Unit 2: Numerical Linear Algebra

Motivation

Almost everything in Scientific Computing relies on Numerical Linear Algebra!

We often reformulate problems as $Ax = b$, *e.g.* from Unit 1:

- Interpolation (Vandermonde matrix) and linear least-squares (normal equations) are naturally expressed as linear systems
- Gauss–Newton/Levenberg–Marquardt involve approximating nonlinear problem by a sequence of linear systems

Similar themes will arise in remaining units (Numerical Calculus, Optimization, Eigenvalue problems)

Motivation

The goal of this unit is to cover:

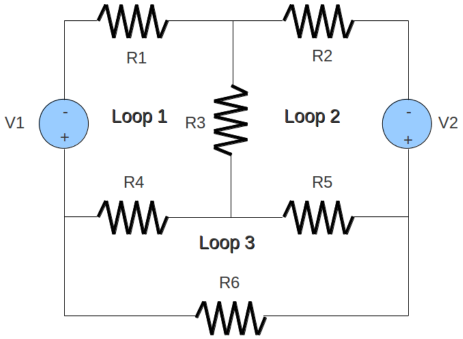
- key linear algebra concepts that underpin Scientific Computing
- algorithms for solving $Ax = b$ in a stable and efficient manner
- algorithms for computing factorizations of A that are useful in many practical contexts (LU, QR)

First, we discuss some practical cases where $Ax = b$ arises directly in mathematical modeling of physical systems

Example: Electric Circuits

Ohm's Law: Voltage drop due to a current i through a resistor R is $V = iR$

Kirchoff's Law: The net voltage drop in a closed loop is zero



Example: Electric Circuits

Let i_j denote the current in "loop j"

Then, we obtain the linear system:

$$\begin{bmatrix} (R_1 + R_3 + R_4) & R_3 & R_4 \\ R_3 & (R_2 + R_3 + R_5) & -R_5 \\ R_4 & -R_5 & (R_4 + R_5 + R_6) \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} V_1 \\ V_2 \\ 0 \end{bmatrix}$$

Circuit simulators solve large linear systems of this type

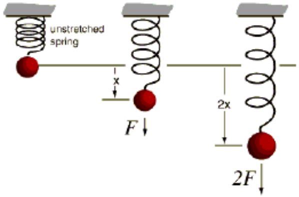
Example: Structural Analysis

Common in structural analysis to use a linear relationship between force and displacement, Hooke's Law

Simplest case is the Hookean spring law

$$F = kx,$$

- k: spring constant (stiffness)
- F: applied load
- x: spring extension



Example: Structural Analysis

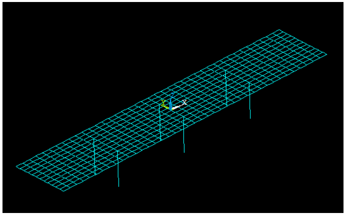
This relationship can be generalized to structural systems in 2D and 3D, which yields a linear system of the form

$$Kx = F$$

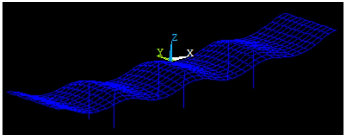
- $K \in \mathbb{R}^{n \times n}$: "stiffness matrix"
- $F \in \mathbb{R}^{n}$: "load vector"
- $x \in \mathbb{R}^{n}$: "displacement vector"

Example: Structural Analysis

Solving the linear system yields the displacement (x), hence we can simulate structural deflection under applied loads (F)



$\xrightarrow{Kx=F}$

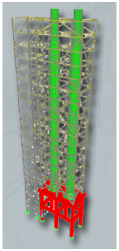
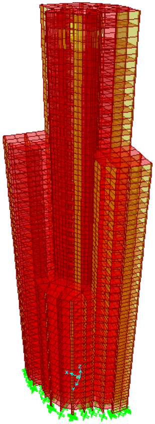
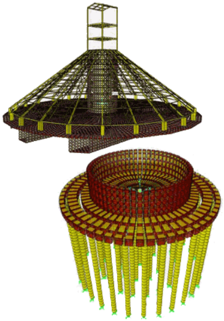


Unloaded structure

Loaded structure

Example: Structural Analysis

It is common engineering practice to use Hooke's Law to simulate complex structures, which leads to large linear systems

(From SAP2000, structural analysis software)

Example: Economics

Leontief awarded Nobel Prize in Economics in 1973 for developing linear input/output model for production/consumption of goods

Consider an economy in which n goods are produced and consumed

- $A \in \mathbb{R}^{n \times n}$: a_{ij} represents amount of good j required to produce 1 unit of good i
- $x \in \mathbb{R}^n$: x_i is number of units of good i produced
- $d \in \mathbb{R}^n$: d_i is consumer demand for good i

In general $a_{ii} = 0$, and A may or may not be sparse

Example: Economics

The total amount of x_i produced is given by the sum of consumer demand (d_i) and the amount of x_i required to produce each x_j

$$x_i = \underbrace{a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n}_{\text{production of other goods}} + d_i$$

Hence $x = Ax + d$ or,

$$(\mathrm{I} - A)x = d$$

Solve for x to determine the required amount of production of each good

If we consider many goods (*e.g.* an entire economy), then we get a large linear system

Summary

Matrix computations arise all over the place!

Numerical Linear Algebra algorithms provide us with a toolbox for performing these computations in an efficient and stable manner

In most cases, can use these tools as black boxes, but it's important to understand what the linear algebra black boxes do:

- Pick the right algorithm for a given situation (*e.g.* exploit structure in a problem: symmetry, bandedness, *etc.*)
- Understand how and when the black box can fail

Preliminaries

In this chapter we will focus on linear systems $Ax = b$ for $A \in \mathbb{R}^{n \times n}$ and $b, x \in \mathbb{R}^n$

Recall that it is often helpful to think of matrix multiplication as a linear combination of the columns of A, where x_j are the weights

That is, we have $b = Ax = \sum_{j=1}^{n} x_j a_{(:,j)}$ where $a_{(:,j)} \in \mathbb{R}^n$ is the j^{th} column of A and x_j are scalars

Preliminaries

This can be displayed schematically as

$$
\begin{aligned}
\left[\begin{array}{c} \\ b \\ \\ \end{array}\right] &= \left[\begin{array}{c|c|c|c} & & & \\ a_{(:,1)} & a_{(:,2)} & \cdots & a_{(:,n)} \\ & & & \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array}\right] \\
&= x_1 \left[\begin{array}{c} \\ a_{(:,1)} \\ \\ \end{array}\right] + \cdots + x_n \left[\begin{array}{c} \\ a_{(:,n)} \\ \\ \end{array}\right]
\end{aligned}
$$

Preliminaries

We therefore interpret $Ax = b$ as: "x is the vector of coefficients of the linear expansion of b in the basis of columns of A"

Often this is a more helpful point of view than conventional interpretation of "dot-product of matrix row with vector"

e.g. from "linear combination of columns" view we immediately see that $Ax = b$ has a solution if

$$b \in \text{span}\{a_{(:,1)}, a_{(:,2)}, \cdots, a_{(:,n)}\}$$

(this holds even if A isn't square)

Let us write $\text{image}(A) \equiv \text{span}\{a_{(:,1)}, a_{(:,2)}, \cdots, a_{(:,n)}\}$

Preliminaries

Existence and Uniqueness:

Solution $x \in \mathbb{R}^n$ exists if $b \in \text{image}(A)$

If solution x exists and the set $\{a_{(:,1)}, a_{(:,2)}, \cdots, a_{(:,n)}\}$ is linearly independent, then x is unique[1]

If solution x exists and $\exists z \neq 0$ such that $Az = 0$, then also $A(x + \gamma z) = b$ for any $\gamma \in \mathbb{R}$, hence infinitely many solutions

If $b \notin \text{image}(A)$ then $Ax = b$ has no solution

[1] Linear independence of columns of A is equivalent to $Az = 0 \implies z = 0$

Preliminaries

The inverse map $A^{-1}\colon \mathbb{R}^n \to \mathbb{R}^n$ is well-defined if and only if $Ax = b$ has unique solution for all $b \in \mathbb{R}^n$

Unique matrix $A^{-1} \in \mathbb{R}^{n\times n}$ such that $AA^{-1} = A^{-1}A = I$ exists if any of the following equivalent conditions are satisfied:

- $\det(A) \neq 0$
- $\operatorname{rank}(A) = n$
- For any $z \neq 0$, $Az \neq 0$ (null space of A is $\{0\}$)

A is non-singular if A^{-1} exists, and then $x = A^{-1}b \in \mathbb{R}^n$

A is singular if A^{-1} does not exist

Norms

A norm $\|\cdot\|: V \to \mathbb{R}$ is a function on a vector space V that satisfies

- $\|x\| \geq 0$ and $\|x\| = 0 \implies x = 0$
- $\|\gamma x\| = |\gamma|\|x\|$, for $\gamma \in \mathbb{R}$
- $\|x + y\| \leq \|x\| + \|y\|$

Norms

Also, the triangle inequality implies another helpful inequality: the “reverse triangle inequality”, $|\|x\| - \|y\|| \leq \|x - y\|$

Proof: Let $a = y$, $b = x - y$, then

$$\|x\| = \|a+b\| \leq \|a\|+\|b\| = \|y\|+\|x-y\| \implies \|x\|-\|y\| \leq \|x-y\|$$

Repeat with $a = x$, $b = y - x$ to show $|\|x\| - \|y\|| \leq \|x - y\|$

Vector Norms

Let's now introduce some common norms on $\mathbb{R}^n$

Most common norm is the Euclidean norm (or 2-norm):

$$\|x\|_2 \equiv \sqrt{\sum_{j=1}^{n} x_j^2}$$

2-norm is special case of the p-norm for any $p \geq 1$:

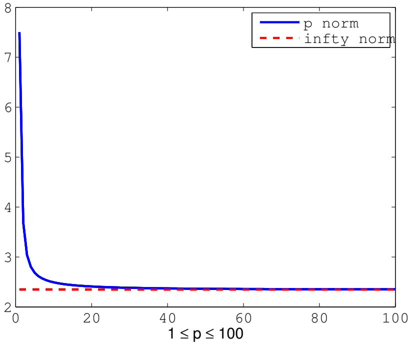
$$\|x\|_p \equiv \left(\sum_{j=1}^{n} |x_j|^p\right)^{1/p}$$

Also, limiting case as $p \to \infty$ is the ∞-norm:

$$\|x\|_\infty \equiv \max_{1 \leq i \leq n} |x_i|$$

Vector Norms

[*norm.py*] $\|x\|_\infty = 2.35$, we see that p-norm approaches ∞-norm: picks out the largest entry in x



Vector Norms

We generally use whichever norm is most convenient/appropriate for a given problem, *e.g.* 2-norm for least-squares analysis

Different norms give different (but related) measures of size

In particular, an important mathematical fact is:

All norms on a finite dimensional space (such as $\mathbb{R}^n$) are equivalent

Vector Norms

That is, let $\|\cdot\|_a$ and $\|\cdot\|_b$ be two norms on a finite dimensional space V, then $\exists c_1, c_2 \in \mathbb{R}_{>0}$ such that for any $x \in V$

$$c_1\|x\|_a \leq \|x\|_b \leq c_2\|x\|_a$$

(Also, from above we have $\frac{1}{c_2}\|x\|_b \leq \|x\|_a \leq \frac{1}{c_1}\|x\|_b$)

Hence if we can derive an inequality in an arbitrary norm on V, it applies (after appropriate scaling) in any other norm too

Vector Norms

In some cases we can explicitly calculate values for c_1, c_2:

e.g. $\|x\|_2 \leq \|x\|_1 \leq \sqrt{n}\|x\|_2$, since

$$\|x\|_2^2 = \left(\sum_{j=1}^{n} |x_j|^2\right) \leq \left(\sum_{j=1}^{n} |x_j|\right)^2 = \|x\|_1^2 \implies \|x\|_2 \leq \|x\|_1$$

[*e.g.* consider $|a|^2 + |b|^2 \leq |a|^2 + |b|^2 + 2|a||b| = (|a| + |b|)^2$]

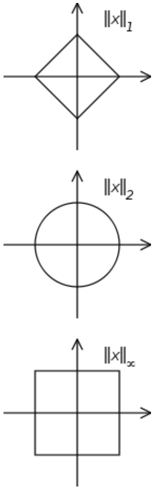
$$\|x\|_1 = \sum_{j=1}^{n} 1 \times |x_j| \leq \left(\sum_{j=1}^{n} 1^2\right)^{1/2} \left(\sum_{j=1} |x_j|^2\right)^{1/2} = \sqrt{n}\,\|x\|_2$$

[We used Cauchy-Schwarz inequality in $\mathbb{R}^n$:
$\sum_{j=1}^{n} a_j b_j \leq (\sum_{j=1}^{n} a_j^2)^{1/2}(\sum_{j=1}^{n} b_j^2)^{1/2}$]

Vector Norms

Different norms give different measurements of size

The "unit circle" in three different norms: $\{x \in \mathbb{R}^2 : \|x\|_p = 1\}$ for $p = 1, 2, \infty$



Matrix Norms

There are many ways to define norms on matrices

For example, the Frobenius norm is defined as:

$$\|A\|_F \equiv \left(\sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}$$

(If we think of A as a vector in $\mathbb{R}^{n^2}$, then Frobenius is equivalent to the vector 2-norm of A)

Matrix Norms

Usually the matrix norms induced by vector norms are most useful, *e.g.*:

$$\|A\|_p \equiv \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} = \max_{\|x\|_p = 1} \|Ax\|_p$$

This definition implies the useful property $\|Ax\|_p \leq \|A\|_p \|x\|_p$, since

$$\|Ax\|_p = \frac{\|Ax\|_p}{\|x\|_p} \|x\|_p \leq \left(\max_{v \neq 0} \frac{\|Av\|_p}{\|v\|_p} \right) \|x\|_p = \|A\|_p \|x\|_p$$

Matrix Norms

The 1-norm and ∞-norm can be calculated straightforwardly:

$$\begin{aligned} \|A\|_1 &= \max_{1\le j\le n} \|a_{(:,j)}\|_1 && \text{(max column sum)} \\ \|A\|_\infty &= \max_{1\le i\le n} \|a_{(i,:)}\|_1 && \text{(max row sum)} \end{aligned}$$

We will see how to compute the matrix 2-norm next chapter

Condition Number

Recall from Unit 0 that the condition number of $A \in \mathbb{R}^{n \times n}$ is defined as

$$\kappa(A) \equiv \|A\| \|A^{-1}\|$$

The value of $\kappa(A)$ depends on which norm we use

Both Python and Matlab can calculate the condition number for different norms

If A is square then by convention A singular $\implies \kappa(A) = \infty$

The Residual

Recall that the residual $r(x) = b - Ax$ was crucial in least-squares problems

It is also crucial in assessing the accuracy of a proposed solution $(\hat{x})$ to a square linear system $Ax = b$

Key point: The residual $r(\hat{x})$ is always computable, whereas in general the error $\Delta x \equiv x - \hat{x}$ isn't

The Residual

We have that $\|\Delta x\| = \|x - \hat{x}\| = 0$ if and only if $\|r(\hat{x})\| = 0$

However, small residual doesn't necessarily imply small $\|\Delta x\|$

Observe that

$$\|\Delta x\| = \|x - \hat{x}\| = \|A^{-1}(b - A\hat{x})\| = \|A^{-1}r(\hat{x})\| \leq \|A^{-1}\|\|r(\hat{x})\|$$

Hence

$$\frac{\|\Delta x\|}{\|\hat{x}\|} \leq \frac{\|A^{-1}\|\|r(\hat{x})\|}{\|\hat{x}\|} = \frac{\|A\|\|A^{-1}\|\|r(\hat{x})\|}{\|A\|\|\hat{x}\|} = \kappa(A)\frac{\|r(\hat{x})\|}{\|A\|\|\hat{x}\|} \quad (*)$$

The Residual

Define the relative residual as $\|r(\hat{x})\|/(\|A\|\|\hat{x}\|)$

Then our inequality states that "relative error is bounded by condition number times relative residual"

This is just like our condition number relationship from Unit 0:

$$\kappa(A) \geq \frac{\|\Delta x\|/\|x\|}{\|\Delta b\|/\|b\|}, \qquad \textit{i.e.} \qquad \frac{\|\Delta x\|}{\|x\|} \leq \kappa(A)\frac{\|\Delta b\|}{\|b\|} \quad (**)$$

The reason $(*)$ and $(**)$ are related is that the residual measures the "input pertubation" in $Ax = b$

To see this, let's consider $Ax = b$ to be a map $b \in \mathbb{R}^n \to x \in \mathbb{R}^n$

The Residual

Then we can consider $\hat{x}$ to be the exact solution for some perturbed input $\hat{b} = b + \Delta b$, *i.e.* $A\hat{x} = \hat{b}$

The residual associated with $\hat{x}$ is $r(\hat{x}) = b - A\hat{x} = b - \hat{b} = -\Delta b$, *i.e.* $\|r(\hat{x})\| = \|\Delta b\|$

In general, a numerically stable algorithm gives us the exact solution to a slightly perturbed problem, *i.e.* a small residual[2]

This is a reasonable expectation for a stable algorithm: rounding error doesn't accumulate, so effective input perturbation is small

[2]More precisely, this is called a "backward stable algorithm"

The Residual (Heath, Example 2.8)

Consider a 2×2 example to clearly demonstrate the difference between residual and error

$$Ax = \begin{bmatrix} 0.913 & 0.659 \\ 0.457 & 0.330 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.254 \\ 0.127 \end{bmatrix} = b$$

The exact solution is given by $x = [1, -1]^T$

Suppose we compute two different approximate solutions (*e.g.* using two different algorithms)

$$\hat{x}_{(i)} = \begin{bmatrix} -0.0827 \\ 0.5 \end{bmatrix}, \qquad \hat{x}_{(ii)} = \begin{bmatrix} 0.999 \\ -1.001 \end{bmatrix}$$

The Residual (Heath, Example 2.8)

Then,

$$\|r(\hat{x}_{(i)})\|_1 = 2.1 \times 10^{-4}, \qquad \|r(\hat{x}_{(ii)})\|_1 = 2.4 \times 10^{-2}$$

but

$$\|x - \hat{x}_{(i)}\|_1 = 2.58, \qquad \|x - \hat{x}_{(ii)}\|_1 = 0.002$$

In this case, $\hat{x}_{(ii)}$ is better solution, but has larger residual!

This is possible here because $\kappa(A) = 1.25 \times 10^4$ is quite large (*i.e.* rel. error $\leq 1.25 \times 10^4 \times$ rel. residual)

Solving $Ax = b$

Familiar idea for solving $Ax = b$ is to use Gaussian elimination to transform $Ax = b$ to a triangular system

What is a triangular system?

- Upper triangular matrix $U \in \mathbb{R}^{n \times n}$: if $i > j$ then $u_{ij} = 0$
- Lower triangular matrix $L \in \mathbb{R}^{n \times n}$: if $i < j$ then $\ell_{ij} = 0$

Question: Why is triangular good?

Answer: Because triangular systems are easy to solve!

Solving $Ax = b$

Suppose we have $Ux = b$, then we can use "back-substitution"

$$\begin{aligned}
x_n &= b_n / u_{nn} \\
x_{n-1} &= (b_{n-1} - u_{n-1,n} x_n) / u_{n-1,n-1} \\
&\vdots \\
x_j &= \left(b_j - \sum_{k=j+1}^{n} u_{jk} x_k \right) / u_{jj} \\
&\vdots
\end{aligned}$$

Solving $Ax = b$

Similarly, we can use forward substitution for a lower triangular system $Lx = b$

$$\begin{aligned}
x_1 &= b_1/\ell_{11} \\
x_2 &= (b_2 - \ell_{21}x_1)/\ell_{22} \\
&\vdots \\
x_j &= \left(b_j - \sum_{k=1}^{j-1} \ell_{jk}x_k\right)/\ell_{jj} \\
&\vdots
\end{aligned}$$

Solving $Ax = b$

Back and forward substitution can be implemented with doubly nested for-loops

The computational work is dominated by evaluating the sum $\sum_{k=1}^{j-1} \ell_{jk} x_k$, $j = 1, \ldots, n$

We have $j - 1$ additions and multiplications in this loop for each $j = 1, \ldots, n$, *i.e.* $2(j-1)$ operations for each j

Hence the total number of floating point operations in back or forward substitution is asymptotic to:

$$2 \sum_{j=1}^{n} j = 2n(n+1)/2 \sim n^2$$

Solving $Ax = b$

Here "$\sim$" refers to asymptotic behavior, *e.g.*

$$f(n) \sim n^2 \iff \lim_{n\to\infty} \frac{f(n)}{n^2} = 1$$

We often also use "big-O" notation, *e.g.* for remainder terms in Taylor expansion

$f(x) = O(g(x))$ if there exists $M \in \mathbb{R}_{>0}, x_0 \in \mathbb{R}$ such that $|f(x)| \leq M|g(x)|$ for all $x \geq x_0$

In the present context we prefer "$\sim$" since it indicates the correct scaling of the leading-order term

e.g. let $f(n) \equiv n^2/4 + n$, then $f(n) = O(n^2)$, whereas $f(n) \sim n^2/4$

Solving $Ax = b$

So transforming $Ax = b$ to a triangular system is a sensible goal, but how do we achieve it?

Observation: If we premultiply $Ax = b$ by a nonsingular matrix M then the new system $MAx = Mb$ has the same solution

Hence, want to devise a sequence of matrices $M_1, M_2, \cdots, M_{n-1}$ such that $MA \equiv M_{n-1} \cdots M_1 A \equiv U$ is upper triangular

This process is Gaussian Elimination, and gives the transformed system $Ux = Mb$

LU Factorization

We will show shortly that it turns out that if $MA = U$, then we have that $L \equiv M^{-1}$ is lower triangular

Therefore we obtain $A = LU$: product of lower and upper triangular matrices

This is the LU factorization of A

LU Factorization

LU factorization is the most common way of solving linear systems!

$Ax = b \Longleftrightarrow LUx = b$

Let $y \equiv Ux$, then $Ly = b$: solve for y via forward substitution[3]

Then solve for $Ux = y$ via back substitution

[3] $y = L^{-1}b$ is the transformed right-hand side vector (*i.e.* Mb from earlier) that we are familiar with from Gaussian elimination

LU Factorization

Next question: How should we determine $M_1, M_2, \cdots, M_{n-1}$?

We need to be able to annihilate selected entries of A, below the diagonal in order to obtain an upper-triangular matrix

To do this, we use “elementary elimination matrices”

Let L_j denote j^{th} elimination matrix (we use “L_j” rather than “M_j” from now on as elimination matrices are lower triangular)

LU Factorization

Let $X (\equiv L_{j-1}L_{j-2}\cdots L_1 A)$ denote matrix at the start of step j, and let $x_{(:,j)} \in \mathbb{R}^n$ denote column j of X

Then we define L_j such that

$$L_j x_{(:,j)} \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -x_{j+1,j}/x_{jj} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -x_{nj}/x_{jj} & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_{1j} \\ \vdots \\ x_{jj} \\ x_{j+1,j} \\ \vdots \\ x_{nj} \end{bmatrix} = \begin{bmatrix} x_{1j} \\ \vdots \\ x_{jj} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

LU Factorization

To simplify notation, we let $\ell_{ij} \equiv \frac{x_{ij}}{x_{jj}}$ in order to obtain

$$L_j \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\ell_{nj} & 0 & \cdots & 1 \end{bmatrix}$$

LU Factorization

Using elementary elimination matrices we can reduce A to upper triangular form, one column at a time

Schematically, for a 4×4 matrix, we have

$$\underbrace{\left[\begin{array}{cccc} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{array}\right]}_{A} \xrightarrow{L_1} \underbrace{\left[\begin{array}{cccc} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{array}\right]}_{L_1 A} \xrightarrow{L_2} \underbrace{\left[\begin{array}{cccc} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{array}\right]}_{L_2 L_1 A}$$

Key point: L_k does not affect columns $1, 2, \ldots, k-1$ of $L_{k-1}L_{k-2}\ldots L_1 A$

LU Factorization

After $n-1$ steps, we obtain the upper triangular matrix $U = L_{n-1}\cdots L_2 L_1 A$

$$U = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

LU Factorization

Finally, we wish to form the factorization $A = LU$, hence we need $L = (L_{n-1} \cdots L_2 L_1)^{-1} = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$

This turns out to be surprisingly simple due to two strokes of luck!

First stroke of luck: L_j^{-1} is obtained simply by negating the subdiagonal entries of L_j

$$L_j \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\ell_{nj} & 0 & \cdots & 1 \end{bmatrix}, \quad L_j^{-1} \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & \ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \ell_{nj} & 0 & \cdots & 1 \end{bmatrix}$$

LU Factorization

Explanation: Let $\ell_j \equiv [0, \ldots, 0, \ell_{j+1,j}, \ldots, \ell_{nj}]^T$ so that
$L_j = \mathrm{I} - \ell_j e_j^T$

Now consider $L_j(\mathrm{I} + \ell_j e_j^T)$:

$$L_j(\mathrm{I}+\ell_j e_j^T) = (\mathrm{I}-\ell_j e_j^T)(\mathrm{I}+\ell_j e_j^T) = \mathrm{I}-\ell_j e_j^T \ell_j e_j^T = \mathrm{I}-\ell_j(e_j^T \ell_j)e_j^T$$

Also, $(e_j^T \ell_j) = 0$ (why?) so that $L_j(\mathrm{I} + \ell_j e_j^T) = \mathrm{I}$

By the same argument $(\mathrm{I} + \ell_j e_j^T)L_j = \mathrm{I}$, and hence
$L_j^{-1} = (\mathrm{I} + \ell_j e_j^T)$

LU Factorization

Next we want to form the matrix $L \equiv L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$

Note that we have

$$\begin{aligned} L_j^{-1} L_{j+1}^{-1} &= (\mathrm{I} + \ell_j e_j^T)(\mathrm{I} + \ell_{j+1} e_{j+1}^T) \\ &= \mathrm{I} + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T + \ell_j (e_j^T \ell_{j+1}) e_{j+1}^T \\ &= \mathrm{I} + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T \end{aligned}$$

Interestingly, this convenient result doesn't hold for $L_{j+1}^{-1} L_j^{-1}$, why?

LU Factorization

Similarly,

$$\begin{aligned} L_j^{-1} L_{j+1}^{-1} L_{j+2}^{-1} &= (\mathrm{I} + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T)(\mathrm{I} + \ell_{j+2} e_{j+2}^T) \\ &= \mathrm{I} + \ell_j e_j^T + \ell_{j+1} e_{j+1}^T + \ell_{j+2} e_{j+2}^T \end{aligned}$$

That is, to compute the product $L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$ we simply collect the subdiagonals for $j = 1, 2, \ldots, n-1$

LU Factorization

Hence, second stroke of luck:

$$L \equiv L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1} = \begin{bmatrix} 1 & & & & \\ \ell_{21} & 1 & & & \\ \ell_{31} & \ell_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ \ell_{n1} & \ell_{n2} & \cdots & \ell_{n,n-1} & 1 \end{bmatrix}$$

LU Factorization

Therefore, basic LU factorization algorithm is

```
1: $U = A$, $L = \mathrm{I}$
2: for $j = 1 : n - 1$ do
3:     for $i = j + 1 : n$ do
4:         $\ell_{ij} = u_{ij}/u_{jj}$
5:         for $k = j : n$ do
6:             $u_{ik} = u_{ik} - \ell_{ij}u_{jk}$
7:         end for
8:     end for
9: end for
```

Note that the entries of U are updated each iteration so at the start of step j, $U = L_{j-1}L_{j-2}\cdots L_1 A$

Here line 4 comes straight from the definition $\ell_{ij} \equiv \frac{u_{ij}}{u_{jj}}$

LU Factorization

Line 6 accounts for the effect of L_j on columns $k = j, j+1, \ldots, n$ of U

For $k = j : n$ we have

$$L_j u_{(:,k)} \equiv \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\ell_{j+1,j} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\ell_{nj} & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{1k} \\ \vdots \\ u_{jk} \\ u_{j+1,k} \\ \vdots \\ u_{nk} \end{bmatrix} = \begin{bmatrix} u_{1k} \\ \vdots \\ u_{jk} \\ u_{j+1,k} - \ell_{j+1,j} u_{jk} \\ \vdots \\ u_{nk} - \ell_{nj} u_{jk} \end{bmatrix}$$

The vector on the right is the updated k^{th} column of U, which is computed in line 6

LU Factorization

LU Factorization involves a triply-nested for-loop, hence $O(n^3)$ calculations

Careful operation counting shows LU factorization requires $\sim \frac{1}{3}n^3$ additions and $\sim \frac{1}{3}n^3$ multiplications, $\sim \frac{2}{3}n^3$ operations in total

Solving a linear system using LU

Hence to solve $Ax = b$, we perform the following three steps:

Step 1: Factorize A into L and U: $\sim \frac{2}{3}n^3$

Step 2: Solve $Ly = b$ by forward substitution: $\sim n^2$

Step 3: Solve $Ux = y$ by back substitution: $\sim n^2$

Total work is dominated by Step 1, $\sim \frac{2}{3}n^3$

Solving a linear system using LU

An alternative approach would be to compute A^{-1} explicitly and evaluate $x = A^{-1}b$, but this is a bad idea!

Question: How would we compute A^{-1}?

Solving a linear system using LU

Answer: Let $a^{\text{inv}}_{(:,k)}$ denote the kth column of A^{-1}, then $a^{\text{inv}}_{(:,k)}$ must satisfy

$$Aa^{\text{inv}}_{(:,k)} = e_k$$

Therefore to compute A^{-1}, we first LU factorize A, then back/forward substitute for rhs vector e_k, $k = 1, 2, \ldots, n$

The n back/forward substitutions alone require $\sim 2n^3$ operations, inefficient!

A rule of thumb in Numerical Linear Algebra: It is almost always a bad idea to compute A^{-1} explicitly

Solving a linear system using LU

Another case where LU factorization is very helpful is if we want to solve $Ax = b_i$ for several different right-hand sides b_i, $i = 1, \ldots, k$

We incur the $\sim \frac{2}{3}n^3$ cost only once, and then each subequent forward/back subsitution costs only $\sim 2n^2$

Makes a huge difference if n is large!

Stability of Gaussian Elimination

There is a problem with the LU algorithm presented above

Consider the matrix

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

A is nonsingular, well-conditioned ($\kappa(A) \approx 2.62$) but LU factorization fails at first step (division by zero)

Stability of Gaussian Elimination

LU factorization doesn't fail for

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}$$

but we get

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}$$

Stability of Gaussian Elimination

Let's suppose that $-10^{20} \in \mathbb{F}$ (a floating point number) and that $\text{round}(1 - 10^{20}) = -10^{20}$

Then in finite precision arithmetic we get

$$\widetilde{L} = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \qquad \widetilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}$$

Stability of Gaussian Elimination

Hence due to rounding error we obtain

$$\widetilde{L}\widetilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix}$$

which is not close to

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}$$

Then, for example, let $b = [3, 3]^T$

- Using $\widetilde{L}\widetilde{U}$, we get $\tilde{x} = [3, 3]^T$
- True answer is $x = [0, 3]^T$

Hence large relative error (rel. err. $= 1$) even though the problem is well-conditioned

Stability of Gaussian Elimination

In this example, standard Gaussian elimination yields a large residual

Or equivalently, it yields the exact solution to a problem corresponding to a large input perturbation: $\Delta b = [0, 3]^T$

Hence unstable algorithm! In this case the cause of the large error in x is numerical instability, not ill-conditioning

To stabilize Gaussian elimination, we need to permute rows, *i.e.* perform pivoting

Pivoting

Recall the Gaussian elimination process

$$\begin{bmatrix} \times & \times & \times & \times \\ & x_{jj} & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ & x_{jj} & \times & \times \\ & 0 & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

But we could just as easily do

$$\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & x_{ij} & \times & \times \\ & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ & 0 & \times & \times \\ & x_{ij} & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

Partial Pivoting

The entry x_{ij} is called the pivot, and flexibility in choosing the pivot is essential otherwise we can't deal with:

$$A = \left[\begin{array}{cc} 0 & 1 \\ 1 & 1 \end{array} \right]$$

From a numerical stability point of view, it is crucial to choose the pivot to be the largest entry in column j: "partial pivoting"[4]

This ensures that each ℓ_{ij} entry — which acts as a multiplier in the LU factorization process — satisfies $|\ell_{ij}| \leq 1$

[4]Full pivoting refers to searching through columns $j : n$ for the largest entry; this is more expensive and only marginal benefit to stability in practice

Partial Pivoting

To maintain the triangular LU structure, we permute rows by premultiplying by permutation matrices

$$\begin{bmatrix} \times & \times & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \\ & x_{ij} & \times & \times \end{bmatrix} \xrightarrow{P_1} \begin{bmatrix} \times & \times & \times & \times \\ & x_{ij} & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \end{bmatrix} \xrightarrow{L_1} \begin{bmatrix} \times & \times & \times & \times \\ & x_{ij} & \times & \times \\ & 0 & \times & \times \\ & 0 & \times & \times \end{bmatrix}$$

Pivot selection   Row interchange

In this case

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

and each P_j is obtained by swapping two rows of I

Partial Pivoting

Therefore, with partial pivoting we obtain

$$L_{n-1}P_{n-1}\cdots L_2P_2L_1P_1A = U$$

It can be shown (we omit the details here, see Trefethen & Bau) that this can be rewritten as

$$PA = LU$$

where[5] $P \equiv P_{n-1}\cdots P_2P_1$

Theorem: Gaussian elimination with partial pivoting produces nonsingular factors L and U if and only if A is nonsingular.

[5]The L matrix here is lower triangular, but not the same as L in the non-pivoting case: we have to account for the row swaps

Partial Pivoting

Pseudocode for LU factorization with partial pivoting (blue text is new):

```
1:  $U = A$, $L = \mathrm{I}$, $P = \mathrm{I}$
2:  for $j = 1 : n - 1$ do
3:      Select $i (\geq j)$ that maximizes $|u_{ij}|$
4:      Interchange rows of $U$: $u_{(j,j:n)} \leftrightarrow u_{(i,j:n)}$
5:      Interchange rows of $L$: $\ell_{(j,1:j-1)} \leftrightarrow \ell_{(i,1:j-1)}$
6:      Interchange rows of $P$: $p_{(j,:)} \leftrightarrow p_{(i,:)}$
7:      for $i = j + 1 : n$ do
8:          $\ell_{ij} = u_{ij} / u_{jj}$
9:          for $k = j : n$ do
10:             $u_{ik} = u_{ik} - \ell_{ij} u_{jk}$
11:         end for
12:     end for
13: end for
```

Again this requires $\sim \frac{2}{3} n^3$ floating point operations

Partial Pivoting: Solve $Ax = b$

To solve $Ax = b$ using the factorization $PA = LU$:

- Multiply through by P to obtain $PAx = LUx = Pb$
- Solve $Ly = Pb$ using forward substitution
- Then solve $Ux = y$ using back substitution

Partial Pivoting in Python

Python's `scipy.linalg.lu` function can do LU factorization with pivoting.

```
Python 3.8.5 (default, Sep  6 2020, 03:54:05)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> import scipy.linalg
>>> a=np.random.random((4,4))
>>> a
array([[0.7565817 , 0.69602842, 0.46957017, 0.51744408],
       [0.91458599, 0.6853307 , 0.0949905 , 0.41508579],
       [0.29342843, 0.833517  , 0.7024628 , 0.16247883],
       [0.33738185, 0.56514679, 0.89047111, 0.62546956]])
>>> (p,l,u)=scipy.linalg.lu(a)
>>> p
array([[0., 0., 0., 1.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
>>> l
array([[1.        , 0.        , 0.        , 0.        ],
       [0.32083197, 1.        , 0.        , 0.        ],
       [0.36889024, 0.50898649, 1.        , 0.        ],
       [0.82723954, 0.21037669, 0.48621147, 1.        ]])
>>> u
array([[ 0.91458599,  0.6853307 ,  0.0949905 ,  0.41508579],
       [ 0.        ,  0.613641  ,  0.67198681,  0.02930604],
       [ 0.        ,  0.        ,  0.51339783,  0.45743209],
       [ 0.        ,  0.        ,  0.        , -0.05450533]])
```

Stability of Gaussian Elimination

Numerical stability of Gaussian Elimination has been an important research topic since the 1940s

Major figure in this field: James H. Wilkinson (English numerical analyst, 1919–1986)

Showed that for $Ax = b$ with $A \in \mathbb{R}^{n \times n}$:

- Gaussian elimination without partial pivoting is numerically unstable (as we've already seen)
- Gaussian elimination with partial pivoting satisfies

$$\frac{\|r\|}{\|A\|\|x\|} \leq 2^{n-1} n^2 \epsilon_{\text{mach}}$$

Stability of Gaussian Elimination

That is, pathological cases exist where the relative residual, $\|r\|/\|A\|\|x\|$, grows exponentially with n due to rounding error

Worst case behavior of Gaussian Elimination with partial pivoting is explosive instability but such pathological cases are extremely rare!

In over 50 years of Scientific Computation, instability has only been encountered due to deliberate construction of pathological cases

In practice, Gaussian elimination is stable in the sense that it produces a small relative residual

Stability of Gaussian Elimination

In practice, we typically obtain

$$\frac{\|r\|}{\|A\|\|x\|} \lesssim n\epsilon_{\text{mach}},$$

i.e. grows only linearly with *n*, and is scaled by ϵ_{mach}

Combining this result with our inequality:

$$\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A)\frac{\|r\|}{\|A\|\|x\|}$$

implies that in practice Gaussian elimination gives small error for well-conditioned problems!

Cholesky Factorization

Cholesky factorization

Suppose that $A \in \mathbb{R}^{n \times n}$ is an "SPD" matrix, *i.e.*:

- Symmetric: $A^T = A$
- Positive Definite: for any $v \neq 0$, $v^T A v > 0$

Then the LU factorization of A can be arranged so that $U = L^T$, *i.e.* $A = LL^T$ (but in this case L may not have 1s on the diagonal)

Consider the 2×2 case:

$$\begin{bmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 \\ \ell_{21} & \ell_{22} \end{bmatrix} \begin{bmatrix} \ell_{11} & \ell_{21} \\ 0 & \ell_{22} \end{bmatrix}$$

Equating entries gives

$$\ell_{11} = \sqrt{a_{11}}, \quad \ell_{21} = a_{21}/\ell_{11}, \quad \ell_{22} = \sqrt{a_{22} - \ell_{21}^2}$$

Cholesky factorization

This approach of equating entries can be used to derive the Cholesky factorization for the general $n \times n$ case

```
$L = A$
**for** $j = 1 : n$ **do**
    $\ell_{jj} = \sqrt{\ell_{jj}}$
    **for** $i = j + 1 : n$ **do**
        $\ell_{ij} = \ell_{ij} / \ell_{jj}$
    **end for**
    **for** $k = j + 1 : n$ **do**
        **for** $i = k : n$ **do**
            $\ell_{ik} = \ell_{ik} - \ell_{ij}\ell_{kj}$
        **end for**
    **end for**
**end for**
```

Cholesky factorization

Notes on Cholesky factorization:

- For an SPD matrix A, Cholesky factorization is numerically stable and does not require any pivoting
- Operation count: $\sim \frac{1}{3}n^3$ operations in total, *i.e.* about half as many as Gaussian elimination
- Only need to store L, hence uses less memory than LU

Timing algorithms

Up to now, we have shown that for a matrix $A \in \mathbb{R}^{n \times n}$, the LU factorization requires $\sim \frac{2}{3}n^3$ operations

If A is symmetric and positive definite, then the Cholesky factorization can be performed in $\sim \frac{1}{3}n^3$ operations

These asymptotic expressions were found by analyzing the algorithms and counting the floating point operations required

It is useful to examine the real-world performance of these algorithms, since various computer hardware factors (*e.g.* memory bandwidth, caching) may be important

Two measures of time

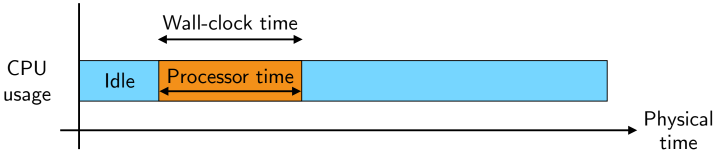
There are different ways to think about time taken on a computer, and we'll consider two

Wall-clock time measures the time as perceived by the computer user (*i.e.* by looking at the clock on the wall)

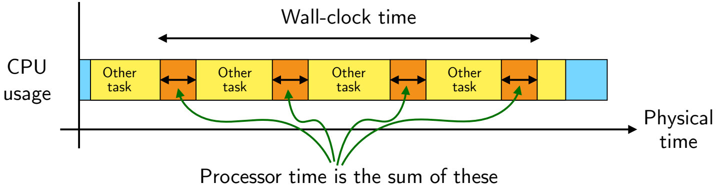
Processor time measures the time that a program spends being processed on a CPU

How the two timing measures differ

A job (shown in orange) is run on an idle CPU core. The wall-clock time and processor time are similar.



If the CPU core is under load and running other tasks, the job will be rapidly switched on and off the core. Wall-clock time will increase but processor time will be similar.



Examples of timing

Which timing method should we use?

Wall-clock time is often the most important aspect to the user

But as the example shows, processor time is more insensitive to various real-world factors

Often it is useful to have both measurements available to us

Modern CPU features

Most modern CPUs have multiple cores that can process independent jobs simultaneously

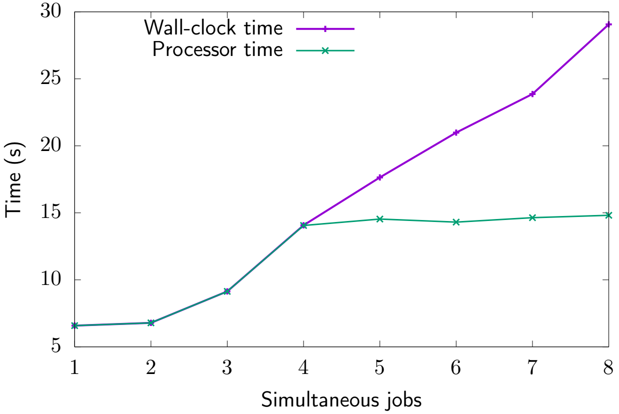
Laptops typically have CPUs with 2–8 cores, and desktops have CPUs with 4–16 cores

In addition, over the last decade CPU makers such as Intel and AMD have developed a technology called hyperthreading, where each physical core appears to the system as two virtual cores

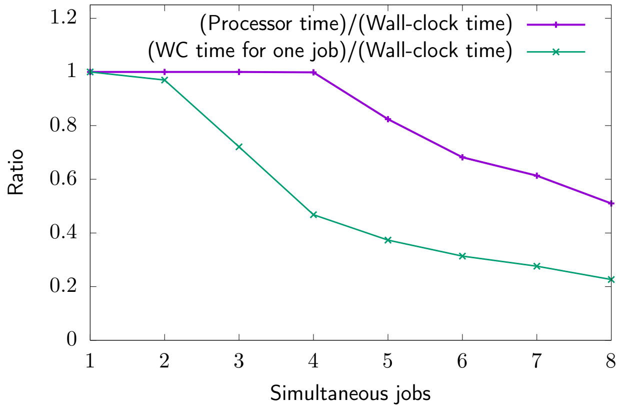
Two programs are rapidly switched on the same physical core. Can be good for system responsiveness, but physical cores usually the most important for scientific computing performance.

Timing example

[*timing_test.py*] Comparing wall-clock time and processor time for a sample calculation



Timing example

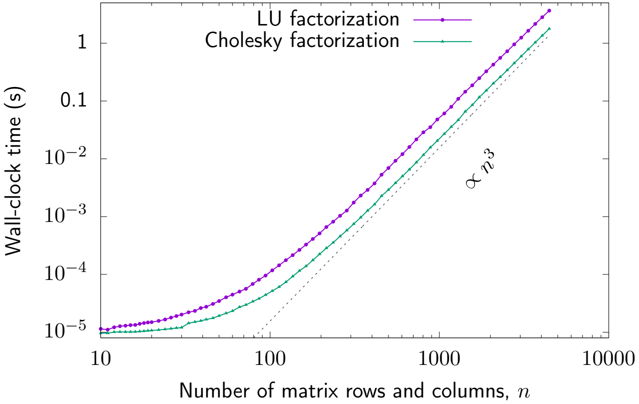


(Proc. time)/(WC time) starts to decrease after 4 jobs (*i.e* once the number of virtual cores is exceeded)

But comparing WC time for *n* jobs with the WC time for one job, shows that actual performance is lowered after 2 jobs (*i.e.* once the number of physical cores is exceeded)

Timing numerical linear algebra routines

[*lu_time.py* and *chol_time.py*] Measuring the time of the LU and Cholesky factorizations



Sparse Matrices

In applications, we often encounter sparse matrices

A prime example is in discretization of partial differential equations (covered in the next section)

"Sparse matrix" is not precisely defined, roughly speaking it is a matrix that is "mostly zeros"

From a computational point of view it is advantageous to store only the non-zero entries

The set of non-zero entries of a sparse matrix is referred to as its sparsity pattern

Sparse Matrices

```
 A =
       2   2   2   2   2
       0   1   0   0   0
       0   0   1   0   0
       0   0   0   1   0
       0   0   0   0   1

A_sparse =
  (1,1)   2
  (1,2)   2
  (2,2)   1
  (1,3)   2
  (3,3)   1
  (1,4)   2
  (4,4)   1
  (1,5)   2
  (5,5)   1
```

Sparse Matrices

From a mathematical point of view, sparse matrices are no different from dense matrices

But from Sci. Comp. perspective, sparse matrices require different data structures and algorithms for computational efficiency

e.g., can apply LU or Cholesky to sparse A, but "new" non-zeros (*i.e.* outside sparsity pattern of A) are introduced in the factors

These new non-zero entries are called "fill-in" — many methods exist for reducing fill-in by permuting rows and columns of A

QR Factorization

A square matrix $Q \in \mathbb{R}^{n \times n}$ is called orthogonal if its columns and rows are orthonormal vectors

Equivalently, $Q^T Q = QQ^T = \mathrm{I}$

Orthogonal matrices preserve the Euclidean norm of a vector, *i.e.*

$$\|Qv\|_2^2 = v^T Q^T Q v = v^T v = \|v\|_2^2$$

Hence, geometrically, we picture orthogonal matrices as reflection or rotation operators

Orthogonal matrices are very important in scientific computing, norm-preservation implies no amplification of numerical error!

QR Factorization

A matrix $A \in \mathbb{R}^{m \times n}$, $m \geq n$, can be factorized into

$$A = QR$$

where

- $Q \in \mathbb{R}^{m \times m}$ is orthogonal
- $R \equiv \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} \in \mathbb{R}^{m \times n}$
- $\hat{R} \in \mathbb{R}^{n \times n}$ is upper-triangular

QR is very good for solving overdetermined linear least-squares problems, $Ax \simeq b$ [6]

[6] QR can also be used to solve a square system $Ax = b$, but requires $\sim 2\times$ as many operations as Gaussian elimination hence not the standard choice

QR Factorization

To see why, consider the 2-norm of the least squares residual:

$$\begin{aligned}
\|r(x)\|_2^2 &= \|b - Ax\|_2^2 = \|b - Q \left[\begin{array}{c} \hat{R} \\ 0 \end{array} \right] x\|_2^2 \\
&= \|Q^T \left(b - Q \left[\begin{array}{c} \hat{R} \\ 0 \end{array} \right] x \right) \|_2^2 \\
&= \|Q^T b - \left[\begin{array}{c} \hat{R} \\ 0 \end{array} \right] x\|_2^2
\end{aligned}$$

(We used the fact that $\|Q^T z\|_2 = \|z\|_2$ in the second line)

QR Factorization

Then, let $Q^T b = [c_1, c_2]^T$ where $c_1 \in \mathbb{R}^n, c_2 \in \mathbb{R}^{m-n}$, so that

$$\|r(x)\|_2^2 = \|c_1 - \hat{R}x\|_2^2 + \|c_2\|_2^2$$

Question: Based on this expression, how do we minimize $\|r(x)\|_2$?

QR Factorization

Answer: We can't influence the second term, $\|c_2\|_2^2$, since it doesn't contain an x

Hence we minimize $\|r(x)\|_2^2$ by making the first term zero

That is, we solve the $n \times n$ triangular system $\hat{R}x = c_1$ — this what Python does in its `lstsq` function for solving least squares

Also, this tells us that $\min_{x \in \mathbb{R}^n} \|r(x)\|_2 = \|c_2\|_2$

QR Factorization

Recall that solving linear least-squares via the normal equations requires solving a system with the matrix $A^T A$

But using the normal equations directly is problematic since $\text{cond}(A^T A) = \text{cond}(A)^2$ (this is a consequence of the SVD, which we'll cover soon)

The QR approach avoids this condition-number-squaring effect and is much more numerically stable!

QR Factorization

How do we compute the QR Factorization?

There are three main methods

- Gram–Schmidt orthogonalization
- Householder triangularization
- Givens rotations

Gram–Schmidt Orthogonalization

Suppose $A \in \mathbb{R}^{m \times n}$, $m \geq n$

One way to picture the QR factorization is to construct a sequence of orthonormal vectors $q_1, q_2, \ldots$ such that

$$\text{span}\{q_1, q_2, \ldots, q_j\} = \text{span}\{a_{(:,1)}, a_{(:,2)}, \ldots, a_{(:,j)}\}, \quad j = 1, \ldots, n$$

We seek coefficients r_{ij} such that

$$\begin{aligned} a_{(:,1)} &= r_{11} q_1, \\ a_{(:,2)} &= r_{12} q_1 + r_{22} q_2, \\ &\vdots \\ a_{(:,n)} &= r_{1n} q_1 + r_{2n} q_2 + \cdots + r_{nn} q_n. \end{aligned}$$

This can be done via the Gram–Schmidt process, as we'll discuss shortly

Gram–Schmidt Orthogonalization

In matrix form we have:

$$\left[\begin{array}{c|c|c|c} & & & \\ a_{(:,1)} & a_{(:,2)} & \cdots & a_{(:,n)} \\ & & & \end{array}\right] = \left[\begin{array}{c|c|c|c} & & & \\ q_1 & q_2 & \cdots & q_n \\ & & & \end{array}\right] \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix}$$

This gives $A = \hat{Q}\hat{R}$ for $\hat{Q} \in \mathbb{R}^{m\times n}$, $\hat{R} \in \mathbb{R}^{n\times n}$

This is called the reduced QR factorization of A, which is slightly different from the definition we gave earlier

Note that for $m > n$, $\hat{Q}^T\hat{Q} = \mathrm{I}$, but $\hat{Q}\hat{Q}^T \neq \mathrm{I}$ (the latter is why the full QR is sometimes nice)

Full vs Reduced QR Factorization

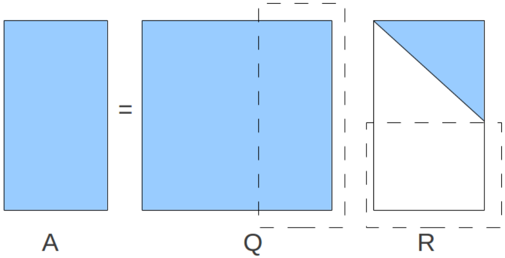
The full QR factorization (defined earlier)

$$A = QR$$

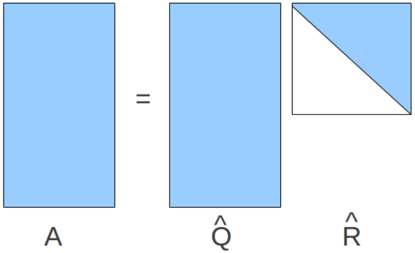
is obtained by appending $m - n$ arbitrary orthonormal columns to $\hat{Q}$ to make it an $m \times m$ orthogonal matrix

We also need to append rows of zeros to $\hat{R}$ to "silence" the last $m - n$ columns of Q, to obtain $R = \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix}$

Full vs Reduced QR Factorization



Full QR



Reduced QR

Full vs Reduced QR Factorization

Exercise: Show that the linear least-squares solution is given by $\hat{R}x = \hat{Q}^T b$ by plugging $A = \hat{Q}\hat{R}$ into the Normal Equations

This is equivalent to the least-squares result we showed earlier using the full QR factorization, since $c_1 = \hat{Q}^T b$

Full versus Reduced QR Factorization

In Python, `numpy.linalg.qr` gives the reduced QR factorization by default

```
Python 3.8.5 (default, Sep  6 2020, 03:54:05)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((5,3))
>>> (q,r)=np.linalg.qr(a)
>>> q
array([[-0.60657572,  0.29216376, -0.14455158],
       [-0.23608755, -0.66242317, -0.48304524],
       [-0.29339096, -0.36132885, -0.25299114],
       [-0.35279836, -0.43526372,  0.82530864],
       [-0.60480048,  0.39474401, -0.02516495]])
>>> r
array([[-1.63178556, -1.29232138, -0.92550283],
       [ 0.        , -0.93369372, -0.25039136],
       [ 0.        ,  0.        ,  0.36307913]])
```

Full versus Reduced QR Factorization

In Python, supplying the `mode='complete'` option gives the complete QR factorization

```
Python 3.8.5 (default, Sep  6 2020, 03:54:05)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((5,3))
>>> (q,r)=np.linalg.qr(a,mode='complete')
>>> q
array([[-0.60657572,  0.29216376, -0.14455158, -0.2162523 , -0.6921315 ],
       [-0.23608755, -0.66242317, -0.48304524, -0.48923968,  0.18102498],
       [-0.29339096, -0.36132885, -0.25299114,  0.84156254, -0.10550441],
       [-0.35279836, -0.43526372,  0.82530864, -0.06510356, -0.02657044],
       [-0.60480048,  0.39474401, -0.02516495,  0.03759649,  0.6901788 ]])
>>> r
array([[-1.63178556, -1.29232138, -0.92550283],
       [ 0.        , -0.93369372, -0.25039136],
       [ 0.        ,  0.        ,  0.36307913],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ]])
```

Gram–Schmidt Orthogonalization

Returning to the Gram–Schmidt process, how do we compute the q_i, $i = 1, \ldots, n$?

In the jth step, find a unit vector $q_j \in \text{span}\{a_{(:,1)}, a_{(:,2)}, \ldots, a_{(:,j)}\}$ that is orthogonal to $\text{span}\{q_1, q_n, \ldots, q_{j-1}\}$

We set

$$v_j \equiv a_{(:,j)} - (q_1^T a_{(:,j)})q_1 - \cdots - (q_{j-1}^T a_{(:,j)})q_{j-1},$$

and then $q_j \equiv v_j / \|v_j\|_2$ satisfies our requirements

We can now determine the required values of r_{ij}

Gram–Schmidt Orthogonalization

We then write our set of equations for the q_i as

$$\begin{aligned}
q_1 &= \frac{a_{(:,1)}}{r_{11}}, \\
q_2 &= \frac{a_{(:,2)} - r_{12} q_1}{r_{22}}, \\
&\vdots \\
q_n &= \frac{a_{(:,n)} - \sum_{i=1}^{n-1} r_{in} q_i}{r_{nn}}.
\end{aligned}$$

Then from the definition of q_j, we see that

$$\begin{aligned}
r_{ij} &= q_i^T a_{(:,j)}, \qquad\qquad i \neq j \\
|r_{jj}| &= \|a_{(:,j)} - \sum_{i=1}^{j-1} r_{ij} q_i\|_2
\end{aligned}$$

The sign of r_{jj} is not determined uniquely, *e.g.* we could choose $r_{jj} > 0$ for each j

Classical Gram–Schmidt Process

The Gram–Schmidt algorithm we have described is provided in the pseudocode below

```
1: for $j = 1 : n$ do
2:     $v_j = a_{(:,j)}$
3:     for $i = 1 : j-1$ do
4:         $r_{ij} = q_i^T a_{(:,j)}$
5:         $v_j = v_j - r_{ij} q_i$
6:     end for
7:     $r_{jj} = \|v_j\|_2$
8:     $q_j = v_j / r_{jj}$
9: end for
```

This is referred to the classical Gram–Schmidt (CGS) method

Gram–Schmidt Orthogonalization

The only way the Gram–Schmidt process can fail is if $|r_{jj}| = \|v_j\|_2 = 0$ for some j

This can only happen if $a_{(:,j)} = \sum_{i=1}^{j-1} r_{ij} q_i$ for some j, *i.e.* if $a_{(:,j)} \in \operatorname{span}\{q_1, q_n, \ldots, q_{j-1}\} = \operatorname{span}\{a_{(:,1)}, a_{(:,2)}, \ldots, a_{(:,j-1)}\}$

This means that columns of A are linearly dependent

Therefore, Gram–Schmidt fails $\implies$ cols. of A linearly dependent

Gram–Schmidt Orthogonalization

Equivalently, by contrapositive: cols. of A linearly independent $\implies$ Gram–Schmidt succeeds

Theorem: Every $A \in \mathbb{R}^{m\times n} (m \geq n)$ of full rank has a unique reduced QR factorization $A = \hat{Q}\hat{R}$ with $r_{ii} > 0$

The only non-uniqueness in the Gram–Schmidt process was in the sign of r_{ii}, hence $\hat{Q}\hat{R}$ is unique if $r_{ii} > 0$

Gram–Schmidt Orthogonalization

Theorem: Every $A \in \mathbb{R}^{m \times n} (m \geq n)$ has a full QR factorization.

Case 1: A has full rank

- We compute the reduced QR factorization from above
- To make Q square we pad $\hat{Q}$ with $m - n$ arbitrary orthonormal columns
- We also pad $\hat{R}$ with $m - n$ rows of zeros to get R

Case 2: A doesn't have full rank

- At some point in computing the reduced QR factorization, we encounter $\|v_j\|_2 = 0$
- At this point we pick an arbitrary q_j orthogonal to $\text{span}\{q_1, q_2, \ldots, q_{j-1}\}$ and then proceed as in Case 1

Modified Gram–Schmidt Process

The classical Gram–Schmidt process is numerically unstable!
(sensitive to rounding error, orthogonality of the q_j degrades)

The algorithm can be reformulated to give the modified Gram–Schmidt process, which is numerically more robust

Key idea: when each new q_j is computed, orthogonalize each remaining column of A against it

Modified Gram–Schmidt Process

Modified Gram–Schmidt (MGS):

```
1: for $i = 1 : n$ do
2:     $v_i = a_{(:,i)}$
3: end for
4: for $i = 1 : n$ do
5:     $r_{ii} = \|v_i\|_2$
6:     $q_i = v_i / r_{ii}$
7:     for $j = i + 1 : n$ do
8:         $r_{ij} = q_i^T v_j$
9:         $v_j = v_j - r_{ij} q_i$
10:    end for
11: end for
```

Modified Gram–Schmidt Process

Key difference between MGS and CGS:

- In CGS we compute orthogonalization coefficients r_{ij} wrt the "raw" vector $a_{(:,j)}$
- In MGS we remove components of $a_{(:,j)}$ in $\text{span}\{q_1, q_2, \ldots, q_{i-1}\}$ before computing r_{ij}

This makes no difference mathematically: In exact arithmetic components in $\text{span}\{q_1, q_2, \ldots, q_{i-1}\}$ are annihilated by q_i^T

But in practice it reduces degradation of orthogonality of the q_j
$\Longrightarrow$ superior numerical stability of MGS over CGS

Operation Count

Work in MGS is dominated by lines 8 and 9, the innermost loop:

$$\begin{aligned} r_{ij} &= q_i^T v_j \\ v_j &= v_j - r_{ij} q_i \end{aligned}$$

First line requires m multiplications, $m - 1$ additions; second line requires m multiplications, m subtractions

Hence $\sim 4m$ operations per single inner iteration

Hence total number of operations is asymptotic to

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} 4m \sim 4m \sum_{i=1}^{n} i \sim 2mn^2$$

Alternative QR computation methods

The QR factorization can also be computed using Householder triangularization and Givens rotations.

Both methods take the approach of applying a sequence of orthogonal matrices $Q_1, Q_2, Q_3, \ldots$ to the matrix that successively remove terms below the diagonal (similar to the method employed by the LU factorization).

Householder Triangularization

We will now discuss Householder[7] triangularization

The Householder algorithm is more numerically stable and more efficient than Gram–Schmidt

But Gram–Schmidt allows us to build up orthogonal basis for successive spaces spanned by columns of A

$$\text{span}\{a_{(:,1)}\}, \text{span}\{a_{(:,1)}, a_{(:,2)}\}, \ldots$$

which can be important in some cases

[7]Alston Householder, 1904–1993, American numerical analyst

Householder Triangularization

Householder idea: Apply a succession of orthogonal matrices $Q_k \in \mathbb{R}^{m \times m}$ to A to compute upper triangular matrix R

$$Q_n \cdots Q_2 Q_1 A = R$$

Hence we obtain the full QR factorization $A = QR$, where $Q \equiv Q_1^T Q_2^T \ldots Q_n^T$

Householder Triangularization

In 1958, Householder proposed a way to choose Q_k to introduce zeros below diagonal in col. k while preserving previous zeros

$$\underset{A}{\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}} \xrightarrow{Q_1} \underset{Q_1A}{\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix}} \xrightarrow{Q_2} \underset{Q_2Q_1A}{\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix}} \xrightarrow{Q_3} \underset{Q_3Q_2Q_1A}{\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}}$$

This is achieved by Householder reflectors

Householder Reflectors

We choose

$$Q_k = \begin{bmatrix} \mathrm{I} & 0 \\ 0 & F \end{bmatrix}$$

where

- $\mathrm{I} \in \mathbb{R}^{(k-1)\times(k-1)}$
- $F \in \mathbb{R}^{(m-k+1)\times(m-k+1)}$ is a Householder reflector

The I block ensures the first $k-1$ rows are unchanged

F is an orthogonal matrix that operates on the bottom $m-k+1$ rows

(Note that F orthogonal $\implies$ Q_k orthogonal)

Householder Reflectors

Let $x \in \mathbb{R}^{m-k+1}$ denote entries $k, \ldots, m$ of of the k^{th} column

We have two requirements for F:

1. F is orthogonal, so must have $\|Fx\|_2 = \|x\|_2$
2. Only the first entry of Fx should be non-zero

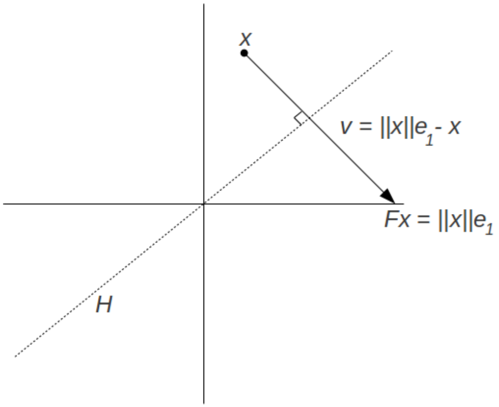
Hence we must have

$$x = \begin{bmatrix} \times \\ \times \\ \vdots \\ \times \end{bmatrix} \longrightarrow Fx = \begin{bmatrix} \|x\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \|x\|_2 e_1$$

Question: How can we achieve this?

Householder Reflectors

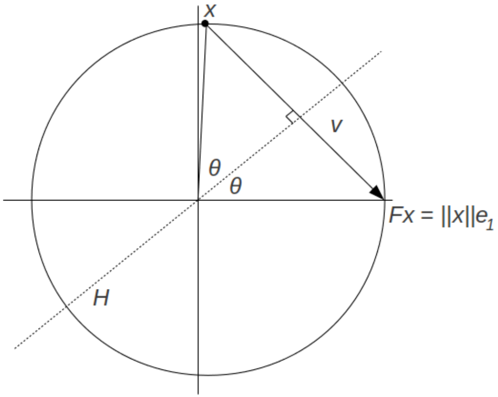
We can see geometrically that this can be achieved via reflection across a subspace H



Here H is the subspace orthogonal to $v \equiv \|x\|e_1 - x$, and the key point is that H "bisects" v

Householder Reflectors

We see that this bisection property is because x and Fx both live on the hypersphere centered at the origin with radius $\|x\|_2$



Householder Reflectors

Next, we need to determine the matrix F which maps x to $\|x\|_2 e_1$

F is closely related to the orthogonal projection of x onto H, since that projection takes us "half way" from x to $\|x\|_2 e_1$

Hence we first consider orthogonal projection onto H, and subsequently derive F

See Lecture: The orthogonal projection of a onto b is given by $\frac{(a \cdot b)}{\|b\|^2} b$

Therefore the matrix $\frac{vv^T}{v^T v}$ orthogonally projects x onto v

Householder Reflectors

Let $P_H \equiv \mathrm{I} - \frac{vv^T}{v^Tv}$

It follows that P_Hx is the orthogonal projection of x onto H, since it satisfies:

- $P_Hx \in H$
  $(v^TP_Hx = v^Tx - v^T\frac{vv^T}{v^Tv}x = v^Tx - \frac{v^Tv}{v^Tv}v^Tx = 0)$
- The projection error $x - P_Hx$ is orthogonal to H
  $(x - P_Hx = x - x + \frac{vv^T}{v^Tv}x = \frac{v^Tx}{v^Tv}v$, parallel to $v)$

Householder Reflectors

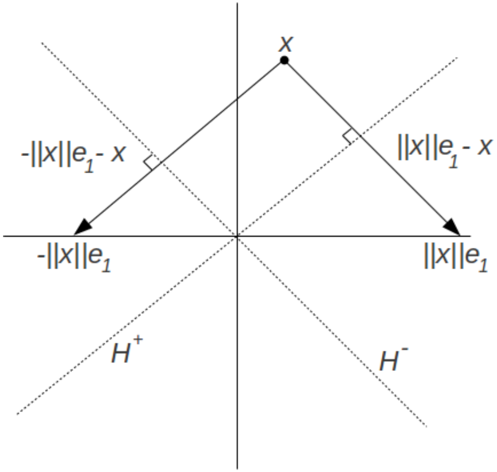
But recall that F should reflect across H rather than project onto H

Hence we obtain F by going "twice as far" in the direction of v compared to P_H, i.e. $F = \mathrm{I} - 2\frac{vv^T}{v^Tv}$

Exercise: Show that F is an orthogonal matrix, i.e. that $F^TF = \mathrm{I}$

Householder Reflectors

But in fact, we can see that there are two Householder reflectors that we can choose from[8]



[8]This picture is “not to scale”; H^- should bisect $-\|x\|e_1 - x$

Householder Reflectors

In practice, it's important to choose the "better of the two reflectors"

If x and $\|x\|_2 e_1$ (or x and $-\|x\|_2 e_1$) are close, we could obtain loss of precision due to cancellation (cf. Unit 0) when computing v

To ensure x and its reflection are well separated we should choose the reflection to be $-\operatorname{sign}(x_1)\|x\|_2 e_1$ (more details on next slide)

Therefore, we want v to be $v = -\operatorname{sign}(x_1)\|x\|_2 e_1 - x$; but since the sign of v does not affect F, we scale v by -1 to get

$$v = \operatorname{sign}(x_1)\|x\|_2 e_1 + x$$

Householder Reflectors

Let's compare the two options for v in the potentially problematic case when $x \approx \pm\|x\|_2 e_1$, *i.e.* when $|x_1| \approx \|x\|_2$:

$$v_{\text{bad}} \equiv -\operatorname{sign}(x_1)\|x\|_2 e_1 + x$$

$$v_{\text{good}} \equiv \operatorname{sign}(x_1)\|x\|_2 e_1 + x$$

$$\begin{aligned}
\|v_{\text{bad}}\|_2^2 &= \|-\operatorname{sign}(x_1)\|x\|_2 e_1 + x\|_2^2 \\
&= (-\operatorname{sign}(x_1)\|x\|_2 + x_1)^2 + \|x_{2:(m-k+1)}\|_2^2 \\
&= (-\operatorname{sign}(x_1)\|x\|_2 + \operatorname{sign}(x_1)|x_1|)^2 + \|x_{2:(m-k+1)}\|_2^2 \\
&\approx 0
\end{aligned}$$

$$\begin{aligned}
\|v_{\text{good}}\|_2^2 &= \|\operatorname{sign}(x_1)\|x\|_2 e_1 + x\|_2^2 \\
&= (\operatorname{sign}(x_1)\|x\|_2 + x_1)^2 + \|x_{2:(m-k+1)}\|_2^2 \\
&= (\operatorname{sign}(x_1)\|x\|_2 + \operatorname{sign}(x_1)|x_1|)^2 + \|x_{2:(m-k+1)}\|_2^2 \\
&\approx (2\operatorname{sign}(x_1)\|x\|_2)^2
\end{aligned}$$

Householder Reflectors

Recall that v is computed from two vectors of magnitude $\|x\|_2$

The argument above shows that with v_{bad} we can get $\|v\|_2 \ll \|x\|_2$
$\implies$ "loss of precision due to cancellation" is possible

In contrast, with v_{good} we always have $\|v_{\text{good}}\|_2 \geq \|x\|_2$, which rules out loss of precision due to cancellation

Householder Triangularization

We can now write out the Householder algorithm:

```
1: for $k = 1 : n$ do
2:     $x = a_{(k:m,k)}$
3:     $v_k = \text{sign}(x_1)\|x\|_2 e_1 + x$
4:     $v_k = v_k / \|v_k\|_2$
5:     $a_{(k:m,k:n)} = a_{(k:m,k:n)} - 2v_k(v_k^T a_{(k:m,k:n)})$
6: end for
```

This replaces A with R and stores $v_1, \ldots, v_n$

Note that we don't divide by $v_k^T v_k$ in line 5 since we normalize v_k in line 4

Householder algorithm requires $\sim 2mn^2 - \frac{2}{3}n^3$ operations[9]

[9]Compared to $2mn^2$ for Gram–Schmidt

Householder Triangularization

Note that we don't explicitly form Q

We can use the vectors $v_1, \ldots, v_n$ to compute Q in a post-processing step

Recall that

$$Q_k = \begin{bmatrix} \mathrm{I} & 0 \\ 0 & F \end{bmatrix}$$

and $Q \equiv (Q_n \cdots Q_2 Q_1)^T = Q_1^T Q_2^T \cdots Q_n^T$

Also, the Householder reflectors are symmetric (refer to the definition of F), so $Q = Q_1^T Q_2^T \cdots Q_n^T = Q_1 Q_2 \cdots Q_n$

Householder Triangularization

Hence, we can evaluate $Qx = Q_1 Q_2 \cdots Q_n x$ using the v_k:

```
1: for $k = n : -1 : 1$ do
2:     $x_{(k:m)} = x_{(k:m)} - 2v_k(v_k^T x_{(k:m)})$
3: end for
```

Question: How can we use this to form the matrix Q?

Householder Triangularization

Answer: Compute Q via Qe_i, $i = 1, \ldots, m$

Similarly, compute $\hat{Q}$ via Qe_i, $i = 1, \ldots, n$

However, often not necessary to form Q or $\hat{Q}$ explicitly, e.g. to solve $Ax \simeq b$ we only need the product $Q^T b$

Note the product $Q^T b = Q_n \cdots Q_2 Q_1 b$ can be evaluated as:

```
1: for $k = 1 : n$ do
2:     $b_{(k:m)} = b_{(k:m)} - 2v_k(v_k^T b_{(k:m)})$
3: end for
```

A Givens rotation

For $i < j$ and an angle θ, the elements of the $m \times m$ Givens rotation matrix $G(i, j, \theta)$ are

$$\begin{gathered} g_{ii} = c, \qquad g_{jj} = c, \qquad g_{ij} = s, \qquad g_{ji} = -s, \\ g_{kk} = 1 \quad \text{for } k \neq i, j, \\ g_{kl} = 0 \quad \text{otherwise}, \end{gathered} \tag{1}$$

where $c = \cos\theta$ and $s = \sin\theta$.

A Givens rotation

Hence the matrix has the form

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

It applies a rotation within the space spanned by the ith and jth coordinates

Effect of a Givens rotation

Consider a $m \times n$ rectangular matrix A where $m \geq n$

Suppose that a_1 and a_2 are in the ith and jth positions in a particular column of A. Assume at least one a_i is non-zero.

Restricting to just ith and jth dimensions, a Givens rotation $G(i, j, \theta)$ for a particular angle θ can be applied so that

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \end{pmatrix},$$

where α is non-zero, and the jth component is eliminated

Stable computation

α is given by $\sqrt{a_1^2 + a_2^2}$. We could compute

$$c = a_1/\sqrt{a_1^2 + a_2^2}, \qquad s = a_2/\sqrt{a_1^2 + a_2^2}$$

but this is susceptible to underflow/overflow if α is very small.

A better procedure is as follows:

- if $|a_1| > |a_2|$, set $t = \tan\theta = a_2/a_1$, and hence $c = \frac{1}{\sqrt{1+t^2}}, s = ct$.
- if $|a_2| \geq |a_1|$, set $\tau = \cot\theta = a_1/a_2$, and hence $s = \frac{1}{\sqrt{1+\tau^2}}, c = s\tau$.

Givens rotation algorithm

To perform the Givens procedure on a dense $m \times n$ rectangular matrix A where $m \geq n$, the following algorithm can be used:

```
1: $R = A, Q = I$
2: **for** $k = 1 : n$ **do**
3:     **for** $j = m : k + 1$ **do**
4:         Construct $G = G(j - 1, j, \theta)$ to eliminate $a_{jk}$
5:         $R = GR$
6:         $Q = QG^T$
7:     **end for**
8: **end for**
```

Givens rotation advantages

In general, for dense matrices, Givens rotations are not as efficient as the other two approaches (Gram–Schmidt and Householder)

However, they are advantageous for sparse matrices, since non-zero entries can be eliminated one-by-one. They are also amenable to parallelization. Consider the 6×6 matrix:

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

The numbers represent the steps at which a particular matrix entry can be eliminated. *e.g.* on step 3, elements $(4, 1)$ and $(6, 2)$ can be eliminated concurrently using $G(3, 4, \theta_a)$ and $G(5, 6, \theta_b)$, respectively, since these two matrices operate on different rows.

Singular Value Decomposition

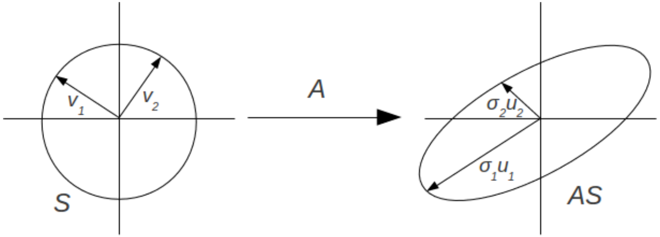
The Singular Value Decomposition (SVD) is a very useful matrix factorization

Motivation for SVD: image of the unit sphere, S, from any $m \times n$ matrix is a hyperellipse

A hyperellipse is obtained by stretching the unit sphere in $\mathbb{R}^m$ by factors $\sigma_1, \ldots, \sigma_m$ in orthogonal directions $u_1, \ldots, u_m$

Singular Value Decomposition

For $A \in \mathbb{R}^{2\times 2}$, we have



Singular Value Decomposition

Based on this picture, we make some definitions:

- Singular values: $\sigma_1, \sigma_2, \ldots, \sigma_n \geq 0$ (we typically assume $\sigma_1 \geq \sigma_2 \geq \ldots$)
- Left singular vectors: $\{u_1, u_2, \ldots, u_n\}$, unit vectors in directions of principal semiaxes of AS
- Right singular vectors: $\{v_1, v_2, \ldots, v_n\}$, preimages of the u_i so that $Av_i = \sigma_i u_i$, $i = 1, \ldots, n$

(The names "left" and "right" come from the formula for the SVD below)

Singular Value Decomposition

The key equation above is that

$$Av_i = \sigma_i u_i, \quad i = 1, \ldots, n$$

Writing this out in matrix form we get

$$\left[\begin{array}{c} \\ \\ A \\ \\ \\ \end{array} \right] \left[\begin{array}{c|c|c|c} & & & \\ v_1 & v_2 & \cdots & v_n \\ & & & \end{array} \right] = \left[\begin{array}{c|c|c|c} & & & \\ & & & \\ u_1 & u_2 & \cdots & u_n \\ & & & \\ & & & \end{array} \right] \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix}$$

Or more compactly:

$$AV = \widehat{U}\widehat{\Sigma}$$

Singular Value Decomposition

Here

- $\widehat{\Sigma} \in \mathbb{R}^{n \times n}$ is diagonal with non-negative, real entries
- $\widehat{U} \in \mathbb{R}^{m \times n}$ with orthonormal columns
- $V \in \mathbb{R}^{n \times n}$ with orthonormal columns

Therefore V is an orthogonal matrix ($V^T V = VV^T = \mathrm{I}$), so that we have the reduced SVD for $A \in \mathbb{R}^{m \times n}$:

$$A = \widehat{U}\widehat{\Sigma}V^T$$

Singular Value Decomposition

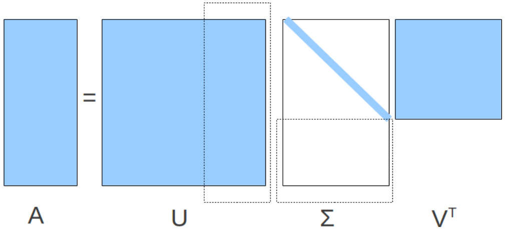
Just as with QR, we can pad the columns of $\widehat{U}$ with $m - n$ arbitrary orthogonal vectors to obtain $U \in \mathbb{R}^{m \times m}$

We then need to "silence" these arbitrary columns by adding rows of zeros to $\widehat{\Sigma}$ to obtain Σ

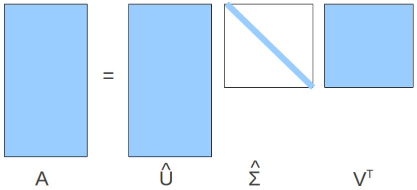
This gives the full SVD for $A \in \mathbb{R}^{m \times n}$:

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

Full vs Reduced SVD



Full SVD



Reduced SVD

Singular Value Decomposition

Theorem: Every matrix $A \in \mathbb{R}^{m \times n}$ has a full singular value decomposition. Furthermore:

- The σ_j are uniquely determined
- If A is square and the σ_j are distinct, the $\{u_j\}$ and $\{v_j\}$ are uniquely determined up to sign

Singular Value Decomposition

This theorem justifies the statement that the image of the unit sphere under any $m \times n$ matrix is a hyperellipse

Consider $A = U\Sigma V^T$ (full SVD) applied to the unit sphere, S, in $\mathbb{R}^n$:

1. The orthogonal map V^T preserves S
2. Σ stretches S into a hyperellipse aligned with the canonical axes e_j
3. U rotates or reflects the hyperellipse without changing its shape

SVD in Python

Python's `numpy.linalg.svd` function computes the full SVD of a matrix

```
Python 3.8.5 (default, Sep  6 2020, 03:54:05)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((4,2))
>>> (u,s,v)=np.linalg.svd(a)
>>> u
array([[-0.63528498, -0.57683859, -0.38592413, -0.3387223 ],
       [-0.39378781, -0.29780026,  0.4298513 ,  0.75595901],
       [-0.47505011,  0.36689119,  0.64669137, -0.47064692],
       [-0.46440451,  0.66630556, -0.49807699,  0.30378391]])
>>> s
array([1.90854638, 0.61500954])
>>> v
array([[-0.87128424, -0.49077874],
       [ 0.49077874, -0.87128424]])
```

SVD in Python

The `full_matrices=0` option computes the reduced SVD

```
Python 3.8.5 (default, Sep  6 2020, 03:54:05)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((4,2))
>>> (u,s,v)=np.linalg.svd(a,full_matrices=0)
>>> u
array([[-0.63528498, -0.57683859],
       [-0.39378781, -0.29780026],
       [-0.47505011,  0.36689119],
       [-0.46440451,  0.66630556]])
>>> s
array([1.90854638, 0.61500954])
>>> v
array([[-0.87128424, -0.49077874],
       [ 0.49077874, -0.87128424]])
```

Matrix Properties via the SVD

- The rank of A is r, the number of nonzero singular values[10]

Proof: In the full SVD $A = U\Sigma V^T$, U and V^T have full rank, hence it follows from linear algebra that $\text{rank}(A) = \text{rank}(\Sigma)$

- $\text{image}(A) = \text{span}\{u_1, \ldots, u_r\}$ and $\text{null}(A) = \text{span}\{v_{r+1}, \ldots, v_n\}$

Proof: This follows from $A = U\Sigma V^T$ and

$$\begin{aligned} \text{image}(\Sigma) &= \text{span}\{e_1, \ldots, e_r\} \in \mathbb{R}^m \\ \text{null}(\Sigma) &= \text{span}\{e_{r+1}, \ldots, e_n\} \in \mathbb{R}^n \end{aligned}$$

[10]This also gives us a good way to define rank in finite precision: the number of singular values larger than some (small) tolerance

Matrix Properties via the SVD

- $\|A\|_2 = \sigma_1$

Proof: Recall that $\|A\|_2 \equiv \max_{\|v\|_2=1} \|Av\|_2$. Geometrically, we see that $\|Av\|_2$ is maximized if $v = v_1$ and $Av = \sigma_1 u_1$.

- The singular values of A are the square roots of the eigenvalues of $A^T A$ or AA^T

Proof: (Analogous for AA^T)

$$A^T A = (U\Sigma V^T)^T (U\Sigma V^T) = V\Sigma U^T U\Sigma V^T = V(\Sigma^T\Sigma)V^T,$$

hence $(A^T A)V = V(\Sigma^T\Sigma)$, or $(A^T A)v_{(:,j)} = \sigma_j^2 v_{(:,j)}$

Matrix Properties via the SVD

The pseudoinverse, A^+, can be defined more generally in terms of the SVD

Define pseudoinverse of a scalar σ to be $1/\sigma$ if $\sigma \neq 0$ and zero otherwise

Define pseudoinverse of a (possibly rectangular) diagonal matrix as transpose of the matrix and taking pseudoinverse of each entry

Pseudoinverse of $A \in \mathbb{R}^{m \times n}$ is defined as

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

A^+ exists for any matrix A, and it captures our definitions of pseudoinverse from previously

Matrix Properties via the SVD

We generalize the condition number to rectangular matrices via the definition $\kappa(A) = \|A\|\|A^+\|$

We can use the SVD to compute the 2-norm condition number:

- $\|A\|_2 = \sigma_{\text{max}}$
- Largest singular value of A^+ is $1/\sigma_{\text{min}}$ so that $\|A^+\|_2 = 1/\sigma_{\text{min}}$

Hence $\kappa(A) = \sigma_{\text{max}}/\sigma_{\text{min}}$

Matrix Properties via the SVD

These results indicate the importance of the SVD, both theoretically and as a computational tool

Algorithms for calculating the SVD are an important topic in Numerical Linear Algebra, but outside scope of this course

Requires $\sim 4mn^2 - \frac{4}{3}n^3$ operations

For more details on algorithms, see Trefethen & Bau, or Golub & van Loan

Low-Rank Approximation via the SVD

One of the most useful properties of the SVD is that it allows us to obtain an optimal low-rank approximation to A

We can recast SVD as

$$A = \sum_{j=1}^{r} \sigma_j u_j v_j^T$$

Follows from writing Σ as sum of r matrices, Σ_j, where $\Sigma_j \equiv \text{diag}(0, \ldots, 0, \sigma_j, 0, \ldots, 0)$

Each $u_j v_j^T$ is a rank one matrix: each column is a scaled version of u_j

Low-Rank Approximation via the SVD

Theorem: For any $0 \leq \nu \leq r$, let $A_\nu \equiv \sum_{j=1}^{\nu} \sigma_j u_j v_j^T$, then

$$\|A - A_\nu\|_2 = \inf_{B \in \mathbb{R}^{m \times n},\ \text{rank}(B) \leq \nu} \|A - B\|_2 = \sigma_{\nu+1}$$

That is:

- A_ν gives us the closest rank ν matrix to A, measured in the 2-norm
- The error in A_ν is given by the first *omitted* singular value

Low-Rank Approximation via the SVD

A similar result holds in the Frobenius norm:

$$\|A - A_\nu\|_F = \inf_{B \in \mathbb{R}^{m \times n},\ \mathrm{rank}(B) \leq \nu} \|A - B\|_F = \sqrt{\sigma_{\nu+1}^2 + \cdots + \sigma_r^2}$$

Low-Rank Approximation via the SVD

These theorems indicate that the SVD is an effective way to *compress* data encapsulated by a matrix!

If singular values of A decay rapidly, can approximate A with few rank one matrices (only need to store σ_j, u_j, v_j for $j = 1, \ldots, \nu$)

Example: Image compression via the SVD