

Statistics and Numerical Methods, Tsinghua University

Numerical Linear Algebra

Xuening Bai (白雪宁)

Institute for Advanced Study (IASTU) & Department of Astronomy (DoA)



清華大學

Tsinghua University

Sep. 29, 2024

What is numerical linear algebra?

- The primary task is to numerically solve linear system of equations: $Ax=b$

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N = b_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N = b_3$$

$$\cdots \qquad \cdots$$

$$a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N = b_M$$

- Eigenvalue problem
- Linear regression
- Factorization/decomposition (e.g., QR, SVD)

It's all about matrix manipulations by scientific computing.

Why linear algebra?

- Systems of linear equations are probably the simplest mathematical model encountered in computational science.
- Matrix is usually the format how data are presented.

- Extremely wide range of applications:

Data analysis and statistics

Scientific and engineering computing

Not just for linear models, but also heavily employed in non-linear systems of equations, optimization, ODEs, PDEs, etc.

Image processing

Machine learning (e.g., convolution)

...

Feature-1	Feature-2	Feature-3	Feature-4	Feature-n	
x_1^1	x_2^1	x_3^1	x_4^1	x_n^1	Sample-1
x_1^2	x_2^2	x_3^2	x_4^2	x_n^2	Sample-2
x_1^3	x_2^3	x_3^3	x_4^3	x_n^3	Sample-3
...	
x_1^m	x_2^m	x_3^m	x_4^m	x_n^m	Sample-m

Why numerical linear algebra?

- Solving $Ax=b$ and/or other matrix problems numerically can be much trickier than you may naively think!
 - Matrix can get large (and sparse), and need algorithms to make computation **efficient** (in time, **memory** and for **parallelization**).
 - Matrix can be ill-conditioned, need algorithms to be **stable** and robust.
 - Computation unavoidably involves errors (e.g. floating point roundoff), which need to be well controlled to guarantee sufficient **accuracy**.
- There are matured libraries/packages, but you have to know how they work and choose what you need.
 - Different algorithms are designed for different types of matrices.
 - Different algorithms trade off on different factors, whose performance can vastly vary in terms of efficiency, accuracy, etc.

Standard linear algebra packages

- LINPACK

A collection of Fortran subroutines for linear algebra, developed in 1970-1980s for supercomputers of that time, now largely obsolete.

[LINPACK Benchmark](#) (now used to build the TOP500 list):

Measure of a system's floating point computing power by solving a dense n by n system of linear equations $Ax=b$.

Conducted using the HPL (High-performance Linpack) package.

- (Sca)LAPACK (Linear Algebra PACKage)

Successor of LINPACK designed to run on modern machines with cache-based architectures. Written in Fortran, while also available in other languages.

- Many other libraries/packages/software

GNU scientific library, NAG, MKL, Sundials libraries, LIS, HyPre, PETSc
NumPy/SciPy, Julia, Matlab, Scilab, Maple, Mathematica

Solvability

Solve a set of linear algebra equations (N unknowns related by M equations):

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \dots & \dots & \dots & \dots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_M \end{pmatrix} \Rightarrow \mathbf{A} \mathbf{x} = \mathbf{b}$$

For $M > N$, the system is usually **over-determined** (e.g., linear regression)

For $M < N$, the system is **under-determined**, though progress can still be made.

Here **we mostly consider $M=N$** (the system is closed).

Unique solution requires $\text{rank}(\mathbf{A}) = N$.

However, the system can be degenerate due to round-off error, and the accumulated round-off error in the numerical procedure may swamp the solution.

A naive approach

The general solution is given by **Cramer's rule**:

$$x_i = \frac{\det A^{(i)}}{\det A} \quad \text{where } A^{(i)} \text{ is obtained by replacing the } i^{\text{th}} \text{ column of } A \text{ by } \mathbf{b}.$$

But this is not practical for $N > 3$, and is not very stable.

It requires $O(N \cdot N!)$ operations for large N !

That means $\sim 10^{160}$ operations for $N=100$, or 10^{140} years on a teraflop computer.
Clearly we need something better.

General methodology

Methods can be classified into two types:

- **Direct methods**: get exact answer (apart from round-off error) within finite number of operations.
- **Iterative methods**: provide better and better approximate solutions with increasing work.

For small N , direct methods are usually the best.

For large matrices (say $10^6 \times 10^6$), iterative methods may be the only viable choice.

Outline

- Solving linear systems: direct methods
- Sensitivity analysis
- QR factorization
- Solving linear systems: classical iterative methods
- Solving linear systems: conjugate gradient methods

Gaussian-elimination

System of equations	Row operations	Augmented matrix
$2x + y - z = 8$ $-3x - y + 2z = -11$ $-2x + y + 2z = -3$		$\left[\begin{array}{ccc c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array} \right]$
$2x + y - z = 8$ $\frac{1}{2}y + \frac{1}{2}z = 1$ $2y + z = 5$	$L_2 + \frac{3}{2}L_1 \rightarrow L_2$ $L_3 + L_1 \rightarrow L_3$	$\left[\begin{array}{ccc c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array} \right]$
$2x + y - z = 8$ $\frac{1}{2}y + \frac{1}{2}z = 1$ $-z = 1$	$L_3 + -4L_2 \rightarrow L_3$	$\left[\begin{array}{ccc c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array} \right]$

Upper triangular matrix

Pivot elements

Backward substitution

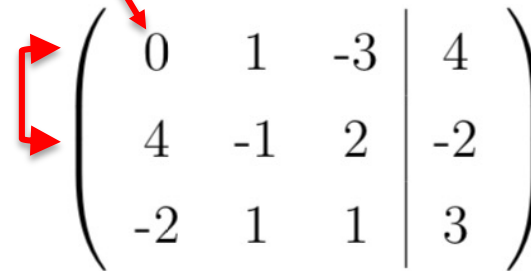
$\begin{aligned} 2x + y - z &= 8 \\ \frac{1}{2}y + \frac{1}{2}z &= 1 \\ -z &= 1 \end{aligned}$	$L_3 + -4L_2 \rightarrow L_3$	$\left[\begin{array}{ccc c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array} \right]$
$\begin{aligned} 2x + y &= 7 \\ \frac{1}{2}y &= \frac{3}{2} \\ -z &= 1 \end{aligned}$	$\begin{aligned} L_2 + \frac{1}{2}L_3 &\rightarrow L_2 \\ L_1 - L_3 &\rightarrow L_1 \end{aligned}$	$\left[\begin{array}{ccc c} 2 & 1 & 0 & 7 \\ 0 & \frac{1}{2} & 0 & \frac{3}{2} \\ 0 & 0 & -1 & 1 \end{array} \right]$
$\begin{aligned} 2x + y &= 7 \\ y &= 3 \\ z &= -1 \end{aligned}$	$\begin{aligned} 2L_2 &\rightarrow L_2 \\ -L_3 &\rightarrow L_3 \end{aligned}$	$\left[\begin{array}{ccc c} 2 & 1 & 0 & 7 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array} \right]$
$\begin{aligned} x &= 2 \\ y &= 3 \\ z &= -1 \end{aligned}$	$\begin{aligned} L_1 - L_2 &\rightarrow L_1 \\ \frac{1}{2}L_1 &\rightarrow L_1 \end{aligned}$	$\left[\begin{array}{ccc c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array} \right]$

Pivoting

If we start with a matrix like this:

$$\begin{array}{rcl} y - 3z & = & 4 \\ 4x - y + 2z & = & -2 \\ -2x + y + z & = & 3 \end{array}$$

Pivot element


$$\left(\begin{array}{ccc|c} 0 & 1 & -3 & 4 \\ 4 & -1 & 2 & -2 \\ -2 & 1 & 1 & 3 \end{array} \right)$$

Gaussian elimination fails if pivot element=0...

There is no guarantee the diagonal elements are always non-zero during elimination.

Solution: interchanging the rows or rows/columns to select the largest element.

Partial Full
pivoting pivoting

Gaussian elimination without pivoting is unstable to round-off errors.

Usually, partial pivoting is almost as good as full pivoting.

Gaussian-elimination with pivoting

$$\begin{array}{rcl}
 \begin{array}{l} y - 3z = 4 \\ 4x - y + 2z = -2 \\ -2x + y + z = 3 \end{array} & \begin{array}{c} \text{red arrows} \\ \text{swap rows 1 and 2} \end{array} & \begin{array}{l} 4x - y + 2z = -2 \\ y - 3z = 4 \\ -2x + y + z = 3 \end{array} \\
 & \begin{array}{c} \text{yellow arrow} \\ \text{elimination} \end{array} & \begin{array}{l} 4x - y + 2z = -2 \\ y - 3z = 4 \\ \frac{1}{2}y + 2z = 2 \end{array} \\
 & \begin{array}{c} \text{yellow arrow} \\ \text{elimination} \end{array} & \begin{array}{l} 4x - y + 2z = -2 \\ y - 3z = 4 \\ \frac{7}{2}z = 0 \end{array}
 \end{array}$$

$$\begin{pmatrix} 0 & 1 & -3 & | & 4 \\ 4 & -1 & 2 & | & -2 \\ -2 & 1 & 1 & | & 3 \end{pmatrix} \xrightarrow{\text{yellow arrow}} \begin{pmatrix} 4 & -1 & 2 & | & -2 \\ 0 & 1 & -3 & | & 4 \\ -2 & 1 & 1 & | & 3 \end{pmatrix} \xrightarrow{\text{yellow arrow}} \begin{pmatrix} 4 & -1 & 2 & | & -2 \\ 0 & 1 & -3 & | & 4 \\ 0 & 1/2 & 2 & | & 2 \end{pmatrix} \xrightarrow{\text{yellow arrow}} \begin{pmatrix} 4 & -1 & 2 & | & -2 \\ 0 & 1 & -3 & | & 4 \\ 0 & 0 & 7/2 & | & 0 \end{pmatrix}$$

This is **partial (row) pivoting**.

Full pivoting, one would permute columns 2 and 3 at last step.

Backward substitution remains the same.

Matrix operations in Gaussian elimination

Each step in the Gaussian method with (row) pivoting is equivalent to multiplying a matrix of the following types **from left**:

$$E = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix}$$

Multiply row 1 by a factor 2 and add to row 3.
(elimination)

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Interchange row 2 and row 3.
(permutation)

$$S = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rescale row 2 by a factor of 1/3.
(scaling)

Therefore, the process of solving $Ax=b$ becomes:

$$(S_3 S_2 S_1 E_5 E_4 E_3 E_2 P_2 E_1 P_1) A x = (S_3 S_2 S_1 E_5 E_4 E_3 E_2 P_2 E_1 P_1) b$$

backward Elimination w.
substitution pivoting



Computational cost (floating point operations)

For an $N \times N$ matrix, the cost of Gaussian elimination + backward substitution is

Permutation: $2(N+1) + 2N + \dots + 2 \cdot 3 \sim N^2$

Elimination: $(N+1) \cdot (N-1) + N \cdot (N-2) + \dots + 3 \cdot 1 \sim N^3/3$

Scaling: $N + (N-1) + \dots + 1 \sim N^2/2$

Backward substitution: $(N-1) + (N-2) + \dots + 1 \sim N^2/2$



$\sim N^3/3$

To invert an matrix, can let \mathbf{b} be an identity matrix, cost increases to $\sim N^3$.

This is about the best one can do for a dense matrix, but with one limitation:

The right hand side vector \mathbf{b} must be known in advance.

To repetitively solve $A\mathbf{x}=\mathbf{b}$ for different \mathbf{b} s, need more efficient algorithms.

LU decomposition

We already see that backward substitution, which is to solve an equation:

$$U\mathbf{x} = \mathbf{y}$$

takes only $\sim N^2$ operations (not N^3) with U being an **upper triangular matrix**.

Similarly, for a **lower triangular matrix** L , the cost of solving $L\mathbf{y} = \mathbf{b}$, which is called forward substitution, is also $\sim N^2$.

For an arbitrary matrix A , if we can decompose it into $A = LU$

Then the task of solving $A\mathbf{x} = \mathbf{b}$ becomes a two-stage process:

$$A\mathbf{x} = (LU)\mathbf{x} = L(U\mathbf{x}) = \mathbf{b} \quad \left\{ \begin{array}{l} L\mathbf{y} = \mathbf{b} \\ U\mathbf{x} = \mathbf{y} \end{array} \right.$$

This would substantially reduce the cost if one were to do it repetitively.

LU decomposition: procedures

Feasibility:

The process of Gaussian elimination already yields:

$$MA = U \quad \text{so that} \quad A = M^{-1}U$$

Without pivoting, one can easily show that M and M^{-1} are lower triangular matrices.

With pivoting, one can obtain, $A = PLU$, which is essentially the same.

LU decomposition is essentially a generalization of Gaussian elimination.

LU decomposition: procedures

In practice, use *Crout's algorithm*:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

First, one trivially has $\beta_{11} = a_{11}$.

Next, one solves $\alpha_{21} = a_{21}/\beta_{11}$, $\alpha_{31} = a_{31}/\beta_{11}$, ..., $\alpha_{n1} = a_{n1}/\beta_{11}$.

Once knowing α_{21} , we can obtain $\beta_{12} = a_{12}$ $\beta_{22} = a_{22} - \alpha_{21}\beta_{12}$

Then, one can solve for

$$\alpha_{i2} = (a_{i2} - \alpha_{i1}\beta_{12})/\beta_{22} \quad (i=3,\dots,n)$$

This process continues...

Total cost: $\sim N^3/3$

LU decomposition: pivoting

In practice, use *Crout's algorithm*:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Compare among a_{i1} , and choose (exchange rows) the largest to become a_{11} .

Assign $\beta_{11} = a_{11}$, $\alpha_{i1} = a_{i1}/\beta_{11}$ ($i > 1$) .

Compute $\beta'_{i2} = (a_{i2} - \alpha_{i1}\beta_{12})$ ($i \geq 2$)

Choose (exchange rows) the largest for β_{22} .

Then, one can solve for $\alpha_{i2} = (a_{i2} - \alpha_{i1}\beta_{12})/\beta_{22}$.

This process continues...

For storage: the L and U matrices can be stored in a single matrix array.

Original matrix elements can be replaced one by one (no extra storage needed!).

Cholesky decomposition

If A is **symmetric** (more generally Hermitian) and **positive definite**, that is

$$A^T = A, \text{ and } \mathbf{x}^T A \mathbf{x} > 0 \text{ (for any } \mathbf{x} \in \mathbb{R}^n \text{)}$$

it turns out pivoting is unnecessary.

Moreover, LU decomposition can be cast into a special form (**Cholesky decomposition**):

$$A = LL^T \quad \text{where } L \text{ is a lower triangular matrix.}$$

Note: the symmetric and positive definite nature of A ensures $l_{ii} > 0$.

Once this is done, the task of solving $A\mathbf{x}=\mathbf{b}$ becomes: $L\mathbf{y} = \mathbf{b}$, $L^T \mathbf{x} = \mathbf{y}$

Cholesky decomposition: procedures

Basic task is to solve:

$$\begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ l_{N1} & l_{N2} & \dots & l_{NN} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & \dots & l_{N1} \\ 0 & l_{22} & \dots & l_{N2} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & l_{NN} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \dots & \dots & \dots & \dots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{pmatrix}$$

Method:

$$l_{11}^2 = a_{11} ,$$

$$l_{21} = a_{21}/l_{11} , \quad l_{31} = a_{31}/l_{11} , \quad \dots \quad l_{N1} = a_{N1}/l_{11} ,$$

$$l_{22}^2 = a_{22} - l_{21}^2$$

$$l_{32} = (a_{32} - l_{31}l_{21})/l_{22} , \quad \dots \quad l_{N2} = (a_{N2} - l_{N1}l_{21})/l_{22} ,$$

Cost: $\sim N^3/6 + N$ square roots
(\sim half of LU decomposition).

This process continues...

General applications

Solving linear equations.

Calculating the determinant:

$$\det(A) = (-1)^{n_p} \prod_{j=1}^n \beta_{jj} \quad (\text{general matrix. } n_p: \# \text{ of permutations})$$

$$\det(A) = \prod_{j=1}^n l_{jj}^2 \quad (A \text{ is symmetric, positive-definite})$$

Matrix inversion:

Do LU decomposition, then solve $LUX = I$. (general matrix)

Do Cholesky decomposition, then solve $LL^T X = I$. (A is symmetric, positive-definite)

Iterative solvers for non-linear equations.

Example: the least mean square problem

Data of n observations: $\{y_i, x_{ij} |_{j=1}^p\}_{i=1}^n$

Linear regression model:

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i,$$

$\log(\text{Re})$ 1.4 $\log(\sigma_0)$ 0.36 μ_B

The problem is *overdetermined* ($n > p$).

Would like to find β_j so as to minimize ε_i .

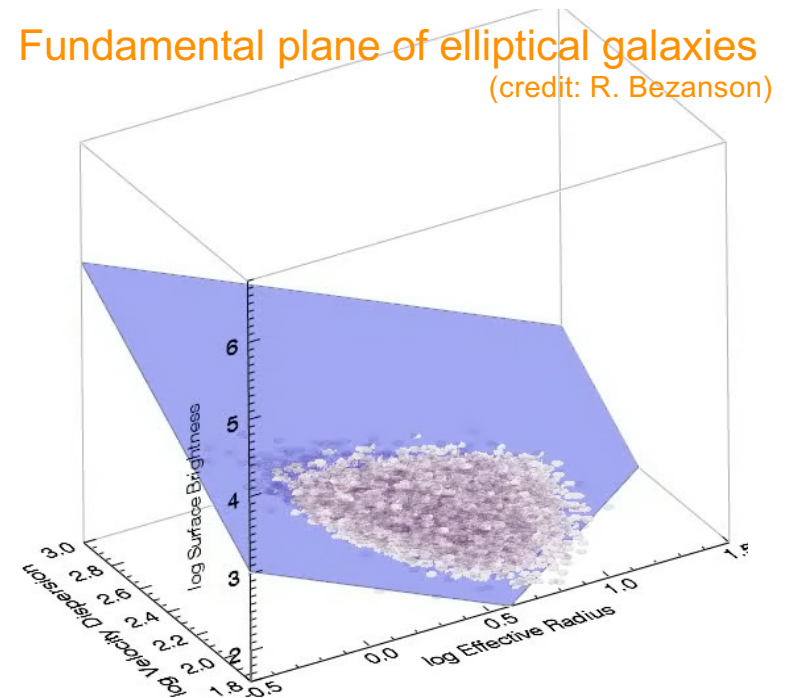
In matrix notation:
$$\begin{matrix} & \uparrow & \uparrow & \uparrow \\ X & \beta & = & y - \epsilon \\ \begin{matrix} n \times p & p & n \end{matrix} & & & \end{matrix}$$

Mathematically, can show the least mean square problem is equivalent to solving:

$$\underline{(X^T X)} \beta = X^T y .$$

$p \times p$ matrix, symmetric, positive-semidefinite.

Can be solved by Cholesky decomposition



Tridiagonal matrix

Often times, numerical algorithms result in a **tridiagonal matrix** of unknowns (e.g., finite difference methods for PDEs), which is a special form of **sparse matrix**:

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \dots & \dots & \\ & & & \dots & \dots \\ & & & & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & & & a_N & b_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{N-1} \\ d_N \end{pmatrix}$$

Tridiagonal matrices are special because:

- Direct solution is fast and only takes $O(N)$ operations.
- Only 3 vectors are needed to store the matrix.
- Pivoting not needed provided it is diagonally dominant, which is often the case.

Cyclic tridiagonal matrix

Real applications more often yield **cyclic tridiagonal matrix** (e.g., from cubic spline interpolation):

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \dots & & & \\ & & & \dots & & \\ & & & & \dots & \\ & & a_{N-1} & b_{N-1} & c_{N-1} & \\ & & & a_N & b_N & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{N-1} \\ d_N \end{pmatrix}$$

Solution: first invert the tridiagonal matrix, then use **Sherman-Morris formula** to make “corrections”

$$(A + \mathbf{u}\mathbf{v}^T)^{-1} = A^{-1} + \frac{[A^{-1}\mathbf{u}][\mathbf{v}^T A^{-1}]^T}{1 + \mathbf{v}^T A^{-1}\mathbf{u}}$$

Here, can choose u, v to have non-zero values only in 1st and last element.

Outline

- Solving linear systems: direct methods
- Sensitivity analysis
- QR factorization
- Solving linear systems: classical iterative methods
- Solving linear systems: conjugate gradient methods

Sensitivity analysis

Numerical solution unavoidably contains errors.

Goal: assess how accurate our numerical method can be.

Two perspectives:

1. Given $Ax = b$,

How wrong it can be if I'm solving: $(A + \delta A)x = b + \delta b$

2. If I have obtained an approximate solution x_0 , how much should I improve the solution (how sensitive the system is to slightly incorrect answers δx)?

First, we need a measure to describe the “length” of a vector and a matrix, as well as the difference between two vectors/matrices.

Vector norms

A **vector norm** is a function $\| \cdot \| : \mathbb{R}^n \rightarrow [0, \infty)$ satisfying:

- 1) $\|x\| = 0$ if and only if $x = 0$; (separate points)
- 2) $\|cx\| = c\|x\|$ for all scalars c and vectors x ; (scalability)
- 3) $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in \mathbb{R}^n$. (triangle inequality)

Most commonly, we define **p-norm** as $\|\vec{x}\|_p \equiv (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}$.

In particular, we have:

$$\begin{array}{ll} \text{2-norm:} & \|\vec{x}\|_2 \equiv \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \\ \text{1-norm:} & \|\vec{x}\|_1 \equiv \sum_{k=1}^n |x_k|. \\ \text{\(\infty\)-norm:} & \|\vec{x}\|_\infty \equiv \max(|x_1|, |x_2|, \cdots, |x_n|). \end{array}$$

All these norms are mathematically “equivalent”, and can be chosen as needed.

Matrix norms

A **matrix norm** is a function $\| \cdot \| : \mathbb{R}^{m \times n} \rightarrow [0, \infty)$, satisfying:

- 1) $\|A\| = 0$ if and only if $A=0$; (positivity)
- 2) $\|cA\| = c\|A\|$ for all scalars c and matrices A ; (scalability)
- 3) $\|A + B\| \leq \|A\| + \|B\|$ for any A, B ; (triangle inequality)
- 4) For square matrix, $\|AB\| \leq \|A\|\|B\|$ for any A, B .

For instance, the *Frobenius norm*: $\|A\|_{\text{Fro}} \equiv \sqrt{\sum_{i,j} a_{i,j}^2}$

Similarly, all matrix norms are mathematically “equivalent”.

However, a more meaningful definition should be consistent with vector norms.

Matrix norms

The matrix norm for $A \in \mathbb{R}^{m \times n}$ induced by a vector norm is defined as:

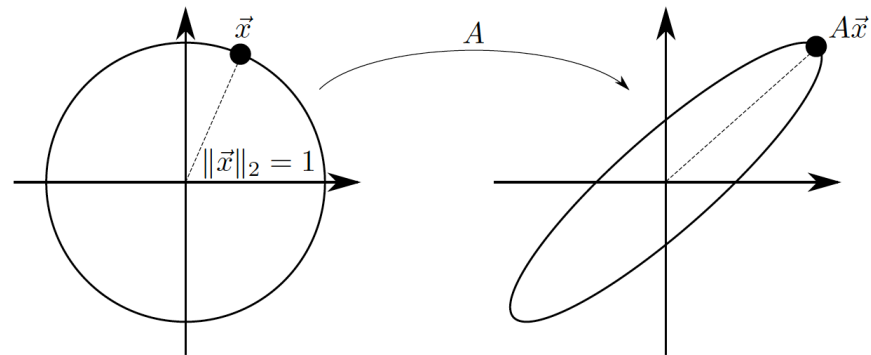
$$\|A\| \equiv \max\{\|A\mathbf{x}\| : \mathbf{x} \in \mathbb{R}^n \text{ with } \|\mathbf{x}\| = 1\}.$$

Without proof, we show:

1-norm: $\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|.$ (maximum absolute column sum)

∞ -norm: $\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|.$ (maximum absolute row sum)

2-norm: $\|A\|_2 = \sqrt{\lambda_{\max}(A^T A)}$
(spectral norm)



Ill-conditioned problem

Example:
$$\begin{pmatrix} 2.0002 & 1.9998 \\ 1.9998 & 2.0002 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \end{pmatrix} \quad \text{Solution: } (x_1, x_2) = (1, 1)$$

Now suppose there is an error on the RHS, with $\delta \mathbf{b} = 2.0 \times 10^{-4} (1, -1)$,

$$\begin{pmatrix} 2.0002 & 1.9998 \\ 1.9998 & 2.0002 \end{pmatrix} \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{pmatrix} = \begin{pmatrix} 4.0002 \\ 3.9998 \end{pmatrix} \quad \text{Solution: } (x_1, x_2) = (1.5, 0.5)$$

We see that
$$\frac{\|\delta \mathbf{x}\|_\infty}{\|\mathbf{x}\|_\infty} = \frac{1}{2}, \quad \frac{\|\delta \mathbf{b}\|_\infty}{\|\mathbf{b}\|_\infty} = \frac{1}{20000}$$

A slight change in \mathbf{b} dramatically changes the solution!

This usually happens when the matrix A is close to being singular.

Condition number

Let us systematically look at how sensitive the solution is subject to errors.

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

We obtain: $\delta \mathbf{x} = (A + \delta A)^{-1}(\delta \mathbf{b} - \delta A \mathbf{x}) = (I + A^{-1} \delta A)^{-1} A^{-1}(\delta \mathbf{b} - \delta A \mathbf{x})$

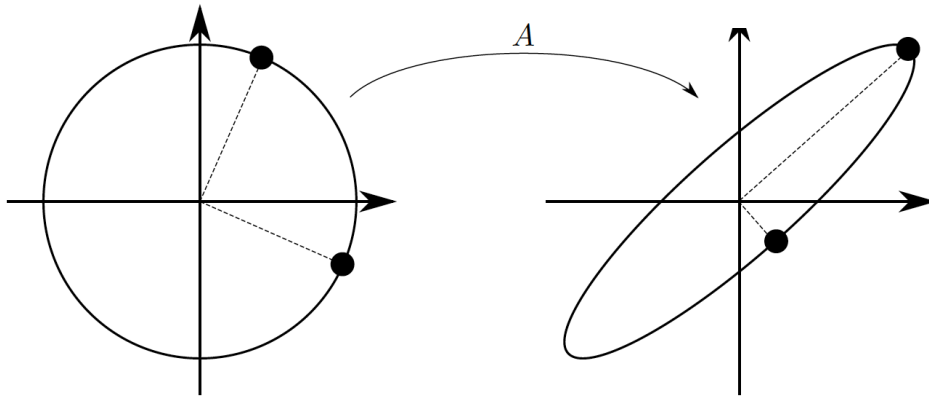
Taking the norm on both sides, one can obtain (after some algebra): $\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\|A^{-1}\| \|A\|}{1 - \|A^{-1}\| \|\delta A\|} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|} \right).$

Define the **condition number** as $\text{cond } A \equiv \|A^{-1}\| \|A\|$

We essentially have $\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \text{cond } A \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|} \right).$

Condition number

Physical meaning (by definition): $\text{cond } A = \left(\max_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} \right) \left(\min_{\mathbf{y} \neq 0} \frac{\|A\mathbf{y}\|}{\|\mathbf{y}\|} \right)^{-1}$



For 2-norm, it is essentially the ratio of largest to smallest eigenvalues.

Larger cond A means A is closer to singular.

How to compute cond A?

We need to compute its inverse.

However, an ill-conditioned matrix is difficult to invert (close to singularity).

Using methods that compute eigenvalues (e.g., SVD).

Improving your solution

Solving a linear equation involves $\sim N^3$ operations, final error can be \gg roundoff.

To improve the solution, consider the residual: $\mathbf{r} \equiv \mathbf{b} - A\hat{\mathbf{x}} = A(\mathbf{x} - \hat{\mathbf{x}})$

We can further solve $A\delta\mathbf{x} = \mathbf{r}$.

On the other hand, we can show:
$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \|A^{-1}\| \|A\| \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$$

As long as $\text{cond } A$ is not too large, this procedure can greatly improve the precision. However, it does little help for ill-conditioned matrices.

Back to least mean square problem

The problem is reduced to solving: $A^T A \mathbf{x} = A^T \mathbf{b}$

Suppose A is a square matrix (for now), we have

$$\begin{aligned}\text{cond } (A^T A) &= \|A^T A\| \|(A^T A)^{-1}\| \\ &\approx \|A^T\| \|A\| \|A^{-1}\| \|(A^T)^{-1}\| && \text{(for many choices of matrix norms)} \\ &= (\text{cond } A)^2\end{aligned}$$

If A is ill-conditioned, then $A^T A$ becomes much much worse!

Outline

- Solving linear systems: direct methods
- Sensitivity analysis
- **QR factorization**
- Solving linear systems: classical iterative methods
- Solving linear systems: conjugate gradient methods

From rows to columns

When a matrix is ill-conditioned, we need better insight into its structure.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \dots & \dots & \dots & \dots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_M \end{pmatrix}$$

Consider \mathbf{b} and column vectors of A as vectors in \mathbb{R}^M , then the task is:

Write \mathbf{b} as a linear combination of columns in \mathbf{A} .

Ill-conditioned matrix means columns of A are not fully independent.

Before, we've mostly been doing row operations.

Now, we will look closely into the column space of A .

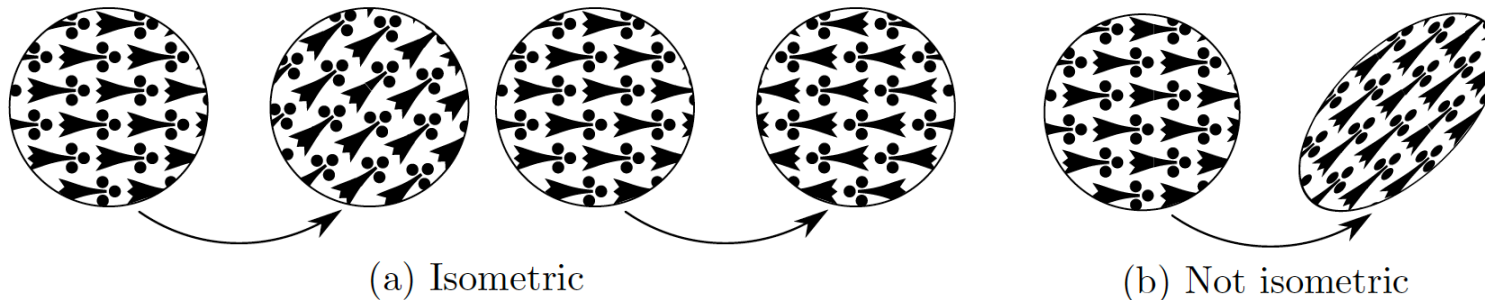
Orthogonality

Let $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n \in \mathbb{R}^n$ satisfy $\mathbf{q}_i \cdot \mathbf{q}_j \equiv \mathbf{q}_i^T \mathbf{q}_j = \delta_{ij}$, they form an **orthogonal basis** for \mathbb{R}^n .

A matrix **Q** is **orthogonal** if its column vectors form an orthogonal basis.

$$Q^T Q = I \Rightarrow Q^{-1} = Q^T.$$

For any two vectors \mathbf{x}, \mathbf{y} , we have $(Q\mathbf{x}) \cdot (Q\mathbf{y}) = \mathbf{x}^T Q^T Q \mathbf{y} = \mathbf{x}^T I \mathbf{y} = \mathbf{x} \cdot \mathbf{y}$ (**isometric**)



Geometrically, an orthogonal matrix corresponds to **rotation** and/or **reflection**, but no stretch or shear.

QR factorization

For any $A \in \mathbb{R}^{m \times n}$ ($m \geq n$), it can be shown that A can be decomposed as

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$$

Q: $m \times m$ orthogonal matrix
R: $n \times n$ upper triangular matrix with non-negative diagonal elements.

When $m=n$ and if A is non-singular, this factorization is unique.

With QR factorization, solving $Ax=b$ becomes: $QRx = b \Rightarrow Rx = Q^T b$

This turns out to be more stable/accurate in general (at more computational cost).

Basis of the *QR algorithm* to solve for eigenvalue problem (which we do not cover).

Least mean square problem (again!)

Let $A \in \mathbb{R}^{m \times n}$ ($m \geq n$) with $A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$. We can write $Q = \begin{bmatrix} Q_1 & \cancel{Q_2} \\ n & m-n \end{bmatrix}$

We are trying to minimize: $\|A\mathbf{x} - \mathbf{b}\|_2^2 = \|Q^T A\mathbf{x} - Q^T \mathbf{b}\|_2^2$ (where $\mathbf{b} \in \mathbb{R}^m$)

We can separate $Q^T \mathbf{b} = \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} \mathbf{b} \equiv \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix}$ $\begin{matrix} n \\ m-n \end{matrix}$

Now it becomes obvious that $\|A\mathbf{x} - \mathbf{b}\|_2^2 = \|R\mathbf{x} - \mathbf{c}_1\|_2^2 + \|\mathbf{c}_2\|_2^2$

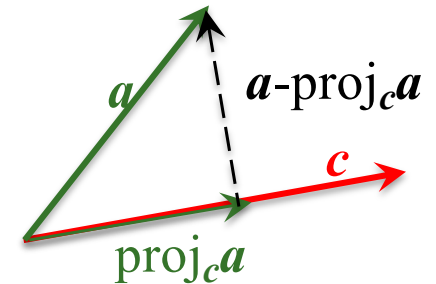
So, the least mean square problem is reduced to simply solving: $R\mathbf{x} = \mathbf{c}_1$.

This is much more effective for ill-conditioned input matrix A.

Projection operator

Consider two vectors \mathbf{a} and \mathbf{c} , the **projection** of \mathbf{a} on \mathbf{c} is given by

$$\text{proj}_{\mathbf{c}} \mathbf{a} = \frac{\mathbf{a} \cdot \mathbf{c}}{\|\mathbf{c}\|_2^2} \mathbf{c} \quad (\text{simplified if } \|\mathbf{c}\|_2=1)$$



The remaining component, $\mathbf{a} - \text{proj}_{\mathbf{c}} \mathbf{a}$, becomes orthogonal to \mathbf{c} .

Let $\{\mathbf{q}_1, \dots, \mathbf{q}_k\}$ ($k < n$) be a subset of an orthogonal basis in \mathbb{R}^n , then for any vector \mathbf{a} , it can be projected into the subspace spanned from this bases as

$$\text{proj}_{\text{span}\{\mathbf{q}_1, \dots, \mathbf{q}_k\}} \mathbf{a} = (\mathbf{a} \cdot \mathbf{q}_1) \mathbf{q}_1 + \dots + (\mathbf{a} \cdot \mathbf{q}_k) \mathbf{q}_k$$

The remaining component must be orthogonal to the subspace spanned from this basis.

QR by Gram-Schmidt orthogonalization

For $A = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_n \end{bmatrix}$, the procedure of [Gram-Schmidt orthogonalization](#) is as follows

$$\mathbf{u}_1 = \mathbf{a}_1, \quad \mathbf{q}_1 = \mathbf{u}_1 / \|\mathbf{u}_1\|_2$$

$$\mathbf{u}_2 = \mathbf{a}_2 - (\mathbf{a}_2 \cdot \mathbf{q}_1) \mathbf{q}_1, \quad \mathbf{q}_2 = \mathbf{u}_2 / \|\mathbf{u}_2\|_2$$

$$\mathbf{u}_k = \mathbf{a}_k - (\mathbf{a}_k \cdot \mathbf{q}_1) \mathbf{q}_1 - \dots - (\mathbf{a}_k \cdot \mathbf{q}_{k-1}) \mathbf{q}_{k-1}, \quad \mathbf{q}_k = \mathbf{u}_k / \|\mathbf{u}_k\|_2$$

The resulting QR factorization is

$$A = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_n \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \dots & \mathbf{q}_n \end{bmatrix} \begin{bmatrix} \mathbf{a}_1 \cdot \mathbf{q}_1 & \mathbf{a}_2 \cdot \mathbf{q}_1 & \dots & \mathbf{a}_n \cdot \mathbf{q}_1 \\ 0 & \mathbf{a}_2 \cdot \mathbf{q}_2 & \dots & \mathbf{a}_n \cdot \mathbf{q}_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \mathbf{a}_n \cdot \mathbf{q}_n \end{bmatrix}$$

Total cost: $\sim N^3$ ($\sim 3 \times \text{LU}$)

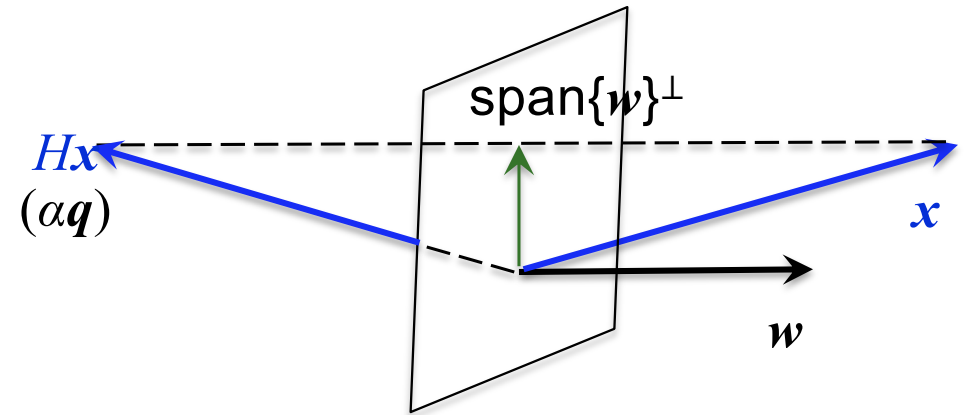
While straightforward, Gram-Schmidt can be numerically unstable to round-off errors, particularly for ill-conditioned matrices.

Householder transformation

Let $\mathbf{w} \in \mathbb{R}^n$, $\|\mathbf{w}\|_2 = 1$, we define

$$H \equiv I - 2\mathbf{w}\mathbf{w}^T$$

as **Householder transformation**.



Note that H is an orthogonal matrix: $H^T = H$, $H^T H = H^2 = I$.

For any given vector $\mathbf{x} \in \mathbb{R}^n$, $H\mathbf{x}$ gives the **mirror reflection** of \mathbf{x} with respect to the subspace orthogonal to \mathbf{w} .

By properly choosing \mathbf{w} , we can transform a vector \mathbf{x} to any direction \mathbf{q} , as follows

$$\mathbf{w} = \frac{\mathbf{x} - \alpha\mathbf{q}}{\|\mathbf{x} - \alpha\mathbf{q}\|_2} \quad \text{where} \quad \alpha \equiv \|\mathbf{x}\|_2$$

QR by Householder orthogonalization

For $A = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_n \end{bmatrix}$, the procedure of Householder orthogonalization is:

$$H_1 A = \begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ 0 & \times & \dots & \times \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \times & \dots & \times \end{pmatrix} \quad \text{A}_2$$

First, let $r_{11} = \|\mathbf{a}_1\|_2$, one can find a Householder transform H_1 so that: $H_1 \mathbf{a}_1 = (r_{11}, \mathbf{0})$

In practice, it is handy to write Householder transform as

$$H = I - \beta \mathbf{v} \mathbf{v}^T, \quad \beta = \frac{2}{\mathbf{v}^T \mathbf{v}}. \quad \mathbf{v} \text{ is called Householder vector.}$$

We can normalize \mathbf{v} such that $v(1)=1$.

This way, $v_1(2:n)$ can be stored to the original matrix as:

$$\begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ v_1(2) & \times & \dots & \times \\ \vdots & \vdots & \ddots & \vdots \\ v_1(n) & \times & \dots & \times \end{pmatrix} \quad 44$$

QR by Householder orthogonalization

Next, consider the lower right block A_2 .

We can find another Householder transform \tilde{H}_2 , which reduces the first column of A_2 into $(r_{22}, \mathbf{0})^T$:

$$H_2 = \text{diag}(I_1, \tilde{H}_2), \text{ and } H_2 H_1 A = \begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ 0 & r_{22} & \dots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \times \end{pmatrix}$$

Similarly, the resulting Householder vector \tilde{v}_2 can be stored in the original matrix below r_{22} .

One further saves the beta-values as a vector $\beta = (\beta_1, \beta_2, \dots, \beta_n)^T$

This algorithm has much better stability and accuracy.

Total cost: $\sim(2/3)N^3$ (workload will further double to recover Q)

Optional: Givens transformation

A **Givens rotation** is represented in matrix form

$$G(i, k, \theta) = \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix} \begin{matrix} i \\ \\ k \\ \\ \end{matrix}$$

For any vector \mathbf{x} , applying Givens rotation yields

$$\mathbf{y} = G(i, k, \theta)\mathbf{x}$$

$$y_j = \begin{cases} cx_i - sx_k, & j = i \\ sx_i + cx_k, & j = k \\ x_j, & j \neq i, k \end{cases}$$

where $c = \cos \theta$, $s = \sin \theta$

One can choose $c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}}$, $s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}}$ so that y_k vanishes.

Optional: QR by Givens transformation

Applying Givens rotation to a matrix A only affects its i th and k th rows.

General idea: zero out one element at a time by applying Givens rotation, and gradually zero out all elements in the lower triangle.

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(1,2)} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix}$$

For a **dense matrix**, **cost** of Givens QR method $\sim (4/3)N^3$.

Advantageous when there are relatively few non-zero elements in A (e.g., tridiagonal matrix), and it is more easily parallelizable.

How to compute matrix products

It is common task to compute matrix product:

$$C = C + AB \quad \text{where} \quad C \in \mathbb{R}^{m \times n}, A \in \mathbb{R}^{m \times r}, B \in \mathbb{R}^{r \times n}$$

Naive approach:

```
for i=1:m
    for j=1:n
        for k=1:r
            C[i][j] += A[i][k]*B[k][j];
        end
    end
end
```

This will be slow when A, B are large matrices due to limited cache size.

How to compute matrix products

It is common task to compute matrix product:

$$C = C + AB \quad \text{where} \quad C \in \mathbb{R}^{m \times n}, A \in \mathbb{R}^{m \times r}, B \in \mathbb{R}^{r \times n}$$

To improve cache performance, divide them into blocks so that each block can fit (L1 cache):

$$C = \begin{bmatrix} C_{11} & \cdots & C_{1r} \\ \vdots & \ddots & \vdots \\ C_{q1} & \cdots & C_{qr} \end{bmatrix} \quad A = \begin{bmatrix} A_{11} & \cdots & A_{1p} \\ \vdots & \ddots & \vdots \\ A_{q1} & \cdots & A_{qp} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & \cdots & B_{1r} \\ \vdots & \ddots & \vdots \\ B_{p1} & \cdots & B_{pr} \end{bmatrix}$$

Compute matrix product by summation over results from individual blocks:

$$C_{ij} = C_{ij} + \sum_{k=1}^p A_{ik} B_{kj}$$

Typical block size of 32x32 should be reasonable.

Summary

- Gaussian elimination with pivoting
- LU decomposition with pivoting
- Cholesky/LDL decomposition for symmetric matrices

- Vector and matrix norms
- Condition numbers

- Orthogonal matrix
- QR factorization by Gram-Schmidt orthogonalization
- QR factorization by Householder/Givens transformation

Reading material:

Singular value decomposition

Singular values and vectors

Definition: for a matrix $A \in \mathbb{R}^{m \times n}$, its **singular values** are defined as the **square root of all positive eigenvalues of the matrix $A^T A$** (and AA^T).

The matrices $A^T A$ and AA^T are **real, symmetric, and positive semi-definite**.

We now show **they share the same eigenvalues**.

Let $\lambda_i > 0$ be an eigenvalue of $A^T A$, and define $\sigma_i \equiv \sqrt{\lambda_i}$. Let $\mathbf{v}_i \in \mathbb{R}^n$ be the corresponding eigenvector, and set it to be a unit vector, so that:

$$A^T A \mathbf{v}_i = \lambda_i \mathbf{v}_i = \sigma_i^2 \mathbf{v}_i$$

$$\text{and} \quad \|A \mathbf{v}_i\|_2^2 = \mathbf{v}_i^T A^T A \mathbf{v}_i = \lambda_i \|\mathbf{v}_i\|_2^2 = \sigma_i^2$$

(we will see that this gives a geometric interpretation of the singular values)

Singular values and vectors

We further define $\mathbf{u}_i \equiv \sigma_i^{-1} A \mathbf{v}_i \in \mathbb{R}^m$, which is also a unit vector.

Now we have: $(AA^T)\mathbf{u}_i = \sigma_i^{-1}(AA^T)(A\mathbf{v}_i) = \sigma_i A\mathbf{v}_i = \lambda_i \mathbf{u}_i$

Clearly, \mathbf{u}_i is an eigenvector of AA^T with **eigenvalue** λ_i .

It is also straightforward to see the following relation on singular values:

$$\begin{array}{ccc} A\mathbf{v}_i = \sigma_i \mathbf{u}_i & , & A^T \mathbf{u}_i = \sigma_i \mathbf{v}_i \\ \uparrow & & \uparrow \\ \text{right singular vectors} & & \text{left singular vectors} \end{array}$$

Also, we can see that \mathbf{u}_i is within column space of A , and \mathbf{v}_i is within row space of A .

Singular value decomposition (SVD)

Let there be k positive eigenvalues λ_i ($i=1 \dots k$). We can use their columns to make two matrices:

$$\bar{U} = (\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k) \in \mathbb{R}^{m \times k} \quad (\text{column space})$$

$$\bar{V} = (\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k) \in \mathbb{R}^{n \times k} \quad (\text{row space})$$

Taking \mathbf{e}_i to be the i th unit vector in \mathbb{R}^k , we can examine the following:

$$\bar{U}^T A \bar{V} \mathbf{e}_i = \bar{U}^T A \mathbf{v}_i = \sigma_i \bar{U}^T \mathbf{u}_i = \sigma_i \mathbf{e}_i$$

This means that $\bar{U}^T A \bar{V} = \bar{\Sigma} \equiv \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_k)$

Rearranging this expression yields:

$$A = \bar{U} \bar{\Sigma} \bar{V}^T = \sum_{i=1}^k \sigma_i \mathbf{u}_i \mathbf{v}_i^T \quad (\text{usually take } \sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_k > 0)$$

This is known as (*compact form of*) singular value decomposition.

Singular value decomposition (SVD)

There are other conventions of writing SVD, e.g., the full version is obtained by:

We can fill additional columns in \bar{U} and \bar{V} to form orthogonal square matrices U and V , and add more zeros in $\bar{\Sigma}$ to form Σ . Then, A can be decomposed into

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

m×n m×m m×n n×n

This decomposition is non-unique, but is in many cases useful (U, V are orthogonal matrices).

Full version:

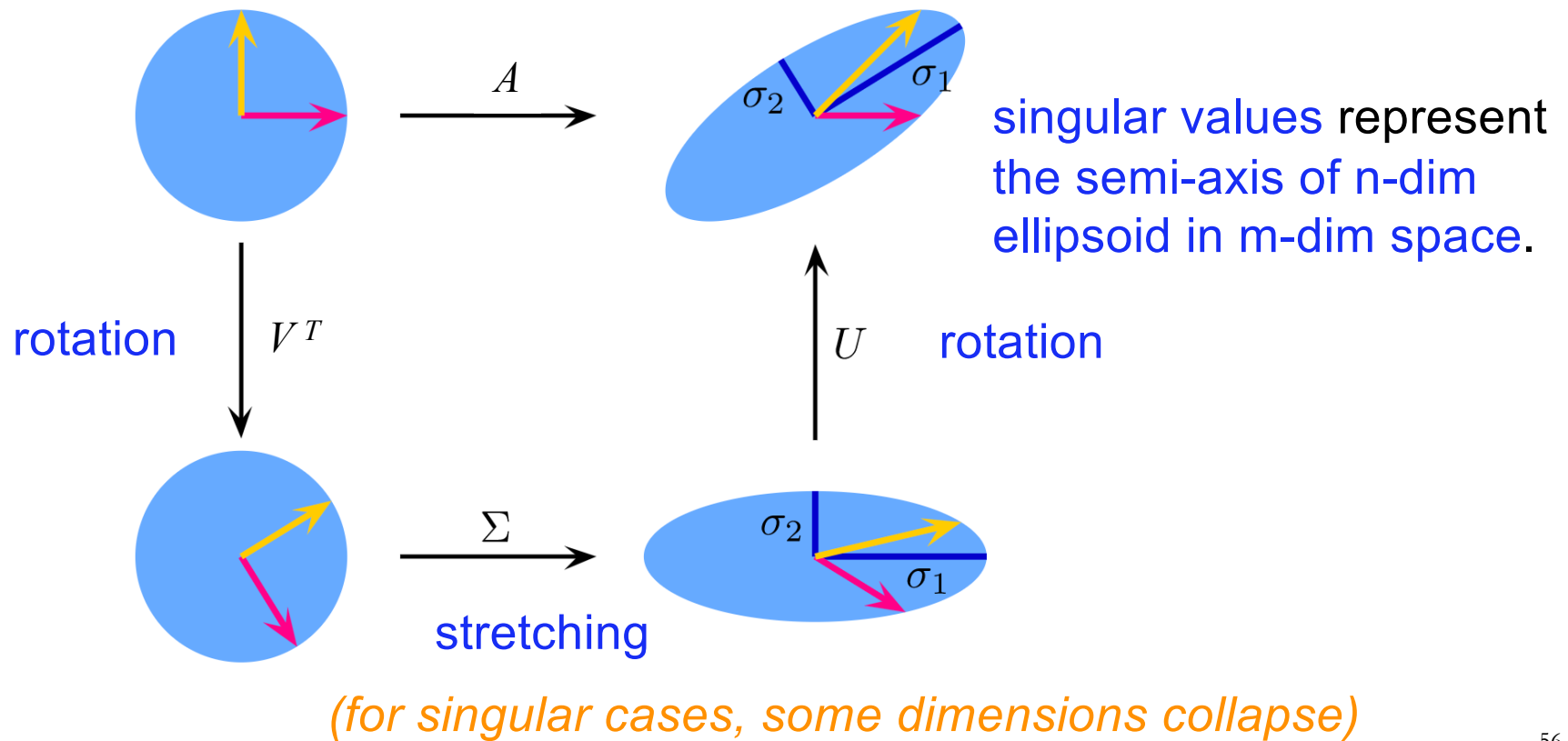
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} -0.1525 & -0.8226 & -0.3945 & -0.3800 \\ -0.3499 & -0.4214 & 0.2428 & 0.8007 \\ -0.5474 & -0.0201 & 0.6979 & -0.4614 \\ -0.7448 & 0.3812 & -0.5462 & 0.0407 \end{bmatrix} \begin{bmatrix} 14.27 & 0 \\ 0 & 0.6268 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} -0.6414 & 0.7672 \\ -0.7672 & -0.6414 \end{bmatrix}$$

Compact/thin form:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} -0.1525 & -0.8226 \\ -0.3499 & -0.4214 \\ -0.5474 & -0.0201 \\ -0.7448 & 0.3812 \end{bmatrix} \begin{bmatrix} 14.27 & 0 \\ 0 & 0.6268 \end{bmatrix} \begin{bmatrix} -0.6414 & 0.7672 \\ -0.7672 & -0.6414 \end{bmatrix}$$

SVD: geometric interpretation

Consider A as a map from \mathbb{R}^n to \mathbb{R}^m .



SVD: how close is A to being singular?

Consider a square, invertible matrix A with $A = U\Sigma V^T$

Clearly, its inverse is $A^{-1} = V\Sigma^{-1}U^T$

Now take the 2-norm: $\|A\|_2 \equiv \sqrt{\lambda_{\max}(A^T A)} = \sigma_1$

$$\|A^{-1}\|_2 = \sigma_N^{-1}$$

Therefore, its 2-condition number is $\text{cond}_2(A) = \frac{\sigma_1}{\sigma_N}$

With SVD, we know how close a matrix is close to singular.

Another important result is: $\sum_{i=1}^k \sigma_i^2 = \sum_{i,j} a_{ij}^2 = \|A\|_F^2$

Solving linear systems

When solving $Ax = b$, there are two pathological situations:

1. **Overdetermined**: b not in the range of Ax , e.g., when $m > n$.

The least mean square problem: we solve $A^T Ax = A^T b$ to minimize $\|Ax - b\|_2$.

2. **Underdetermined**, e.g., when $m < n$.

This can be handled by solving $A^T Ax = A^T b$ plus a regularization condition.

Let this condition be to minimize $\|x\|_2$.

$$A^T Ax = A^T b \Rightarrow \Sigma^T \Sigma y = \Sigma d \quad \text{where} \quad y \equiv V^T x, \quad d \equiv U^T b$$

The regularization condition is equivalent to discarding the zeros in the diagonal elements in Σ .

Unified solutions with pseudo-inverse

This allows us to define the pseudo-inverse of matrix $A \in \mathbb{R}^{m \times n}$:

Let $A = U\Sigma V^T$, with non-zero singular values $\sigma_1, \sigma_2, \dots, \sigma_k$ in the diagonal of Σ .

Define Σ^+ as $\Sigma_{ij}^+ = \sigma_i^{-1}$ ($i = j \leq k$) but 0 otherwise.

Then the pseudo-inverse of A is $A^+ \equiv V\Sigma^+U^T$

With this definition, the solution to $Ax = b$ is simply given by $x = A^+b$

In fact, this applies to all cases: complete, underdetermined, and overdetermined.

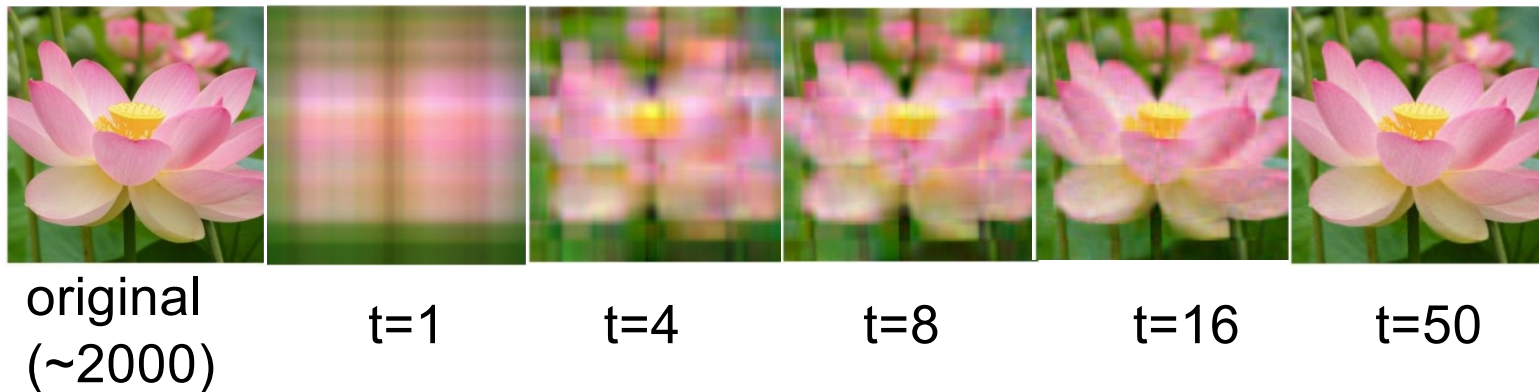
In practice, SVD is much more computationally expensive than, e.g., LU decomposition, so this is useful mainly in pathological cases.

Approximate A by truncation

Given the SVD of A: $A = \bar{U}\bar{\Sigma}\bar{V}^T = \sum_{i=1}^k \sigma_i \mathbf{u}_i \mathbf{v}_i^T$

When most singular values are small, then A will be well approximated by only a few terms in the above.

This is useful for accelerating matrix operations, and wide applications in signal and image processing.



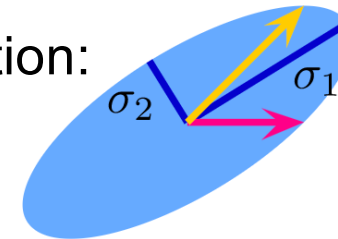
Another important application is Principle Component Analysis, to be covered later in the course.

Summary: SVD

Definition and derivation:

$$A = U\Sigma V^T = \bar{U}\bar{\Sigma}\bar{V}^T = \sum_{i=1}^k \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

Interpretation:



Semi-axis in
hyper-ellipsoids

Applications:

- Assess how close a matrix is to being singular.
- Obtain null space and range
- Pseudo-inverse for solving general linear systems
- Obtaining reliable approximations to original matrix
- PCA (to be covered later)
- Many others in data science.

Computing SVD: similar to eigenvalue problem for $A^T A$, cost is \sim several N^3 .