krylov subspace.

First step: form an orthonormal basis Q, s.t. solution $x \in \mathcal{K}_k(A, b)$ can be written as $x = Qy$

- Naive idea: Form matrix $[b, Ab, A^2b, \dots, A^{k-1}b]$, then compute its QR factorisation.
  - But $[b, Ab, A^2b, \dots, A^{k-1}b]$ is usually terribly conditioned! Dominated by the leading eigvec
  - Q is therefore extremely ill-conditioned, and hence inaccurately computed

- A much better solution is to do the Arnoldi iteration
  - Multiply A once at a time to the latest orthonormal vector q_i
  - Then orthogonalise Aq_i against previous q_j's ($j = 1, \dots, i-1$), as in Gram-Schmidt. The idea is to orthogonalise the vectors as soon as they become available. This is really a neater idea than it might sound.

11.3 Arnoldi iteration

Here is a pseudocode of the Arnold iteration. Essentially, what it does is multiply the matrix A, orthogonalise against the previous vectors, and repeat. This is massively (often exponentially w.r.t. conditioning) better than multiplying A $k-1$ times and then orthogonalising.

Algorithm 11.1 The Arnoldi iteration for finding an orthonormal basis for Krylov subspace $\mathcal{K}_k(A,b)$. Given $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$, outputs orthonormal $[q_1, q_2, \dots,]$ and Hessenberg $H_{ij} = h_{ij}$.

```
1: Set $q_1 = b/\|b\|_2$
2: For $k = 1, 2, \dots,$
3:    set $v = Aq_k$
4:       for $j = 1, 2, \dots, k$
5:          $h_{jk} = q_j^T v$, $v = v - h_{jk}q_j$ % orthogonalise against $q_j$ via (modified) Gram-Schmidt

6:       end for
7:    $h_{k+1,k} = \|v\|_2$, $q_{k+1} = v/h_{k+1,k}$
8: End for
```

Let's verify a basic result about Arnoldi, that the vectors q_i computed by Arnoldi spans the Krylov subspace.

Theorem 11.1 *Suppose in Algorithm 11.1 that $h_{k+1,k} \neq 0$ for $k = 1, \dots, \ell$. Then for $k = 1, \dots, \ell$,*

$$\mathit{Span}(q_1, \dots, q_k) = \mathcal{K}_k(A, b). \tag{10}$$

Proof: Induction on ℓ. When $\ell = 1$ the result is trivial. Note that $q_1 = q_\ell = p_{\ell-1}(A)b$, where $p_{\ell-1}$ is a polynomial of degree $\ell - 1 = 0$ (constant).

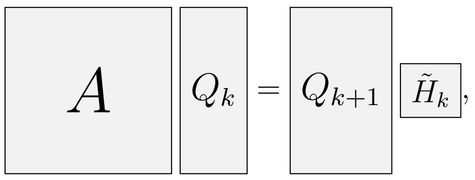
We take the induction hypothesis to be that (10) holds for $k = 1, \dots, \hat{\ell}$, and $q_{\hat{\ell}}$ is of exact degree $\hat{\ell} - 1$.

Then the $(\hat{\ell}+1)$th Arnoldi iteration gives $q_{\hat{\ell}+1} = \frac{1}{h_{\hat{\ell}+1,\hat{\ell}}}(Aq_{\hat{\ell}} - \sum_{j=1}^{\hat{\ell}} h_{j,\hat{\ell}}q_j)$, which is of exact degree $\hat{\ell}$. □

Theorem 11.1 assumes that $h_{k+1,k} \neq 0$; if this doesn't hold it's not bad news, we are (usually) actually in a lucky situation. For example, this means the solution for $Ax = b$ lies in the subspace (so with a reasonable method like GMRES below, we obtain the exact solution), and when solving an eigenvalue problem, an invariant subspace has been found. This situation is called a happy breakdown (and once this happens, we cannot/don't proceed further with Arnoldi).

- Hence after k steps, $AQ_k = Q_{k+1}\tilde{H}_k = Q_kH_k + q_{k+1}[0, \dots, 0, h_{k+1,k}]$, with $Q_k =$

$[q_1, q_2, \dots, q_k]$, $Q_{k+1} = [Q_k, q_{k+1}]$, $\mathrm{span}(Q_k) = \mathrm{span}([b, Ab, \dots, A^{k-1}b])$



$$\tilde{H}_k = \underbrace{\begin{bmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,k} \\ h_{2,1} & h_{2,2} & \dots & h_{2,k} \\ & \ddots & & \vdots \\ & & h_{k,k-1} & h_{k,k} \\ & & & h_{k+1,k} \end{bmatrix}}_{\mathbb{R}^{(k+1)\times k} \text{ upper Hessenberg}}, \quad Q_{k+1}^T Q_{k+1} = I_{k+1}$$

- Cost is k A-multiplications$+O(k^2)$ inner products ($O(nk^2)$ flops)

12 Arnoldi and GMRES for $Ax = b$

This is an exciting section as we will describe the GMRES (Generalised Minimal RESidual) algorithm. This algorithm has been so successful that in the 90s the paper that introduced it was the most cited paper in all of applied mathematics.

GMRES attempts to find an approximate solution in the Krylov subspace that is in some sense the best possible, in that the residual is minimised.

Idea (very simple!): minimise residual in Krylov subspace: [Saad-Schulz 86 [31]]

$$x = \mathrm{argmin}_{x\in\mathcal{K}_k(A,b)} \|Ax - b\|_2. \tag{11}$$

In order to solve this, The algorithm cleverly takes advantage of the structure afforded by Arnoldi.

Algorithm: Given $AQ_k = Q_{k+1}\tilde{H}_k$ and writing $x = Q_k y$, rewrite as

$$\begin{aligned} \min_y \|AQ_k y - b\|_2 &= \min_y \|Q_{k+1}\tilde{H}_k y - b\|_2 \\ &= \min_y \left\| \begin{bmatrix} \tilde{H}_k \\ 0 \end{bmatrix} y - \begin{bmatrix} Q_{k+1}^T \\ Q_{k+1,\perp}^T \end{bmatrix} b \right\|_2 \\ &= \min_y \left\| \begin{bmatrix} \tilde{H}_k \\ 0 \end{bmatrix} y - \|b\|_2 e_1 \right\|_2, \quad e_1 = [1, 0, \dots, 0]^T \in \mathbb{R}^n \end{aligned}$$

(where $[Q_{k+1}, Q_{k+1,\perp}]$ is orthogonal; we're using the same trick as in least-squares)

- Minimised when $\|\tilde{H}_k y - Q_{k+1}^T b\|_2$ is minimised; this is a Hessenberg least-squares problem.

- Solve via the QR-based approach described in Section 6.5. Here it's even simpler as the matrix is Hessenberg: k Givens rotations yields the QR factorisation, then a triangular solve will complete the solution, so $O(k^2)$ work in addition to Arnoldi; recall Section 6.4. Moreover, as k grows, $\tilde{H}_k$ only changes by having a new column and row (the top-left part stays the same). This can be taken advantage of in the solution.

Notice how the Arnoldi decomposition reduced the problem (11) effortlessly to a very convenient one $\|\tilde{H}_k y - Q_{k+1}^T b\|_2$; this wouldn't be the case if we "merely" had a QR factorisation of the Krylov matrix $[b, Ab, A^2b, \dots, A^{k-1}b]$. We'll see this phenomenon many times in this section and next.

12.1 GMRES convergence: polynomial approximation

We now study the convergence of GMRES. As with any Krylov subspace method, the analysis is based on the theory of polynomial approximation.

Theorem 12.1 (GMRES convergence) *Assume that A is diagonalisable, $A = X\Lambda X^{-1}$. Then the kth GMRES iterate x_k satisfies*

$$\|Ax_k - b\|_2 \le \kappa_2(X) \min_{p\in\mathcal{P}_k, p(0)=1} \max_{z\in\lambda(A)} |p(z)| \|b\|_2.$$

Proof: Recall that $x_k \in \mathcal{K}_k(A,b) \Rightarrow x_k = p_{k-1}(A)b$, where p_{k-1} is a polynomial of degree at most $k-1$. Hence the GMRES solution is

$$\begin{aligned}\min_{x_k\in\mathcal{K}_k(A,b)} \|Ax_k - b\|_2 &= \min_{p_{k-1}\in\mathcal{P}_{k-1}} \|Ap_{k-1}(A)b - b\|_2 \\ &= \min_{\tilde{p}\in\mathcal{P}_k, \tilde{p}(0)=0} \|(\tilde{p}(A) - I)b\|_2 \\ &= \min_{p\in\mathcal{P}_k, p(0)=1} \|p(A)b\|_2\end{aligned}$$

If A is diagonalisable, $A = X\Lambda X^{-1}$,

$$\begin{aligned}\|p(A)\|_2 &= \|Xp(\Lambda)X^{-1}\|_2 \le \|X\|_2\|X^{-1}\|_2\|p(\Lambda)\|_2 \\ &= \kappa_2(X) \max_{z\in\lambda(A)} |p(z)|\end{aligned}$$

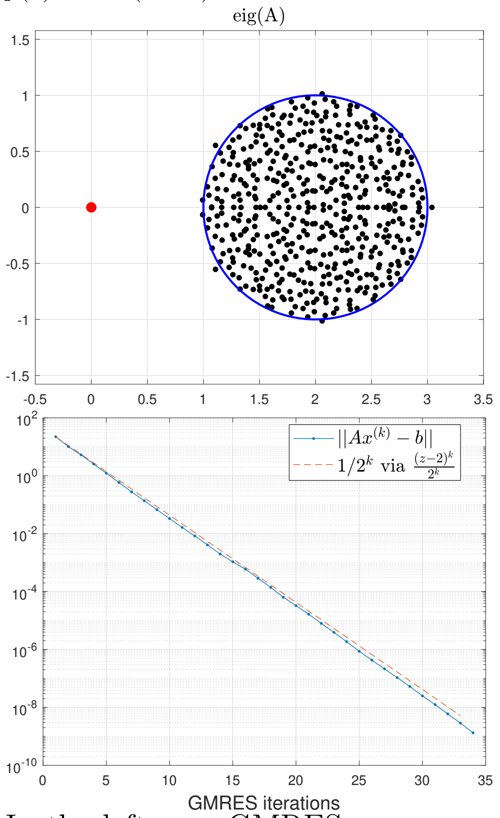
(recall that $\lambda(A)$ is the set of eigenvalues of A) □

Noting that $\|p(A)b\|_2 \le \|p(A)\|_2\|b\|_2$, here is the interpretation of this analysis: We would like to find a polynomial s.t. $p(0) = 1$ and $|p|$ is small at the eigenvalues of A, that is, $|p(\lambda_i)|$ is small for all i.

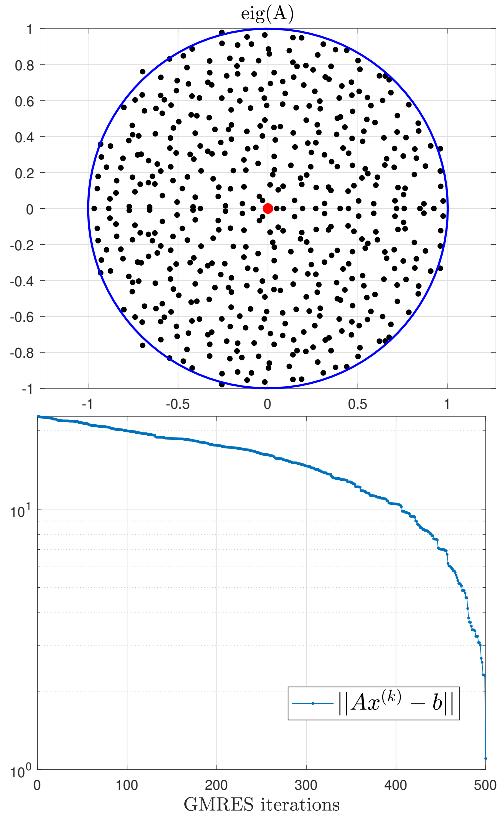
The question becomes, what type of eigenvalues are 'nice' for this to hold?

GMRES example G: Gaussian random matrix ($G_{ij} \sim N(0,1)$, i.i.d.) $G/\sqrt{n}$, $n = 1000$: eigenvalues in unit disk.

$A = 2I + G/\sqrt{n}$,
$p(z) = 2^{-k}(z-2)^k$

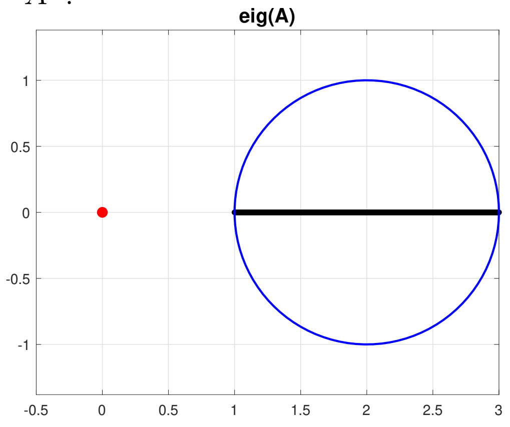


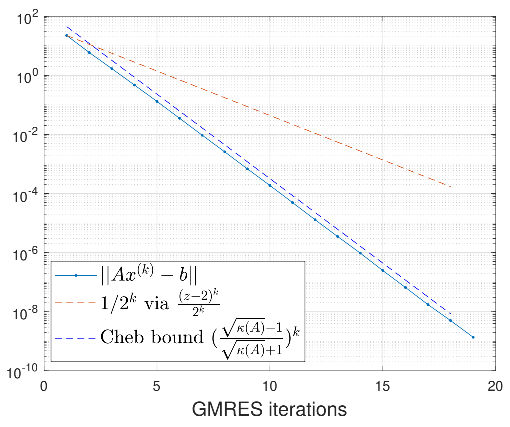
$A = G/\sqrt{n}$



In the left case GMRES converges rapidly, whereas it does not in the right case. The difference can be explained by noting that in the left case, there is a polynomial $p(z) = 2^{-k}(z-2)^k$ that is very small in the yellow disk while $p(0) = 1$, for which $\max_{z\in\lambda(A)} |p(z)| \lesssim 2^{-k}$. By contrast, in the right example we don't have such an obvious choice; and indeed there isn't a polynomial for which $\max_{z\in\lambda(A)} |p(z)|$ decays rapidly.

Now suppose the eigenvalues are real (and positive); this would be the case obviously if $A = A^T$.





In this case the naive convergence bound $1/2^k$ obtained via the polynomial $p(z) = 2^{-k}(z-2)^k$ isn't very sharp. The reason is that a more specific polynomial, namely a Chebyshev polynomial, can "concentrate its energy" on the real interval $[1, 3]$; we'll return to this shortly in Section 13.3 in the analysis of the CG algorithm.

Initial vector. Sometimes a good initial guess x_0 for x_* is available. In this case we take the initial residual $r_0 = Ax_0 - b$, and work in the affine space $x_k = x_0 + \mathcal{K}_k(A, r_0)$. All the analysis above can be modified readily to allow for this situation with essentially the same conclusion. Clearly, if one has a good x_0, by all means that should be used.

12.2 When does GMRES converge fast?

Recall GMRES solution satisfies (assuming A is diagonalisable+nonsingular)

$$\min_{x\in\mathcal{K}_k(A,b)} \|Ax - b\|_2 = \min_{p\in\mathcal{P}_k, p(0)=1} \|p(A)b\|_2 \le \kappa_2(X) \max_{z\in\lambda(A)} |p(z)| \|b\|_2. \tag{12}$$

As the examples above illustrate, $\max_{z\in\lambda(A)} |p(z)|$ is small when

- The eigenvalues $\lambda(A)$ are clustered away from 0, as then a good p can be found quite easily.

  Another interesting case is:

- When $\lambda(A)$ takes $k(\ll n)$ distinct values
  - Then convergence in k GMRES iterations; to see this, note that a polynomial p can be found such that $p(\lambda_i) = 0$ at the k eigenvalues, that is, $p(z) = C\prod_{i=1}^{k}(z - \lambda_i)$, and choose C such that $p(0) = 1$.

12.3 Preconditioning for GMRES

We've seen that GMRES is great if the eigenvalues are clustered away from 0. If this is not true, GMRES can really require close to (or equal to!) the full n iterations for convergence.

This is undesirable—at this point we've spent more computation than a direct method! We need a workaround.

The idea of *preconditioning* is to instead of solving

$$Ax = b,$$

find a "preconditioner" $M \in \mathbb{R}^{n\times n}$ and solve

$$MAx = Mb$$

Of course, M cannot be an arbitrary matrix. It has to be chosen very carefully. Desiderata of M are

- M is simple enough s.t. applying M to a vector is easy (note that each GMRES iteration requires MA-multiplication to a vector, say MAx, which is done as $M(Ax)$), and one of
  1. MA has clustered eigenvalues away from 0.
  2. MA has a small number of distinct nonzero eigenvalues.
  3. MA is well-conditioned $\kappa_2(MA) = O(1)$; then solve the normal equation $(MA)^T MAx = (MA)^T Mb$.

Preconditioners: examples

- ILU (Incomplete LU) preconditioner: $A \approx LU, M = (LU)^{-1} = U^{-1}L^{-1}$, L, U 'as sparse as A' $\Rightarrow MA \approx I$ (hopefully; 'cluster away from 0').

- For $\tilde{A} = \begin{bmatrix} A & B \\ C & 0 \end{bmatrix}$, set $M = \begin{bmatrix} A^{-1} & \\ & (CA^{-1}B)^{-1} \end{bmatrix}$. Then if M nonsingular, $M\tilde{A}$ has eigvals$\in \{1, \frac{1}{2}(1 \pm \sqrt{5})\} \Rightarrow$ 3-step convergence. [Murphy-Golub-Wathen 2000 [26]]
- Multigrid-based, operator preconditioning, ...
- A "perfect" preconditioner is $M = A^{-1}$; as then preconditioned GMRES will converge in one step. Obviously this M isn't easy to apply (if it was then we're done!). Preconditioning, therefore, can be regarded as an act of efficiently approximating the inverse.

Finding effective preconditioners is a never-ending research topic.
Prof. Andy Wathen is our Oxford expert!

12.4 Restarted GMRES

Another practical GMRES technique is restarting. For k iterations, GMRES costs k matrix-vector multiplications$+O(nk^2)$ for orthogonalisation $\rightarrow$ Arnoldi eventually becomes expensive[22].

Practical solution: restart by solving 'iterative refinement':

1. Stop GMRES after $k_{\max}$ (prescribed) steps to get an approximate solution $\hat{x}_1$.
2. Solve $A\tilde{x} = b - A\hat{x}_1$ via GMRES. (This is a linear system with a different right-hand side).
3. Obtain solution $\hat{x}_1 + \tilde{x}$.

Sometimes more than one restarts are needed. Note that restarting means losing the GMRES optimality (12).

13 Symmetric case: Lanczos and Conjugate Gradient method for $Ax = b$, $A \succ 0$

As is often the case, when $A = A^T$ the situation simplifies significantly. First, the Arnoldi decomposition becomes what is called Lanczos.

13.1 Lanczos iteration and Lanczos decomposition

When A is symmetric, the Arnoldi decomposition simplifies to

$$AQ_k = Q_k T_k + q_{k+1}[0, \ldots, 0, t_{k+1,k}],$$

where T_k is symmetric tridiagonal (proof: just note $H_k = Q_k^T A Q_k$ in Arnoldi). This yields the Lanczos decomposition

$$\boxed{\quad A \quad}\,\boxed{Q_k} = \boxed{Q_{k+1}}\,\boxed{\tilde{T}_k}, \qquad \tilde{T}_k = \underbrace{\begin{bmatrix} t_{1,1} & t_{1,2} & & \\ t_{2,1} & t_{2,2} & \ddots & \\ & \ddots & & t_{k-1,k} \\ & & t_{k,k-1} & t_{k,k} \\ & & & t_{k+1,k} \end{bmatrix}}_{\mathbb{R}^{(k+1)\times k} \text{ symmetric tridiagonal}}, \qquad Q_{k+1}^T Q_{k+1} = I_{k+1}$$

- The vectors q_k form a 3-term recurrence $t_{k+1,k}q_{k+1} = (A - t_{k,k})q_k - t_{k-1,k}q_{k-1}$. Orthogonalisation is necessary only against last two vectors q_k, q_{k-1}

[22]If the cost of each matrix-vector multiplication Av is $O(n^2)$ (as would be the case if A is unstructured and dense), this part is always dominant in GMRES. However, Av is often much faster to compute, e.g. when A is sparse.

- This achieves significant speedup over Arnoldi; cost is k A-multiplication plus $O(k)$ inner products ($O(nk)$ operations, instead of Arnoldi's $O(nk^2)$).
- In floating-point arithmetic, sometimes the computed Q_k loses orthogonality and re-orthogonalisation may be necessary (nonexaminable, see e.g. Demmel [9])

13.2 CG algorithm for $Ax = b$, $A \succ 0$

Here we introduce the conjugate gradient (CG) method. CG is not only important in practice but has historical significance in that it was the very first Krylov algorithm to be introduced (and initially made a big hype, but then took a while to be recognised as a competitive method, after good preconditioners started to be identified).

First recall that when A is symmetric, Lanczos gives Q_k, T_k such that $AQ_k = Q_kT_k + q_{k+1}[0, \dots, 0, t_{k+1,k}]$, T_k: tridiagonal.

The idea of CG is as follows: when $A \succ 0$ PD, solve $Q_k^T(AQ_ky - b) = T_ky - Q_k^Tb = 0$, and $x = Q_ky$.
This is known as "Galerkin orthogonality": it imposes that the residual $Ax - b$ is orthogonal to Q_k.

- $T_ky = Q_k^Tb$ is a tridiagonal linear system, so it requires only $O(k)$ operations to solve.
- Three-term recurrence reduces cost to $O(k)$ A-multiplications, making the orthogonalisation cost almost negligible.
- The CG algorithm serendipitously minimises A-norm of error in the Krylov subspace $x_k = \operatorname{argmin}_{x \in Q_k} \|x - x_*\|_A$ ($Ax_* = b$): writing $x_k = Q_ky$, we have
$$\begin{aligned}(x_k - x_*)^TA(x_k - x_*) &= (Q_ky - x_*)^TA(Q_ky - x_*)\\ &= y^T(Q_k^TAQ_k)y - 2b^TQ_ky + b^Tx_*,\end{aligned}$$
and minimiser wrt y (either by completing the squares, or via convexity of the function wrt y) is $y = (Q_k^TAQ_k)^{-1}Q_k^Tb$, so $Q_k^T(AQ_ky - b) = 0$.
  - Note that $\|x\|_A = \sqrt{x^TAx}$ defines a norm (exercise)
  - More generally, for inner-product norm $\|z\|_M = \sqrt{\langle z, z\rangle_M}$, $\min_{x=Qy} \|x_* - x\|_M$ attained when $\langle q_i, x_* - x\rangle_M = 0$, $\forall q_i$ (cf. Part A Numerical Analysis).

We've described the CG algorithm conceptually. To derive the practical algorithm some clever manipulations are necessary. We won't go over them in detail but here is the outcome:
Set $x_0 = 0$, $r_0 = -b$, $p_0 = r_0$ and do for $k = 1, 2, 3, \dots$

$$\begin{aligned}&\alpha_k = \langle r_k, r_k\rangle / \langle p_k, Ap_k\rangle\\ &x_{k+1} = x_k + \alpha_kp_k\\ &r_{k+1} = r_k - \alpha_kAp_k\\ &\beta_k = \langle r_{k+1}, r_{k+1}\rangle / \langle r_k, r_k\rangle\\ &p_{k+1} = r_{k+1} + \beta_kp_k\end{aligned}$$

where $r_k = b - Ax_k$ (residual) and p_k (search direction). x_k is the CG solution after k iterations.

One can show among others (exercise/sheet)

- $\mathcal{K}_k(A,b) = \mathrm{span}(r_0, r_1, \dots, r_{k-1}) = \mathrm{span}(x_1, x_2, \dots, x_k)$ (also equal to $\mathrm{span}(p_0, p_1, \dots, p_{k-1})$)
- $r_j^T r_k = 0$, $j = 0, 1, 2, \dots, k-1$

Thus x_k is kth CG solution, satisfying Galerkin orthogonality $Q_k^T(Ax_k - b) = 0$: residual is orthogonal to the (Krylov) subspace.

13.3 CG convergence

Let's examine the convergence of the CG iterates.

Theorem 13.1 *Let $A \succ 0$ be an $n \times n$ positive definite matrix and $b \in \mathbb{R}^n$. Let $e_k := x_* - x_k$ be the error after the kth CG iteration (x_* is the exact solution $Ax_* = b$). Then*

$$\frac{\|e_k\|_A}{\|e_0\|_A} \le 2\left(\frac{\sqrt{\kappa_2(A)}-1}{\sqrt{\kappa_2(A)}+1}\right)^k.$$

Proof: We have $e_0 = x_*$ ($x_0 = 0$), and

$$\begin{aligned}\frac{\|e_k\|_A}{\|e_0\|_A} &= \min_{x \in \mathcal{K}_k(A,b)} \|x_k - x_*\|_A / \|x_*\|_A \\ &= \min_{p_{k-1} \in \mathcal{P}_{k-1}} \|p_{k-1}(A)b - A^{-1}b\|_A / \|e_0\|_A \\ &= \min_{p_{k-1} \in \mathcal{P}_{k-1}} \|(p_{k-1}(A)A - I)e_0\|_A / \|e_0\|_A \\ &= \min_{p \in \mathcal{P}_k, p(0)=1} \|p(A)e_0\|_A / \|e_0\|_A \\ &= \min_{p \in \mathcal{P}_k, p(0)=1} \left\| V \begin{bmatrix} p(\lambda_1) & & \\ & \ddots & \\ & & p(\lambda_n)\end{bmatrix} V^T e_0 \right\|_A / \|e_0\|_A.\end{aligned}$$

Now $(\text{blue})^2 = \sum_i \lambda_i p(\lambda_i)^2 (V^T e_0)_i^2 \le \max_j p(\lambda_j)^2 \sum_i \lambda_i (V^T e_0)_i^2 = \max_j p(\lambda_j)^2 \|e_0\|_A^2$.

We've shown

$$\frac{\|e_k\|_A}{\|e_0\|_A} \le \min_{p \in \mathcal{P}_k, p(0)=1} \max_j |p(\lambda_j)| \le \min_{p \in \mathcal{P}_k, p(0)=1} \max_{x \in [\lambda_{\min}(A), \lambda_{\max}(A)]} |p(x)|$$

To complete the proof, in the next subsection we will show that

$$\min_{p \in \mathcal{P}_k, p(0)=1} \max_{x \in [\lambda_{\min}(A), \lambda_{\max}(A)]} |p(x)| \le 2\left(\frac{\sqrt{\kappa_2(A)}-1}{\sqrt{\kappa_2(A)}+1}\right)^k. \tag{13}$$

□

- Note that $\kappa_2(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} (=: \frac{b}{a})$. The condition number affects us in terms of the speed here; recall from Section 7 that $\kappa_2(A) \gg 1$ was bad news in terms of accuracy (which persists here too).

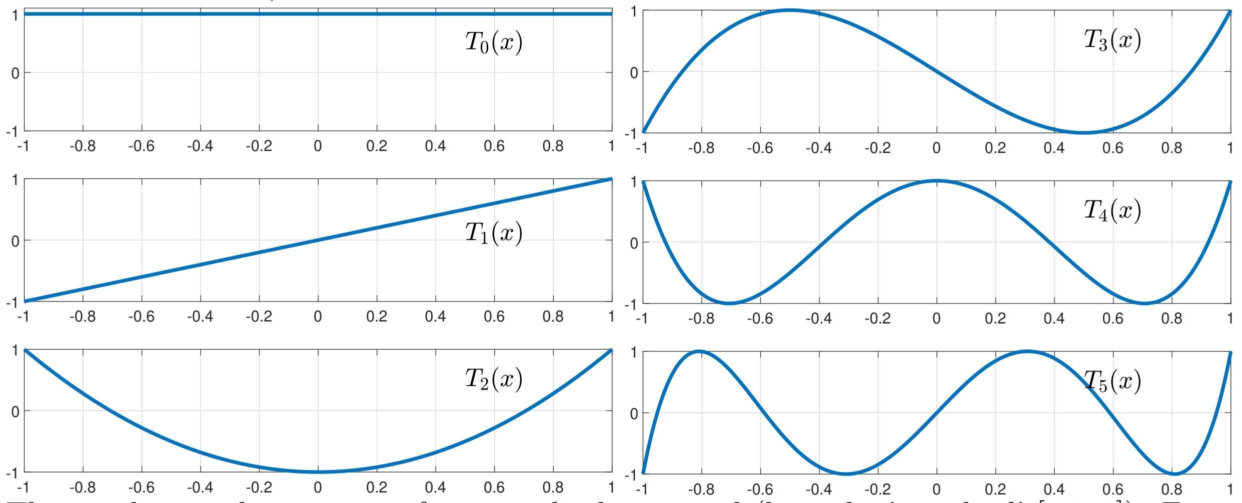
- The above bound is obtained using Chebyshev polynomials on $[\lambda_{\min}(A), \lambda_{\max}(A)]$. This is a class of polynomials that arise in a variety of contexts in computational maths. Let's next look at their properties.

13.3.1 Chebyshev polynomials

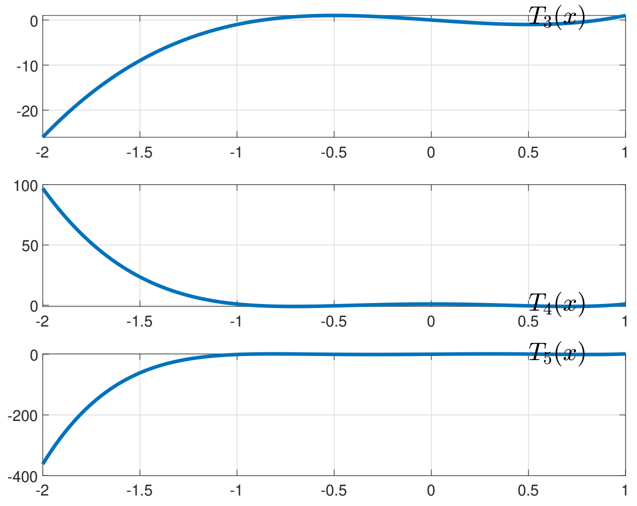
For $z = \exp(i\theta)$, $x = \frac{1}{2}(z + z^{-1}) = \cos\theta \in [-1, 1]$, $\theta = \mathrm{acos}(x)$, define the *Chebyshev* polynomials $T_k(x) = \frac{1}{2}(z^k + z^{-k}) = \cos(k\theta)$. $T_k(x)$ is indeed a polynomial in x:

$$\frac{1}{2}(z+z^{-1})(z^k+z^{-k}) = \frac{1}{2}(z^{k+1}+z^{-(k+1)}) + \frac{1}{2}(z^{k-1}+z^{-(k-1)}) \Leftrightarrow \underbrace{2xT_k(x) = T_{k+1}(x) + T_{k-1}(x)}_{\text{3-term recurrence}}$$

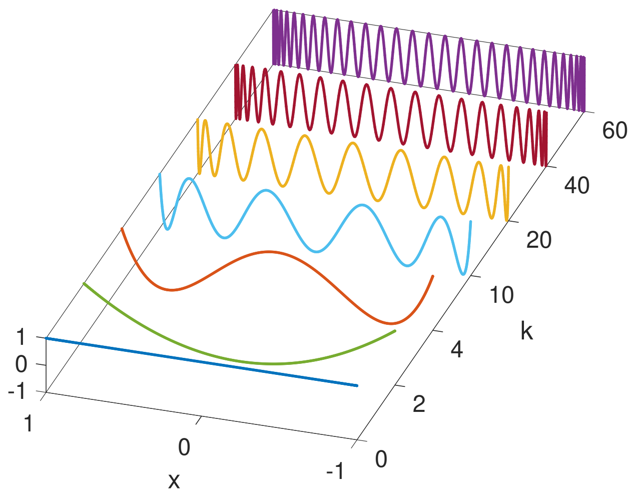
One can argue why Chebyshev polynomials grow rapidly outside $[-1, 1]$ as follows: $T_k(x) = 2^k x^k +$ (polynomial of degree $\leq k-1$), and so $|T_k(x)| = O(2^k x^k)$. We'll derive a more precise estimate (and actual lower bound) below.



These polynomials grow very fast outside the interval (here the 'standard' $[-1, 1]$). For example, plots on $[-2, 1]$ look like



Here's a nice plot of several Chebyshev polynomials:



Put another way, T_k minimises $|\frac{T_k(z)}{T_k(M)}|$ on $z \in [-1, 1]$ for a fixed $|M| > 1$. This is what makes Chebyshev polynomials so useful here.

13.3.2 Properties of Chebyshev polynomials

Chebyshev polynomials have a number of nice properties (see [25], an entire book about them!), which make them an ideal tool in much of numerical computation when working on an interval $[-1, 1]$[23]. We list some of them below.

- Inside $[-1, 1]$, $|T_k(x)| \leq 1$

[23]Of course one can work with a general interval $[a, b]$ via a simple linear transformation. We don't get into this, see Trefethen's book [35] for much more on the beautiful subject of approximation theory.

- Outside $[-1,1]$, $|T_k(x)| \gg 1$ grows rapidly with $|x|$ and k (fastest growth among $p \in \mathcal{P}_k$ with $|p(x)| \le 1$ on $x \in [-1,1]$)
- Symmetry $|T_k(x)| = |T_k(-x)|$ for all $x \in \mathbb{R}$

Consider a shifted+scaled version $p(x) = c_k T_k(\frac{2x-b-a}{b-a})$ where $c_k = 1/T_k(\frac{-(b+a)}{b-a})$. Then $p(0) = 1$, and

- $|p(x)| \le 1/|T_k(\frac{b+a}{b-a})|$ on $x \in [a,b]$
- $T_k(z) = \frac{1}{2}(z^k + z^{-k})$ with $\frac{1}{2}(z+z^{-1}) = \frac{b+a}{b-a} \Rightarrow z = \frac{\sqrt{b/a}+1}{\sqrt{b/a}-1} = \frac{\sqrt{\kappa_2(A)}+1}{\sqrt{\kappa_2(A)}-1}$, so
$$|p(x)| \le \frac{1}{T_k(\frac{b+a}{b-a})} \le 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k.$$

This establishes (13), completing the proof of Theorem 13.1. □

13.4 MINRES: symmetric (indefinite) version of GMRES (nonexaminable)

The content and analysis on MINRES is nonexaminable, although it is really a straightforward mix of GMRES (minimise residual) and CG (simplifications from symmetry). When the matrix is symmetric but not positive definite, GMRES can still be simplified although some care is needed as the Matrix A ceases to define a norm unlike for positive definite matrices.

Symmetric analogue of GMRES: MINRES (minimum-residual method) for $A = A^T$ (but not necessarily $A \succ 0$)
$$x = \mathrm{argmin}_{x \in \mathcal{K}_k(A,b)} \|Ax - b\|_2.$$

Algorithm: Given $AQ_k = Q_{k+1}\tilde{T}_k$ and writing $x = Q_k y$, rewrite as

$$\begin{aligned}\min_y \|AQ_k y - b\|_2 &= \min_y \|Q_{k+1}\tilde{T}_k y - b\|_2 \\ &= \min_y \left\| \begin{bmatrix} \tilde{T}_k \\ 0 \end{bmatrix} y - \begin{bmatrix} Q_k^T \\ Q_{k,\perp}^T \end{bmatrix} b \right\|_2 \\ &= \min_y \left\| \begin{bmatrix} \tilde{T}_k \\ 0 \end{bmatrix} y - \|b\|_2 e_1 \right\|_2, \quad e_1 = [1,0,\dots,0]^T \in \mathbb{R}^n\end{aligned}$$

(where $[Q_k, Q_{k,\perp}]$ orthogonal; same trick as in least-squares)

- Minimised when $\|\tilde{T}_k y - \tilde{Q}_k^T b\| \to \min$; tridiagonal least-squares problem
- Solve via QR (k Givens rotations)+ tridiagonal solve, $O(k)$ in addition to Lanczos

13.4.1 MINRES convergence

As in GMRES, we can examine the MINRES residual in terms of minimising the values of a polynomial at the eigenvalues of A.

$$\min_{x\in\mathcal{K}_k(A,b)}\|Ax-b\|_2=\min_{p_{k-1}\in\mathcal{P}_{k-1}}\|Ap_{k-1}(A)b-b\|_2=\min_{\tilde{p}\in\mathcal{P}_k,\tilde{p}(0)=0}\|(\tilde{p}(A)-I)b\|_2$$
$$=\min_{p\in\mathcal{P}_k,p(0)=1}\|p(A)b\|_2.$$

Since $A=A^T$, A is diagonalisable $A=Q\Lambda Q^T$ with Q orthogonal, so

$$\|p(A)\|_2=\|Qp(\Lambda)Q^T\|_2\leq\|Q\|_2\|Q^T\|_2\|p(\Lambda)\|_2$$
$$=\max_{z\in\lambda(A)}|p(z)|.$$

Interpretation: (again) find polynomial s.t. $p(0)=1$ and $|p(\lambda_i)|$ small

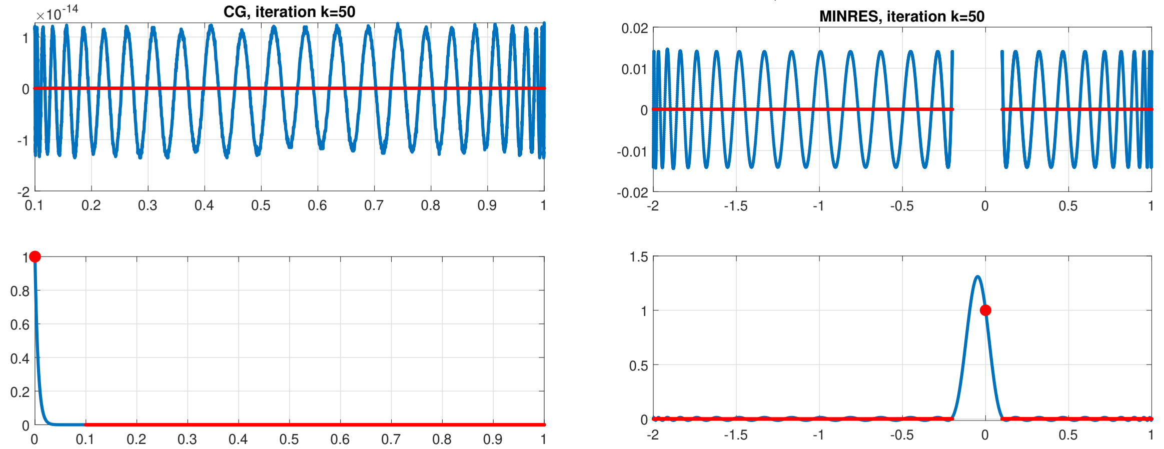
$$\frac{\|Ax-b\|_2}{\|b\|_2}\leq\min_{p\in\mathcal{P}_k,p(0)=1}\max|p(\lambda_i)|.$$

One can prove (nonexaminable)

$$\min_{p\in\mathcal{P}_k,p(0)=1}\max|p(\lambda_i)|\leq 2\left(\frac{\kappa_2(A)-1}{\kappa_2(A)+1}\right)^{k/2}.$$

- Obtained by Chebyshev+Möbius change of variables. See [16] for a full proof.
- Minimisation needed on positive **and** negative sides, hence slower convergence when A is indefinite.

CG and MINRES, optimal polynomials Here are some optimal polynomials for CG and MINRES. Note that $\kappa_2(A)=10$ in both cases; observe how much faster CG is than MINRES (to be clear, this is not because CG is inherently a better algorithm. It is because the problem is harder, as explained in the last bullet point.).



13.5 Preconditioned CG/MINRES

The preceding analysis suggests that the linear system

$$Ax = b, \quad A = A^T (\succ 0)$$

may not converge rapidly with CG or MINRES if the eigenvalues are not distributed in a favorable manner (i.e., clustered away from 0).

In this case, as with GMRES, a workaround is to find a good preconditioner M such that "$M^T M \approx A^{-1}$" and solve

$$M^T AMy = M^T b, \quad My = x$$

As before, desiderata of M:

- $M^T AM$ is easy to multiply to a vector.
- $M^T AM$ has clustered eigenvalues away from 0.

Note that reducing $\kappa_2(M^T AM)$ directly implies rapid convergence.

- It is possible to implement preconditioned CG with just $M^T M$ (no need to find M).

13.6 The Lanczos algorithm for symmetric eigenproblem (nonexaminable)

So far we've discussed Krylov subspace methods for linear systems $Ax = b$; we next consider its use for solving eigenvalue problems $Ax = \lambda x$. The idea is broadly the same: build the Krylov subspace using Arnoldi or Lanczos, and use it to find a solution (i.e., eigenvector) in the subspace.

We are now ready to describe one of the most successful algorithms for large-scale symmetric eigenvalue problems: the Lanczos algorithm. In simple words, it finds an eigenvalue and eigenvector in a Krylov subspace by a projection method, called the Rayleigh-Ritz process.

Algorithm 13.1 Rayleigh-Ritz: given symmetric A and orthonormal Q, find approximate eigenpairs

```
1: Compute $Q^T AQ$.
2: Eigenvalue decomposition $Q^T AQ = V\hat{\Lambda}V^T$.
3: Approximate eigenvalues diag($\hat{\Lambda}$) (Ritz values) and eigenvectors $QV$ (Ritz vectors).
```

This is a **projection** method (similar alg is available for SVD).

Now we can describe the Lanczos algorithm as follows:
Lanczos algorithm=Lanczos iteration+Rayleigh-Ritz

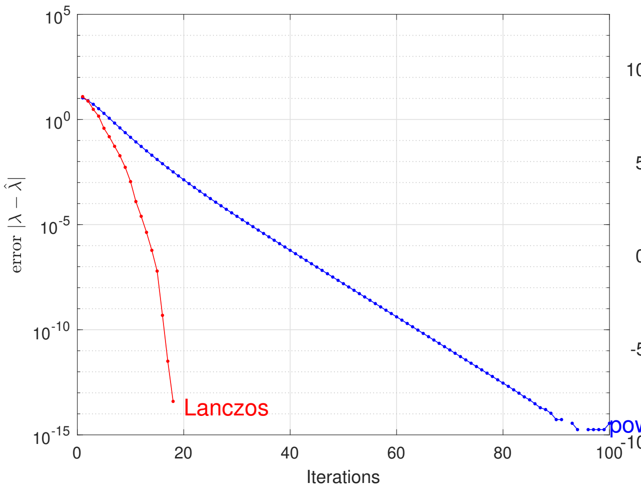
Algorithm 13.2 Lanczos algorithm: $A \in \mathbb{R}^{n\times n}$ find (extremal) eigenpairs.

```
1: Perform Lanczos iterations to obtain $AQ_k = Q_kT_k + q_{k+1}[0,\dots,0,h_{k+1,k}]$.
2: Compute the eigenvalue decomposition of $T_k = V_k\hat{\Lambda}V_k^T$ (Rayleigh-Ritz with subspace $Q_k$).
3: diag($\hat{\Lambda}$) are the approximate eigenvalues (Ritz values), and the columns of $Q_kV_k$ are the approximate eigenvectors (Ritz vectors).
```

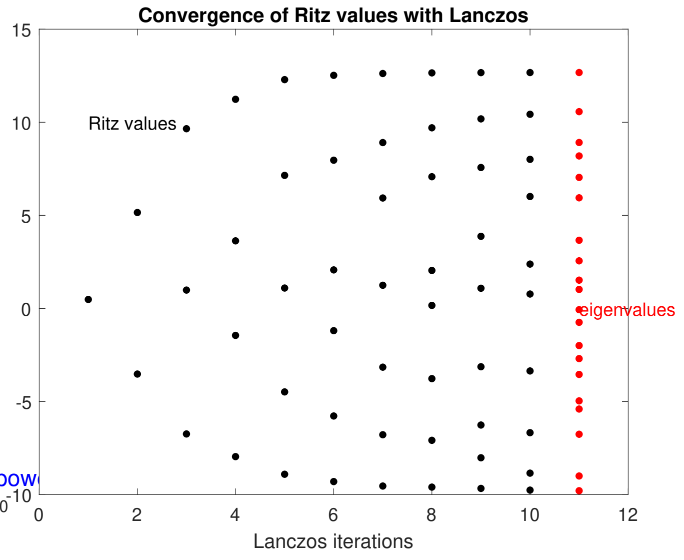
- In this case $Q = Q_k$, so simply $Q_k^TAQ_k = T_k$ (tridiagonal eigenproblem)
- Very good convergence is usually observed to extremal eigenpairs. To see this:
  - Recall from Courant-Fischer $\lambda_{\max}(A) = \max_x \frac{x^TAx}{x^Tx}$
  - Hence $\lambda_{\max}(A) \geq \underbrace{\max_{x\in\mathcal{K}_k(A,b)} \frac{x^TAx}{x^Tx}}_{\text{Lanczos output}} \geq \underbrace{\frac{v^TAv}{v^Tv}, \quad v = A^{k-1}b}_{\text{power method}}$
  - Same for $\lambda_{\min}$, similar for e.g. λ_2

This is admittedly a very crude estimate for the convergence of Lanczos. A thorough analysis turns out to be very complicated; if interested see for example [29].

Experiments with Lanczos Symmetric $A \in \mathbb{R}^{n\times n}, n = 100$, Lanczos/power method with random initial vector b



Convergence to dominant eigenvalue



Convergence of Ritz values (approximate eigenvalues)

The same principles of projecting the matrix onto a Krylov subspace applies to nonsymmetric eigenvalue problems. Essentially it boils down to finding the eigenvalues of the upper Hessenberg matrix H arising in the Arnoldi iteration, rather than the tridiagonal matrix as in Lanczos.

13.7 Arnoldi for nonsymmetric eigenvalue problems (nonexaminable)

Arnoldi for eigenvalue problems: the algorithm, just like Lanczos algorithm, combines Arnoldi iteration+Rayleigh-Ritz.

1. Compute Q^TAQ.
2. Eigenvalue decomposition $Q^TAQ = X\hat{\Lambda}X^{-1}$.
3. Approximate eigenvalues $\text{diag}(\hat{\Lambda})$. (Ritz values) and eigenvectors QX (Ritz vectors).

As in Lanczos, $Q = Q_k = \mathcal{K}_k(A, b)$, so simply $Q_k^TAQ_k = H_k$ (Hessenberg eigenproblem, note that this is ideal for the QR algorithm as the preprocessing step can be skipped).

Which eigenvalues are found by Arnoldi? We give a rather qualitative answer:

- First note that Krylov subspace is invariant under shift: $\mathcal{K}_k(A, b) = \mathcal{K}_k(A - sI, b)$.
- Thus any eigenvector that power method applied to $A - sI$ converges to should be contained in $\mathcal{K}_k(A, b)$.
- To find other (e.g. interior) eigvals, one can use shift-invert Arnoldi: $Q = \mathcal{K}_k((A - sI)^{-1}, b)$.

14 Randomised algorithms in NLA

In this final part of this lecture series we will talk about randomised algorithms. This takes us to the forefront of research in the field: A major idea in numerical linear algebra since 2005 or so has been the use of randomisation, wherein a matrix sketch is used in order to extract information about the huge matrix, for example in order to construct a low-rank approximation.

So far, all algorithms have been deterministic (always same output).

- Direct methods (LU for $Ax = b$, QR alg for $Ax = \lambda x$ or $A = U\Sigma V^T$) are
  - Incredibly reliable, backward stable.
  - Works like magic if $n \lesssim 10000$.
  - But not beyond; cubic complexity $O(n^3)$ or $O(mn^2)$.
- Iterative methods (GMRES, CG, Arnoldi, Lanczos) are
  - Very fast when it works (nice spectrum etc).
  - Otherwise, not so much; need for preconditioning.
- Randomised algorithms

- Output differs at every run.
- Ideally succeed with enormous probability, e.g. $1 - \exp(-cn)$.
- Often by far the fastest&only feasible approach.
- Not for all problems—active field of research.

Why do we need randomisation? Randomised algorithms are somehow attached to a negative connotation that the algorithm is not reliable and always produces different results. Well, this is a valid concern. We hope to remove such concerns in what follows. The reason randomisation is necessary is the sheer size of datasets that we face today, with the rise of data science. As you will see, one core idea of randomisation is that by allowing for a very small error (which can easily be in the order of machine precision) instead of getting an exact solution, one can often dramatically speed up the algorithm.

We'll cover two NLA topics where randomisation has been very successful: **low-rank approximation (randomised SVD)**, and overdetermined **least-squares problems**

14.1 Gaussian matrices

In randomised algorithms we actively introduce a random matrix. For the analysis, Gaussian matrices $G \in \mathbb{R}^{m\times n}$ are the most convenient (not always for computation). These are matrices whose entries are drawn independently from the standard normal (Gaussian) distribution $G_{ij} \sim N(0,1)$. We cannot do justice to random matrix theory (there is a course in Hilary term!), but here is a summary of the properties of Gaussian matrices that we'll be using, namely orthogonal invariance and the Marchenko-Pastur rule, which informally states "rectangular random matrices are well-conditioned".

14.1.1 Orthogonal invariance

A useful fact about Gaussian random matrices G is that its distribution is invariant under orthogonal transformations. That is, if G is Gaussian, so is QG and GQ, where Q is any orthogonal matrix independent of G. To see this (nonexaminable): note that a sum of Gaussian (scalar) random variables is Gaussian, with 0 mean. Recall (non-examinable, but worth knowing if you don't) that the distribution of (multivariate) Gaussian random variables is determined completely by the mean and the (co-)variance. Now let g_i denote the ith column of G. Then the mean is $\mathbb{E}[(Qg_i)] = Q\mathbb{E}[g_i] = 0$. Hence the covariance matrix is $\mathbb{E}[(Qg_i)^T(Qg_i)] = Q\mathbb{E}[g_i^Tg_i]Q^T = QQ^T = I$, so each Qg_i is multivariate Gaussian with the same distribution $N(0,1)$ as g_i. It follows that Qg_i takes iid $N(0,1)$ entries. Independence of Qg_i, Qg_j for $i \neq j$ is immediate.

Another (perhaps quicker) way to see this is to note that the joint pdf of $g_i = [g_{11},\dots,g_{n1}]^T$ is $\frac{1}{(2\pi)^{n/2}}\exp(-\frac{1}{2}(g_{11}^2+\cdots+g_{n1}^2))$, and that of $Qg_i = [\tilde{g}_{11},\dots,\tilde{g}_{n1}]^T$ is (by a change of variables, noting $\det Q = 1$) is also $\frac{1}{(2\pi)^{n/2}}\exp(-\frac{1}{2}(\tilde{g}_{11}^2+\cdots+\tilde{g}_{n1}^2))$.

For much more on random matrices and high-dimensional probability/statistics, we recommend [39, 40], and [37] for matrix concentration inequalities.

14.1.2 Marchenko-Pastur: Rectangular random matrices are well conditioned

Another very useful fact is that when the matrix is rectangular $m > n$ (or $m < n$), the singular values of a Gaussian G where $G_{ij} \sim N(0,1)$ are known to lie in the interval $[\sqrt{m} - \sqrt{n}, \sqrt{m} + \sqrt{n}]$. This is a consequence of the *Marchenko-Pastur* (M-P) rule.

M-P is a classical result in random matrix theory, which we will not be able to prove here (and hence is clearly nonexaminable). We refer those interested to the part C course on Random Matrix Theory. However, understanding the statement and the ability to use this fact is indeed examinable.

The key message is easy to state: a **rectangular random matrix is well conditioned** with extremely high probability. This fact is enormously important and useful in a variety of contexts in computational mathematics.

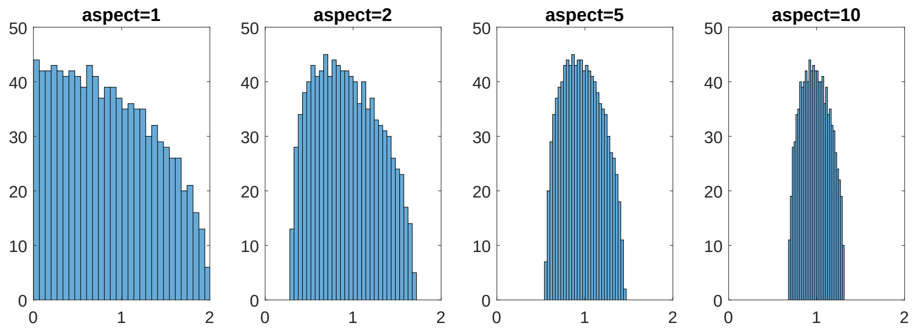
Here is a more precise statement:

Theorem 14.1 (Marchenko-Pastur) *The singular values of random matrix $X \in \mathbb{R}^{m\times n}$ $(m \geq n)$ with iid X_{ij} (mean 0, variance 1) follow the Marchenko-Pastur (M-P) distribution (proof nonexaminable), with density $\sim \frac{1}{x}\sqrt{((\sqrt{m}+\sqrt{n})-x)(x-(\sqrt{m}-\sqrt{n}))}$, and support $[\sqrt{m}-\sqrt{n}, \sqrt{m}+\sqrt{n}]$.*

Proof: omitted, and nonexaminable. Strictly speaking, the theorem concerns the limit $m, n \to \infty$; but the result holds in the nonasymptotic limit with enormous probability. The 'failure' probability that a singular value lies outside the the interval $[\sqrt{m}-\sqrt{n}, \sqrt{m}+\sqrt{n}]$ by t or more is bounded by $\exp(-ct^2)$ for a constant $c > 0$; see [7, Thm II.13] if interested. Here and below, this is what we mean when we say informally that an event holds with high/enormous probability.

Note that the statement does not require X to be Gaussian; it holds for a wide variety of random matrices with independent entries (we'll still mostly stick with Gaussian matrices, as the orthogonal invariance is very attractive for the analysis).

Here is an illustration of the theorem.



Histogram of singular values of random Gaussian matrices with varying aspect ratio m/n.

$\sigma_{\max}(X) \approx \sqrt{m} + \sqrt{n}$, $\sigma_{\min}(X) \approx \sqrt{m} - \sqrt{n}$, hence $\kappa_2(X) \approx \frac{1+\sqrt{n/m}}{1-\sqrt{n/m}} = O(1)$.

As stated above, M-P is a key fact in many breakthroughs in computational maths! Examples include

- Randomised SVD, Blendenpik (randomised least-squares, treated soon)
- (nonexaminable:) Compressed sensing (RIP) [Donoho 06, Candes-Tao 06], Matrix concentration inequalities [Tropp 11], Function approximation by least-squares [Cohen-Davenport-Leviatan 13]
- (nonexaminable:) You might have heard of the Johnson-Lindenstrauss (JL) Lemma. A host (thousands) of papers have appeared that use JL to prove interesting results. It turns out that many of these can be equally well be proven using M-P.

In the remainder, we will use these facts to develop and understand randomised algorithms for (i) least-squares problems, and (ii) low-rank approximation.

14.2 Randomised least-squares

Let's discuss randomised algorithms for least-squares problems that are highly overdetermined, i.e., A is tall-skinny. Throughout this section, we continue to assume $\text{rank}(A) = n$.

$$\min_x \|Ax - b\|_2, \qquad \boxed{A} \in \mathbb{R}^{m\times n},\ m \gg n \tag{14}$$

Let x_* denote the solution $x_* = \text{argmin}_x \|Ax - b\|_2$. First recall that with a traditional method, one either solves the normal equation $x_* = (A^TA)^{-1}A^Tb$ or better yet the thin QR $A = QR$, then $x_* = R^{-1}(Q^Tb)$. Both require $O(mn^2)$ cost, and the latter is stable.

Here we'll look into randomised algorithms, which can have a lower complexity, which means the algorithm can run much faster than traditional methods when m, n are large.

14.3 "Fast" algorithm: row subset selection

To get a sense of why randomisation is needed, let's think of a dirty 'fast' algorithm for highly overdetermined (i.e. $m \gg n$, A is very tall-skinny) least-squares problems, based on row subset selection.

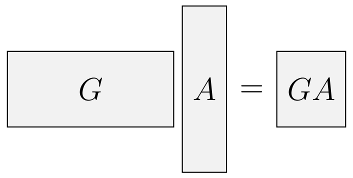
- Let's first suppose that $Ax_* = b$, i.e., $\|Ax_* - b\|_2 = 0$; such overdetermined systems are called *consistent*. Then we see that x_* can be found by solving a smaller problem: x_* is equal to $\hat{x} := \text{argmin}_x \|A_1x - b_1\|_2$ (or $A_1\hat{x} = b_1$), where $A_1 \in \mathbb{R}^{s\times n}$ ($s \geq n$) is any submatrix of A obtained by taking a subset of s rows, and $b_1 \in \mathbb{R}^s$ is the corresponding entries of b. Since we still have $A_1x_* = b_1$, we find the solution x_* as long as this system determines x_* uniquely, that is, when $\text{rank}(A_1) = n$. This suggests that one can try take any s of the rows, for example simply the first $s = n$ rows. If they are linearly independent, we get the solution in $O(sn^2)$ operations, without even looking at the whole matrix!

- The above process can easily fail when $Ax_* \neq b$. Consider, for example, the case where $A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_k \end{bmatrix}$, $b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}$ where $A_1 = \epsilon I_n (\epsilon \ll 1)$, and $A_i = I_n$ for $i \geq 2$, and $b_i = b_j$ if $i, j \geq 2$. Then the least-squares problem has solution $x \approx b_2$ (convince yourself). However, if the top $s = n$ rows are chosen as the row subset, by solving $\min_x \|A_1 x - b_1\|_2$ one gets $\hat{x} = \frac{1}{\epsilon} b_1$; this can be arbitrarily bad!

One would like to avoid such pathological cases, irrespective of how 'badly behaved' A and b are. This is where randomisation comes to the rescue.

14.4 Sketch-and-solve for least-squares problems

In the last problem above where row subset selection failed, one can see that we would have been much better off had we chosen other rows, say the bottom ones. It turns out that there is always a subset selection that gives a good solution. The trouble is, we don't know which! There are ${}_mC_s$ possibilities, and we don't want to try them all.

The goal of randomisation is to mix up the rows so that any subset selection will be good (with enormous probability). The algorithm simply generates a random sketch matrix $G \in \mathbb{R}^{s \times m}$ (which we'll take to be Gaussian for the analysis) with $s = cn (c > 1)$, say $c = 4$, and computes 

which is $s \times n$, and solves the $s \times n$ least-squares problem

$$\min_x \|G(Ax - b)\|_2. \tag{15}$$

To see a connection with the subset selection algorithm, one can think of first left-multiplying a square Gaussian matrix $\hat{G} := \begin{bmatrix} G \\ G_2 \end{bmatrix}$, consider $\min_x \|\hat{G}(Ax - b)\|_2$, and think of selecting the first s rows, which gives (15).

This is an instance of the fundamental idea behind randomised algorithms, called *sketching*: multiply a random matrix to reduce the dimension. Here the matrix G is called the sketching matrix, and it is important that the sketch size s is at least as large as the intrinsic dimension of the problem, which here is n. That is, we require $s = cn$ where $c > 1$, say $s = 2n$.

The algorithm is commonly referred to as a "sketch and solve" approach, as it forms the sketch GA (and Gb), then solves the lower-dimensional problem $\operatorname{argmin}_x \|GAx - Gb\|_2$.

The upshot of sketching is that the solution $\hat{x} = \operatorname{argmin}_x \|G(Ax - b)\|_2$ is a good solution for *any* problem (regardless of A, b), with enormous probability.

Theorem 14.2 (Sketch-and-solve for least-squares) *Let $A \in \mathbb{R}^{m \times n}$ with $m \gg n$ and rank$(A) = n$. Let $x_* = \operatorname{argmin}_x \|Ax - b\|_2$ and $\hat{x} = \operatorname{argmin}_x \|G(Ax - b)\|_2$, where $G \in \mathbb{R}^{s \times m}$*

is a Gaussian matrix with $s > cn, c > 1$. Then with high probability

$$\|A\hat{x} - b\|_2 \leq \frac{\sqrt{s} + \sqrt{n+1}}{\sqrt{s} - \sqrt{n+1}} \|Ax_* - b\|_2 \tag{16}$$

We can verify this claim[24] using the properties of Gaussian matrices in Section 14.1 as follows.

Proof:

- Let $[A, b] = QR$ be the QR factorisation. We have $G[A, b] = (GQ)R$. We can write $\|Av - b\|_2 = \|Qw\|_2$ for some $w \in \mathbb{R}^{n+1}$, and $\|G(Av - b)\|_2 = \|(GQ)w\|_2$.

- The key point is that the GQ matrix is Gaussian, by the orthogonal invariance of Gaussian matrices (Section 14.1.1). Furthermore, GQ is rectangular $s \times (n+1)$, so by M-P, we have $\sigma_i(GQ) \in [\sqrt{s} - \sqrt{n+1}, \sqrt{s} + \sqrt{n+1}]$. Roughly, $\|(GQ)w\|_2 \approx \|Qw\|_2$ for all w.

- It follows that, using the subordinate property $\|XYv\|_2 \leq \|X\|_2 \|Yv\|_2$

  $$\|G(Av - b)\|_2 = \left\| G[A, b] \begin{bmatrix} v \\ -1 \end{bmatrix} \right\|_2 \leq (\sqrt{s} + \sqrt{n+1}) \left\| R \begin{bmatrix} v \\ -1 \end{bmatrix} \right\|_2 = (\sqrt{s} + \sqrt{n+1}) \|Av - b\|_2,$$

  and similarly

  $$\|G(Av - b)\|_2 \geq (\sqrt{s} - \sqrt{n+1}) \|Av - b\|_2.$$

  These hold for any vector v. Since by the definition of $\hat{x}$ we have $\|G(A\hat{x} - b)\|_2 \leq \|G(Ax - b)\|_2$, it follows that

  $$\|A\hat{x} - b\|_2 \leq \frac{1}{\sqrt{s} - \sqrt{n+1}} \|G(Ax - b)\|_2 \leq \frac{\sqrt{s} + \sqrt{n+1}}{\sqrt{s} - \sqrt{n+1}} \|Ax - b\|_2.$$

□

The result shows that by sketch-and-solve, one obtains a solution within $\tau = \frac{\sqrt{s}+\sqrt{n+1}}{\sqrt{s}-\sqrt{n+1}}$ of optimal in terms of minimising the residual. For example, by taking $s = 4(n+1)$, we have $\tau = 3$. If the residual of the original problem can be made small, say $\|Ax_* - b\|_2 = 10^{-10}$, then $\|A\hat{x} - b\|_2 \leq 3 \times 10^{-10}$, giving an excellent least-squares fit. If A is well-conditioned, this also implies the solution x is close to the exact solution.

[24]We are being somewhat imprecise here in terms of the probabilistic statement (this is not a course on probability). A precise statement is that if the prefactor is replaced with $\frac{\sqrt{s}+\sqrt{n+1}+t}{\sqrt{s}-\sqrt{n+1}-t}$, then the probability that the inequality holds is at least $1 - \exp(-t^2/2)$.

Complexity and faster sketches (Nonexaminable) You might have noticed that the sketch-and-solve algorithm isn't faster! The simple-looking computation of GA already requires $O(smn)$ flops, which is at least $O(mn^2)$, the same as the QR-based algorithm discussed in Section 6, which gives the exact solution. What's the point!?

The answer is that we use a structured (but still random) matrix G such that GA can be computed much more efficiently. For example, $G = SFD$ is a popular choice, where F is an FFT matrix, D is random diagonal (e.g. ± 1 with equal probability) and S is a random subsampling matrix. Such matrices can be multiplied to A in $O(mn \log s)$ operations (nonexaminable), so the overall cost of solving (14) becomes $O(mn \log s + sn^2)$. In addition, while the theory above assumed G is Gaussian, in practice very similar behavior is often observed with such structured sketch [24].

14.5 Sketch-to-precondition: Blendenpik

There is even better news: the sketch SA can be used to find an excellent *preconditioner* for an iterative algorithm. This way one can obtain essentially the best solution with residual equal to $\|Ax_* - b\|_2$. The algorithm Blendenpik [1], which takes a "sketch-to-precondition" approach, proceeds as follows. The first step is the same, to sketch the matrix. But the sketch is then used to find a preconditioner, via the QR factorisation.

- Generate random $G \in \mathbb{R}^{4n \times m}$, and $\boxed{\quad G \quad}\boxed{A} = \boxed{\hat{Q}}\boxed{\hat{R}}$

  (QR factorisation), then solve $\min_y \|(A\hat{R}^{-1})y - b\|_2$'s normal equation via Krylov (CG) method. The vector $\hat{x} = \hat{R}^{-1}y$ is then the final solution.

  - The cost is $O(mn \log m + n^3)$ using fast FFT-type transforms[25] for G.
  - Crucially, $A\hat{R}^{-1}$ is well-conditioned. Why? Marchenko-Pastur (next)

The name Blendenpik comes from the fact that the matrix G is 'blending' the rows, and the matrix GA can be seen as picking a submatrix of $\hat{G}A$ as discussed after (15).

14.5.1 Explaining $\kappa_2(A\hat{R}^{-1}) = O(1)$ via Marchenko-Pastur

Let us prove that $\kappa_2(A\hat{R}^{-1}) = O(1)$ with high probability. A key result, once again, is M-P.

Theorem 14.3 *Let $G \in \mathbb{R}^{s \times m}$ ($s = cn, c > 1$) be a Gaussian matrix and let* $\boxed{\quad G \quad}\boxed{A} =$

[25]The FFT (fast Fourier transform) is one of the important topics that we can't treat properly—for now just think of it as a matrix-vector multiplication that can be performed in $O(n \log n)$ flops rather than $O(n^2)$.)

$\boxed{\hat{Q}}\boxed{\hat{R}}$ *(QR factorisation). Then $A\hat{R}^{-1}$ is well-conditioned with high probability.*

Let's prove this for $G \in \mathbb{R}^{4n\times m}$ Gaussian, and for concreteness assuming $s = 4n$:

Proof: Let $A = QR$. Then $GA = (GQ)R =: \tilde{G}R$

- $\boxed{\tilde{G}}$ is $4n \times n$ rectangular Gaussian (by orthogonal invariance), hence well-conditioned.
- So by M-P, $\kappa_2(\tilde{R}^{-1}) = O(1)$ where $\tilde{G} = \tilde{Q}\tilde{R}$ is the QR factorisation.
- Thus $GA = \tilde{G}R = (\tilde{Q}\tilde{R})R = \tilde{Q}(\tilde{R}R) = \tilde{Q}\hat{R}$, so $\hat{R}^{-1} = R^{-1}\tilde{R}^{-1}$.
- Hence $A\hat{R}^{-1} = (QR)(R^{-1}\tilde{R}^{-1}) = Q\tilde{R}^{-1}$, so $\kappa_2(A\hat{R}^{-1}) = \kappa_2(\tilde{R}^{-1}) = O(1)$.

$\square$

14.5.2 Blendenpik: solving $\min_x \|Ax - b\|_2$ using $\hat{R}$

We have seen that $\kappa_2(A\hat{R}^{-1}) =: \kappa_2(B) = O(1)$; defining $\hat{R}x = y$, $\min_x \|Ax - b\|_2 = \min_y \|(A\hat{R}^{-1})y - b\|_2 = \min_y \|By - b\|_2$.

- B is well-conditioned$\Rightarrow$in normal equation
$$B^TBy = B^Tb \tag{17}$$
B^TB is also well-conditioned $\kappa_2(B^TB) = O(1)$; so positive definite and well-conditioned.

- Thus we can solve (17) via CG (or LSQR [30], a more stable variant in this context; nonexaminable)
  - The convergence theory for CG implies it would achieve exponential convergence, so convergence in $O(1)$ iterations! (or $O(\log\frac{1}{\epsilon})$ iterations for ϵ accuracy)
  - each iteration requires $w \leftarrow Bw$ and $w \leftarrow B^Tw$. It is important to note that these can be computed without computing $A\hat{R}^{-1}$ explicitly. For example, to $w \leftarrow Bw$ we first do $w \leftarrow \hat{R}^{-1}w$ ($n\times n$ triangular solve), then $w \leftarrow Aw$ ($m\times n$ matrix-vector multiplication). This takes $O(mn)$ cost overall.

14.5.3 Blendenpik experiments

Let's illustrate our findings. Since Blendenpik finds a preconditioner $\hat{R}^{-1}$ such that $A\hat{R}^{-1}$ is well-conditioned (regardless of $\kappa_2(A)$), we expect the convergence of CG to be independent of $\kappa_2(A)$. This is indeed what we see here, in Figure 3.

In practice, Blendenpik is observed to achieve $\approx \times 5$ speedup over classical (Householder-QR based) method when $m \gg n$.

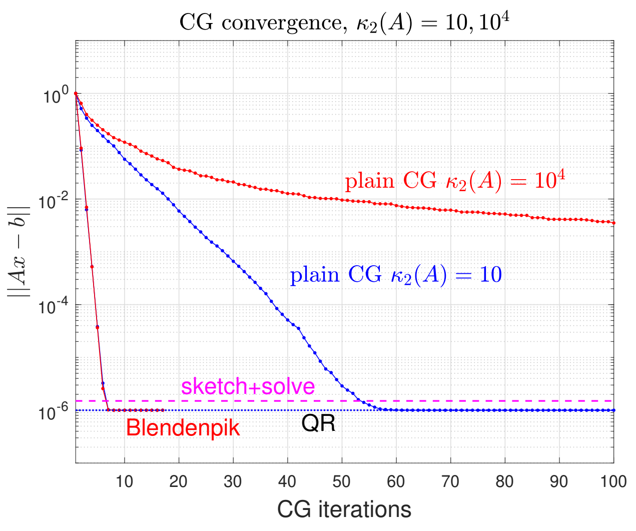


Figure 3: Solving $\min_x \|Ax - b\|_2$ via CG for $A^TAx = A^Tb$ vs. Blendenpik $(AR^{-1})^T(AR^{-1})x = (AR^{-1})^Tb$, $m = 10000, n = 100$. The matrix A is generated to have a specified condition number (10 or 10^4), then we set a unit norm vector $b = Ax + e$ where $\|e\|_2 = 10^{-6}$. Solution with sketch-and-solve (Section 14.4) is also shown, which gets a (suboptimal but good) solution without iterations. The Blendenpik solutions have the same quality as a classical, QR-based method.

15 Randomised algorithms for low-rank approximation

The final topic we treat is low-rank approximation, where the goal is to devise fast algorithms for finding an approximation to the truncated SVD. This is arguably the very topic that made randomised algorithms get appreciated by researchers and practicioners as a compelling approach to solve important NLA problems. This course starts with the SVD, and ends with it too.

15.1 Randomised SVD by Halko-Martinsson-Tropp

We start with what has been arguably the most successful usage of randomisation in NLA, low-rank approximation. Probably the best reference is the paper by Halko, Martinsson and Tropp (HMT) [18]. This paper has been enormously successful, with almost 4000 Google Scholar citations as of 2022. See also the recent survey by the same authors [24].

The algorithm itself is astonishingly simple:

Algorithm 15.1 Randomised SVD (HMT): given $A \in \mathbb{R}^{m\times n}$ and rank r, find a rank-r approximation $\hat{A} \approx A$.

```
1: Form a random matrix $G \in \mathbb{R}^{n\times r}$, usually $r \ll n$.
2: Compute $AG$.
3: Compute the QR factorisation $AG = QR$.
4: $A \approx Q \; Q^TA$ $(= (QU_0)\Sigma_0 V_0^T)$ is a rank-$r$ approximation.
```

Here, G is a random matrix taking independent and identically distributed (iid) entries. A convenient choice (for the theory, not necessarily for computation) is a Gaussian matrix, with iid entries $G_{ij} \sim N(0,1)$.

To gain insight, suppose that $\operatorname{rank}(A) = r$, that is, $A = U_r\Sigma_r V_r^T$ where $U \in \mathbb{R}^{m\times r}$. Then $AG = U_r(\Sigma_r V_r^T G)$, where $(\Sigma_r V_r^T G)$ is nonsingular with probability 1 (nonexaminable but I hope this is convincing), and its conditioning improves if G is fatter. Then Q spans exactly $\operatorname{span}(U_r)$, and hence $QQ^TA = U_rU_r^TA = A$. Usually $\operatorname{rank}(A) > r$, but surprisingly, the HMT algorithm still behaves robustly.

Here are some properties of the HMT algorithm:

- $O(mnr)$ cost for dense A. This is much lower than the $O(mn^2)$ for the SVD!
- Near-optimal approximation guarantee:

  Theorem 15.1 *For any $\hat{r} < r-1$,*

  $$\mathbb{E}\|A-\hat{A}\|_F \le \sqrt{1+\frac{\hat{r}}{r-\hat{r}-1}}\,\|A-A_{\hat{r}}\|_F,$$

  where $A_{\hat{r}}$ is the rank $\hat{r}$-truncated SVD (expectation w.r.t. random matrix G).

  This is a remarkable result; make sure to pause and think about what it says! The approximant $\hat{A}$ has error $\|A-\hat{A}\|_F$ that is within a factor $\sqrt{1+\frac{\hat{r}}{r-\hat{r}-1}}$ of the optimal truncated SVD, for a slightly lower rank $\hat{r} < r$ (think, say, $\hat{r} = \lfloor 0.9r \rfloor$).

  Goal: understand this, or at least why $\mathbb{E}\|A-\hat{A}\| = O(1)\|A-A_{\hat{r}}\|$.

Pseudoinverse To understand why the HMT algorithm works, we need to introduce the pseudoinverse of rectangular matrices.

Given $M \in \mathbb{R}^{m\times n}$ with economical SVD $M = U_r\Sigma_r V_r^T$ $(U_r \in \mathbb{R}^{m\times r}, \Sigma_r \in \mathbb{R}^{r\times r}, V_r \in \mathbb{R}^{n\times r}$ where $r = \operatorname{rank}(M)$ so that $\Sigma_r \succ 0)$, the **pseudoinverse** $M^\dagger$ is

$$M^\dagger = V_r\Sigma_r^{-1}U_r^T \in \mathbb{R}^{n\times m}.$$

- $M^\dagger$ satisfies $MM^\dagger M = M$, $M^\dagger MM^\dagger = M^\dagger$, $MM^\dagger = (MM^\dagger)^T$, $M^\dagger M = (M^\dagger M)^T$ (these are often taken to be the definition of the pseudoinverse—the above definition is much simpler IMO).

- $M^\dagger = M^{-1}$ if M nonsingular.

- $M^\dagger M = I_n$ if $m \geq n$ ($MM^\dagger = I_m$ if $m \leq n$) and M is full rank.

- Given a full-rank underdetermined system $\boxed{\quad A \quad}\boxed{x} = \boxed{b}$ where $A \in \mathbb{R}^{m\times n}$ with $m < n$ and $r = m$, the general solution is $x = A^\dagger b + V_{r,\perp}z$ for arbitrary z, and minimum-norm soln is $x = A^\dagger b$.

Below we're going to use the pseudoinverse for rectangular matrices of full (column or row) rank.

15.2 HMT approximant: analysis (down from 70 pages!)

We are now in a position to explain why the HMT algorithm works. The original HMT paper is over 70 pages with long analysis. Here we attempt to condense the arguments to the essence (and with different proofs). We first give a qualitative argument, then make it more precise.

Qualitative argument First, If $A = \boxed{U_1}\boxed{\Sigma_1}\boxed{\quad V_1^T \quad} + E$ ($\|E\|$ small), then $\boxed{AX} = \boxed{U_1}\boxed{\Sigma_1}\boxed{V_1^T X} + EX$. Note that $V_1^T X$ is Gaussian if X is, and rectangular. So by M-P $\|(V_1^T X)^\dagger\| = O(1)$. Right-multiply $(V_1^T X)^\dagger V_1^T$ to get $\boxed{AX}\boxed{(V_1^T X)^\dagger}\boxed{\quad V_1^T \quad} + \tilde{E} = U_1\Sigma_1 V_1^T + \tilde{E} \approx A$. Hence $\mathrm{Range}(A) \subsetneq \mathrm{Range}(AX)$. HMT then computes the best, orthogonal projection onto $\mathrm{Range}(AX) = \mathrm{Range}(Q)$.

More precise argument Let us be more precise. Recall that our low-rank approximation is $\hat{A} = QQ^T A$, where $AG = QR$. Goal: $\|A - \hat{A}\| = \|(I_m - QQ^T)A\| = O(\|A - A_{\hat{r}}\|)$.

1. $QQ^TAG = AG$ (QQ^T is **orthogonal projector** onto span(AG)). Hence $(I_m - QQ^T)AG = 0$, so $A - \hat{A} = (I_m - QQ^T)A(I_n - GM^T)$ for any $M \in \mathbb{R}^{n\times r}$.

   The idea then is to choose M cleverly so that the expression $(I_m - QQ^T)A(I_n - GM^T)$ can be shown to have small norm.

2. Set $M^T = (V_1^TG)^\dagger V_1^T$ where $V_1 = [v_1, \dots, v_{\hat{r}}] \in \mathbb{R}^{n\times\hat{r}}$ is the top right singular vectors of A ($\hat{r} \le r$). Recall that $\hat{r}$ is any integer bounded by r.

3. $V_1V_1^T(I - GM^T) = V_1V_1^T(I - G(V_1^TG)^\dagger V_1^T) = 0$ if V_1^TG full row-rank (this is a generic assumption that holds with probability 1 if G is Gaussian; note that then V_1^TG is also Gaussian), so $A - \hat{A} = (I_m - QQ^T)A(I - V_1V_1^T)(I_n - GM^T)$.

4. Taking norms yields $\|A - \hat{A}\|_2 = \|(I_m - QQ^T)A(I - V_1V_1^T)(I_n - GM^T)\|_2 = \|(I_m - QQ^T)U_2\Sigma_2V_2^T(I_n - GM^T)\|_2$ where $[V_1, V_2]$ is orthogonal (and $A = [U, U_2]\begin{bmatrix}\Sigma & \\ & \Sigma_2\end{bmatrix}[V_1, V_2]^T$ is the SVD), so

$$\|A - \hat{A}\|_2 \le \|\Sigma_2\|_2\|(I_n - GM^T)\|_2 = \underbrace{\|\Sigma_2\|_2}_{\text{optimal rank-}\hat{r}} \quad \|GM^T\|_2$$

It remains to prove $\|GM^T\|_2 = O(1)$. To see why this should hold with high probability, we again invoke the Marchenko-Pastur rule.

$\|GM^T\|_2 = O(1)$ Recall that we've shown for $M^T = (V_1^TG)^\dagger V_1^T$ where $G \in \mathbb{R}^{n\times r}$ is random, that

$$\|A - \hat{A}\|_2 \le \|\Sigma_2\|_2\|(I_n - GM^T)\|_2 = \underbrace{\|\Sigma_2\|_2}_{\text{optimal rank-}\hat{r}} \quad \|GM^T\|_2.$$

Now $\|GM^T\|_2 = \|G(V_1^TG)^\dagger V_1^T\|_2 = \|G(V_1^TG)^\dagger\|_2 \le \|G\|_2\|(V_1^TG)^\dagger\|_2$.

Now let's analyse the (standard) case where G is random Gaussian $G_{ij} \sim \mathcal{N}(0, 1)$. Then

- By the orthogonal invariance of Gaussian matrices (Sec. 14.1.1) V_1^TG is another Gaussian matrix hence $\|(V_1^TG)^\dagger\| = 1/\sigma_{\min}(V_1^TG) \lesssim 1/(\sqrt{r} - \sqrt{\hat{r}})$ by M-P.
- $\|G\|_2 \lesssim \sqrt{n} + \sqrt{r}$ by M-P.

Together we get $\|GM^T\|_2 \lesssim \frac{\sqrt{n}+\sqrt{r}}{\sqrt{r}-\sqrt{\hat{r}}} =$ "$O(1)$".

Remark:

- When G is a non-Gaussian random matrix, the performance is similar, but is harder to analyze. A popular choice is again the so-called SRFT matrices, which use the FFT (fast Fourier transform) and can be applied to A with $O(mn\log m)$ cost rather than $O(mnr)$.

15.3 Precise analysis for HMT (nonexaminable)

A slightly more elaborate analysis again using random matrix theory will give us a very sharp bound on the expected value of the error $E_{\mathrm{HMT}} =: A - \hat{A}$. (this again is non-examinable).

To understand below, we need the notion of projectors. A square matrix $P \in \mathbb{R}^{n\times n}$ is called a **projector** if $P^2 = P$.

- P is always diagonalisable and all eigenvalues are 1 or 0. To see this (nonexaminable): consider the Jordan decomposition of P, and see that the eigenvalues $1, 0$ cannot have a Jordan block of size > 1.
- $\|P\|_2 \geq 1$ and $\|P\|_2 = 1$ iff $P = P^T$; in this case P is called an orthogonal projector, and P can be written as $P = QQ^T$ where Q is orthonormal.
- One can easily show that $I - P$ is another projector as $(I-P)^2 = I - P$, and unless $P = 0$ or $P = I$, we have $\|I - P\|_2 = \|P\|_2$:
Schur form $QPQ^* = \begin{bmatrix} I & B \\ 0 & 0 \end{bmatrix}$, $Q(I-P)Q^* = \begin{bmatrix} 0 & -B \\ 0 & I \end{bmatrix}$; See Szyld 2006 [34] for many more about projectors.

Theorem 15.2 *[Reproduces HMT 2011 Thm.10.5] If G is Gaussian, for any $\hat{r} < r - 1$,* $\mathbb{E}\|E_{\mathrm{HMT}}\|_F \leq \sqrt{\mathbb{E}\|E_{\mathrm{HMT}}\|_F^2} = \sqrt{1 + \frac{\hat{r}}{r-\hat{r}-1}}\|A - A_{\hat{r}}\|_F$.

Proof: First ineq: Cauchy-Schwarz. Defining $G(V_1^TG)^\dagger V_1^T =: \mathcal{P}_{G,V_1}$ (this is a projector as $\mathcal{P}_{G,V_1}^2 = \mathcal{P}_{G,V_1}$), $\|E_{\mathrm{HMT}}\|_F^2$ is

$$\begin{aligned}
&\|A(I - V_1V_1^T)(I - \mathcal{P}_{G,V_1})\|_F^2 = \|A(I - V_1V_1^T)\|_F^2 + \|A(I - V_1V_1^T)\mathcal{P}_{G,V_1}\|_F^2 \\
&= \|\Sigma_2\|_F^2 + \|\Sigma_2\mathcal{P}_{G,V_1}\|_F^2 = \|\Sigma_2\|_F^2 + \|\Sigma_2(V_2^TG)(V_1^TG)^\dagger V_1^T\|_F^2.
\end{aligned}$$

Now if G is Gaussian then $V_2^TG \in \mathbb{R}^{(n-\hat{r})\times r}$ and $V_1^TG \in \mathbb{R}^{\hat{r}\times r}$ are independent Gaussian. Hence by [HMT Prop. 10.1] $\mathbb{E}\|\Sigma_2(V_2^TG)(V_1^TG)^\dagger\|_F^2 = \frac{\hat{r}}{r-\hat{r}-1}\|\Sigma_2\|_F^2$, so

$$\mathbb{E}\|E_{\mathrm{HMT}}\|_F^2 = \left(1 + \frac{\hat{r}}{r - \hat{r} - 1}\right)\|\Sigma_2\|_F^2.$$

□

Note how remarkable the theorem is—the 'lazily' computed approximant is nearly optimal up to a factor $\sqrt{1 + \frac{r}{r-\hat{r}-1}}$ for a near rank $\hat{r}$ (one can take, e.g. $\hat{r} = 0.9r$).

15.4 Generalised Nyström (nonexaminable)

Let us briefly mention an algorithm for low-rank approximation that is even faster than HMT, especially when $r \gg 1$.

Let $X \in \mathbb{R}^{n\times r}$ be a random matrix, say Gaussian as before; and set another random matrix $Y \in \mathbb{R}^{n\times(r+\ell)}$, and [Nakatsukasa arXiv 2020 [27]]

$$\hat{A} = \boxed{(AX(Y^TAX)^\dagger Y^T)A} = \mathcal{P}_{AX,Y}A.$$

Then $\hat{A}$ is another rank-r approximation to A, and $A-\hat{A} = (I-\mathcal{P}_{AX,Y})A = (I-\mathcal{P}_{AX,Y})A(I-XM^T)$; choose M s.t. $XM^T = X(V_1^TX)^\dagger V_1^T = \mathcal{P}_{X,V_1}$. Then $\mathcal{P}_{AX,Y}, \mathcal{P}_{X,V_1}$ are (nonorthogonal) projectors, and

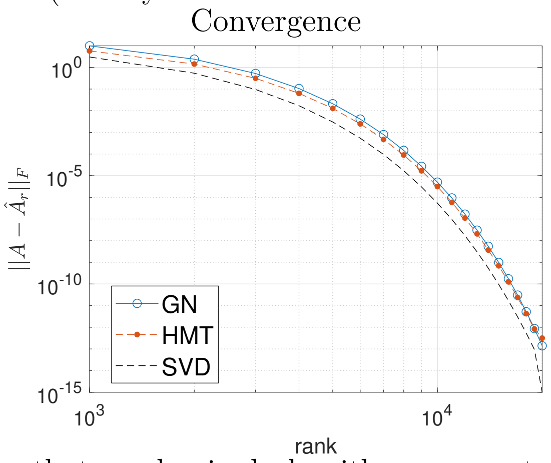
$$\begin{aligned} \|A-\hat{A}\| &= \|(I-\mathcal{P}_{AX,Y})A(I-\mathcal{P}_{X,V_1})\| \\ &\le \|(I-\mathcal{P}_{AX,Y})A(I-V_1V_1^T)(I-\mathcal{P}_{X,V_1})\| \\ &\le \|A(I-V_1V_1^T)(I-\mathcal{P}_{X,V_1})\| + \|\mathcal{P}_{AX,Y}A(I-V_1V_1^T)(I-\mathcal{P}_{X,V_1})\|. \end{aligned}$$

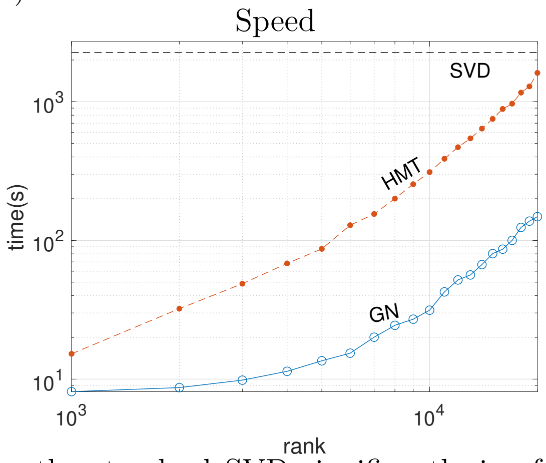
- Note that the $\|A(I-V_1V_1^T)(I-\mathcal{P}_{X,V_1})\|$ term is the exact same as in the HMT error.
- Extra term $\|\mathcal{P}_{AX,Y}\|_2 = O(1)$ as before if $c > 1$ in $Y \in \mathbb{R}^{m\times cr}$ (again, by Marchenko-Pastur).
- Overall, about $(1+\|\mathcal{P}_{AX,Y}\|_2) \approx (1+\frac{\sqrt{n}+\sqrt{r+\ell}}{\sqrt{r+\ell}-\sqrt{r}})$ times bigger expected error than HMT (this bound can be improved by a more careful analysis, similar to Thm. 15.2), **still near-optimal** and **much faster** $O(mn\log n + r^3)$.

Here's some experiments with HMT and GN (generalised Nyström).

Let $A = U\Sigma V_1^T$ be a dense $30,000\times 30,000$ matrix with geometrically decaying singular values σ_i, and choose U, V_1 to be random (orthogonal factors of a random Gaussian matrix).

We use HMT and GN to find low-rank approximations to A varying the rank r. We also compare with MATLAB's SVD, which gives the optimal truncated SVD (up to numerical errors, which are $O(10^{-15})$ so invisible). Here the sketch matrices $G, X(,Y)$ are SRFT matrices (as they run faster than Gaussian matrices).





We see that randomised algorithms can outperform the standard SVD significantly in efficiency, while (unless the truncated SVD is really the goal) the slight suboptimality in accuracy is almost always not an issue.

15.5 MATLAB code

Implementing the HMT and GN algorithms (at least with Gaussian matrices, which don't always give optimal speed performance) is very easy!

Here's some sample MATLAB code. We can set up a matrix with exponentially decaying singular values (the matrix is constructed by computing $A = U\Sigma V^T$, where U, V are orthonormal and Σ is a geometric sequence from 10^{-100} to 1).

```
n = 1000; % size
A = gallery('randsvd',n,1e100); % geometrically decaying singvals
r = 200;  % rank
```

Then to do HMT as follows:

```
X = randn(n,r);
AX = A*X;
[Q,R] = qr(AX,0); % QR fact.
At = Q*(Q'*A);
```

which with high probability gives me an excellent approximation

```
norm(At-A,'fro')/norm(A,'fro')
ans = 1.2832e-15
```

And for Generalised Nyström :

```
X = randn(n,r); Y = randn(n,1.5*r);
AX = A*X;  YA = Y'*A;  YAX = YA*X;
[Q,R] = qr(YAX,0);  % stable pseudo-inverse via the QR factorisation
At = (AX/R)*(Q'*YA);

norm(At-A,'fro')/norm(A,'fro')
ans = 2.8138e-15
```

Both algorithms give an excellent low-rank approximation to A.

15.6 Randomised algorithm for $Ax = b$, $Ax = \lambda x$?

We have seen that randomization can be very powerful for low-rank approximation and least-squares problems. What about the two core problems in NLA, linear systems $Ax = b$ and eigenvalue problems $Ax = \lambda x$? A recent paper [28] proposes the use of sketching (as described above) for GMRES (for $Ax = b$) and Rayleigh-Ritz (for eigenproblems) combined with a generation of a basis for a Krylov subspace that is not orthonormal. The resulting algorithm can be surprisingly efficient, but does not always work. This is certainly not the end of the story.

Randomised algorithms promise to be employed in more and more applications and problems. We will almost surely see more breakthroughs in randomised NLA. Who be the inventors? Would the young and fresh brains like to take a shot?

16 Conclusion and discussion

We have tried to give a good overview of the field of numerical linear algebra in this course. However a number of topics have been omitted completely due to lack of time. Here is an incomplete list of other topics.

16.1 Important (N)LA topics not treated

These are clearly nonexaminable but definitely worth knowing if you want to get seriously into the field. These will be discussed in lecture if and only if time permits, and in any case only superficially.

- tensors [Kolda-Bader 2009 [23]]
- FFT (values↔coefficients map for polynomials) [e.g. Golub and Van Loan 2012 [15]]
- sparse direct solvers [Duff, Erisman, Reid 2017 [11]]
- multigrid [e.g. Elman-Silvester-Wathen 2014 [12]]
- fast (Strassen-type) matrix multiplication etc [Strassen 1969 [33]+many follow-ups]
- functions of matrices [Higham 2008 [20]]
- generalised, polynomial/nonlinear eigenvalue problems [Guttel-Tisseur 2017 [17]]
- perturbation theory (Davis-Kahan [8] etc) [Stewart-Sun 1990 [32]]
- compressed sensing (this deals with $Ax = b$ where A is very 'fat') [Foucart-Rauhut 2013 [13]]
- model order reduction [Benner-Gugercin-Willcox 2015 [4]]
- communication-avoiding algorithms [e.g. Ballard-Demmel-Holtz-Schwartz 2011 [2]]
- Other topics in randomised NLA: trace estimation, CUR, Kaczmarz methods, etc.

16.2 Course summary

This information is provided for MSc students who take two separate exams based on the 1st and 2nd halfs.

1st half

- SVD and its properties (Courant-Fischer etc), applications (low-rank)
- Direct methods (LU) for linear systems and least-squares problems (QR)
- Stability of algorithms

- Basics of eigenvalues

2nd half

- Direct method (QR algorithm) for eigenvalue problems, SVD
- Krylov subspace methods for linear systems (GMRES, CG) and eigenvalue problems (Arnoldi, Lanczos)
- Randomised algorithms for SVD and least-squares

16.3 Related courses you can take

Courses with significant intersection with NLA include

- C7.7 Random Matrix Theory: for theoretical underpinnings of Randomised NLA
- C6.4 Finite Element Method for PDEs: NLA arising in solutions of PDEs
- C6.2 Continuous Optimisation: NLA in optimisation problems

and many more: differential equations, data science, optimisation, machine learning,... NLA is everywhere in computational mathematics.

Thank you for your interest in NLA!

References

[1] H. Avron, P. Maymounkov, and S. Toledo. Blendenpik: Supercharging LAPACK's least-squares solver. *SIAM J. Sci. Comp.*, 32(3):1217–1236, 2010.

[2] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.*, 32(3):866–901, 2011.

[3] J. Banks, J. Garza-Vargas, and N. Srivastava. Global convergence of Hessenberg shifted QR I: Exact arithmetic. *Found. Comput. Math.*, pages 1–34, 2024.

[4] P. Benner, S. Gugercin, and K. Willcox. A survey of projection-based model reduction methods for parametric dynamical systems. *SIAM Rev.*, 57(4):483–531, 2015.

[5] R. Bhatia. *Positive Definite Matrices.* Princeton University Press, 2009.

[6] K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. Part II: Aggressive early deflation. *SIAM J. Matrix Anal. Appl.*, 23:948–973, 2002.

[7] K. R. Davidson and S. J. Szarek. Local operator theory, random matrices and banach spaces. *Handbook of the geometry of Banach spaces*, 1(317-366):131, 2001.

[8] C. Davis and W. M. Kahan. The rotation of eigenvectors by a perturbation. III. *SIAM J. Numer. Anal.*, 7(1):1–46, 1970.

[9] J. Demmel. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, USA, 1997.

[10] J. Dongarra and F. Sullivan. Guest editors' introduction: The top 10 algorithms. *IEEE Computer Architecture Letters*, 2(01):22–23, 2000.

[11] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, 2017.

[12] H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics.* Oxford University Press, USA, 2014.

[13] S. Foucart and H. Rauhut. *A Mathematical Introduction to Compressive Sensing.* Springer, 2013.

[14] D. F. Gleich. Pagerank beyond the web. *SIAM Rev.*, 57(3):321–363, 2015.

[15] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 4th edition, 2012.

[16] A. Greenbaum. *Iterative Methods for Solving Linear Systems.* SIAM, Philadelphia, PA, USA, 1997.

[17] S. Güttel and F. Tisseur. The nonlinear eigenvalue problem. *Acta Numer.*, 26:1–94, 2017.

[18] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.*, 53(2):217–288, 2011.

[19] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, PA, USA, second edition, 2002.

[20] N. J. Higham. *Functions of Matrices: Theory and Computation.* SIAM, Philadelphia, PA, USA, 2008.

[21] R. A. Horn and C. R. Johnson. *Topics in Matrix Analysis.* Cambridge University Press, 1991.

[22] R. A. Horn and C. R. Johnson. *Matrix Analysis.* Cambridge University Press, second edition, 2012.

[23] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, 2009.

[24] P.-G. Martinsson and J. A. Tropp. Randomized numerical linear algebra: Foundations and algorithms. *Acta Numer.*, pages 403—572, 2020.

[25] J. C. Mason and D. C. Handscomb. *Chebyshev Polynomials.* CRC Press, 2010.

[26] M. F. Murphy, G. H. Golub, and A. J. Wathen. A note on preconditioning for indefinite linear systems. *SIAM J. Sci. Comp.*, 21(6):1969–1972, 2000.

[27] Y. Nakatsukasa. Fast and stable randomized low-rank matrix approximation. arXiv:2009.11392.

[28] Y. Nakatsukasa and J. A. Tropp. Fast and accurate randomized algorithms for linear systems and eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 45(2):1183–1214, 2024.

[29] C. C. Paige. Error analysis of the Lanczos algorithm for tridiagonalizing a symmetric matrix. *IMA J. Appl. Math.*, 18(3):341–349, 1976.

[30] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Soft.*, 8(1):43–71, 1982.

[31] Y. Saad and M. H. Schultz. GMRES - A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7(3):856–869, 1986.

[32] G. W. Stewart and J.-G. Sun. *Matrix Perturbation Theory (Computer Science and Scientific Computing).* Academic Press, 1990.

[33] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[34] D. B. Szyld. The many proofs of an identity on the norm of oblique projections. *Numerical Algorithms*, 42(3-4):309–323, 2006.

[35] L. N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition.* SIAM, 2019.

[36] L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[37] J. A. Tropp. An introduction to matrix concentration inequalities. *Foundations and Trends® in Machine Learning*, 8(1-2):1–230, 2015.

[38] M. Udell and A. Townsend. Why are big data matrices approximately low rank? *SIAM Journal on Mathematics of Data Science*, 1(1):144–160, 2019.

[39] R. Vershynin. *High-dimensional probability: An introduction with applications in data science*, volume 47. Cambridge University Press, 2018.

[40] M. J. Wainwright. *High-dimensional statistics: A non-asymptotic viewpoint*, volume 48. Cambridge University Press, 2019.