Numerical Analysis
Solving Linear Systems

Aleksandar Donev
Courant Institute, NYU[1]
donev@courant.nyu.edu

[1]Course MATH-UA.0252/MA-UY_4424, Spring 2021

Spring 2021

Outline

Outline

Matrices and linear systems

- It is said that 70% or more of applied mathematics research involves solving systems of m linear equations for n unknowns:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, \cdots, m.$$

- Linear systems arise directly from **discrete models**, e.g., traffic flow in a city. Or, they may come through representing or more abstract **linear operators** in some finite basis (representation).
  Common abstraction:

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

- Special case: Square invertible matrices, $m = n$, $\det \mathbf{A} \neq 0$:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.$$

- The goal: Calculate solution $\mathbf{x}$ given data $\mathbf{A}, \mathbf{b}$ in the most numerically stable and also efficient way.

Outline

GEM: Eliminating x_1

Step 1: $$Ax = b$$

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix} \begin{array}{l} \leftarrow \text{Multiply FIRST row by} \\ \leftarrow l_{21} = \dfrac{a_{21}^{(1)}}{a_{11}^{(1)}} \\ \leftarrow l_{31} = \dfrac{a_{31}^{(1)}}{a_{11}^{(1)}} \end{array}$$

$$\Big\Downarrow \text{ Eliminate } x_1$$

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 = a_{21}^{(1)} - l_{21} \cdot a_{11} & a_{22}^{(1)} - l_{21} \cdot a_{12}^{(1)} & a_{23}^{(1)} - l_{21} \cdot a_{13}^{(1)} \\ 0 & a_{32}^{(1)} - l_{31} \cdot a_{12}^{(1)} & a_{33}^{(1)} - l_{31} \cdot a_{13}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 - l_{21} \cdot b_1 \\ b_3 - l_{31} \cdot b_1 \end{bmatrix}$$

1

GEM: Eliminating x_2

Step 2:

$$\left[\begin{array}{c|cc} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ \hline 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} \end{array}\right] \left[\begin{array}{c} x_1 \\ \hline x_2 \\ \hline x_3 \end{array}\right] = \left[\begin{array}{c} b_1^{(2)} \\ \hline b_2^{(2)} \\ \hline b_3^{(3)} \end{array}\right] \begin{array}{l} \text{done row!} \\ \hline \leftarrow \text{Multiply second row by} \\ \leftarrow l_{32} = \dfrac{a_{32}^{(2)}}{a_{22}^{(2)}} \end{array}$$

$\Downarrow$ Eliminate x_2

$$\left[\begin{array}{ccc} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(3)} \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array}\right] = \left[\begin{array}{c} b_1^{(3)} \\ b_2^{(3)} \\ b_3^{(3)} \end{array}\right] \begin{array}{l} \text{Upper triangular system} \\ \\ \leftarrow \text{Solve } x_3 = \dfrac{b_3^{(3)}}{a_{33}^{(3)}} \end{array}$$

GEM: Backward substitution

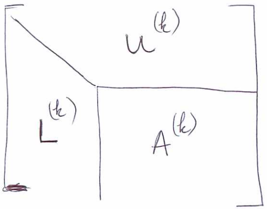
Eliminate x_3 entirely $\longrightarrow$

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} \\ 0 & a_{22}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1^{(3)} - a_{13}^{(1)} x_3 \\ b_2^{(3)} - a_{23}^{(2)} x_3 \end{bmatrix} = \tilde{b}$$

solve for $x_2 = \dfrac{\tilde{b}}{a_{22}^{(2)}}$, then x_1, and done!

IDEA: Store the multipliers in the lower triangle of A:

Matrix at Step k:

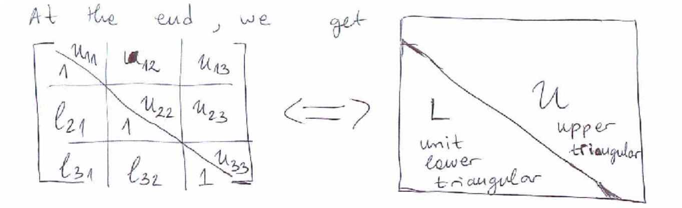


$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & a_{22}^{(2)} & a_{23}^{(2)} \\ l_{31} & a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix}$$

Example step 2

3

GEM as an *LU* factorization tool


At the end, we get
u_{11} u_{12} u_{13}
1
l_{21} u_{22} u_{23}
1
l_{31} l_{32} 1 u_{33}
$\Longleftrightarrow$
L
unit lower triangular
U
upper triangular

- We have actually **factorized A** as

$$\mathbf{A} = \mathbf{LU},$$

  L is **unit lower triangular** ($l_{ii} = 1$ on diagonal), and **U** is **upper triangular**.
- GEM is thus essentially the same as the *LU* **factorization method**.

GEM in MATLAB

```
% Sample MATLAB code (for learning purposes only, not
function A = MyLU(A)
% LU factorization in-place (overwrite A)
[n,m]=size(A);
if (n ~= m); error('Matrix not square'); end
for k=1:(n-1) % For variable x(k)
   % Calculate multipliers in column k:
   A((k+1):n,k) = A((k+1):n,k) / A(k,k);
   % Note: Pivot element A(k,k) assumed nonzero!
   for j=(k+1):n
      % Eliminate variable x(k):
      A((k+1):n,j) = A((k+1):n,j) - ...
         A((k+1):n,k) * A(k,j);
   end
end
end
```

Pivoting

Zero diagonal entries (pivots) pose a problem $\longrightarrow$ PIVOTING (swapping rows and columns)

$$Ax = b$$

$$\begin{bmatrix} 1 & 1 & 3 \\ 2 & 2 & 2 \\ 3 & 6 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \\ 13 \end{bmatrix} \Rightarrow \left[\begin{array}{c|cc} 1 & 1 & 3 \\ \hline 2 & \mathbf{0} & -4 \\ 3 & 3 & -5 \end{array}\right] \circlearrowright$$

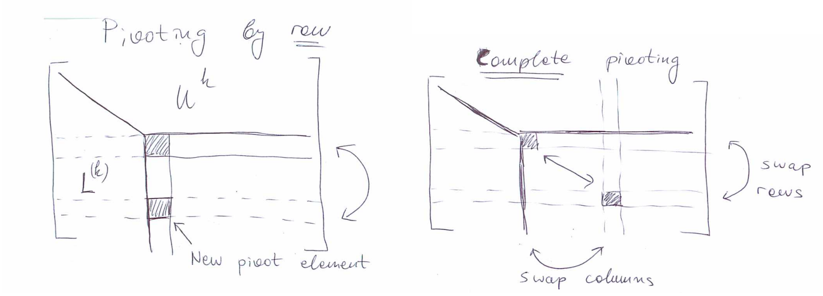
$$\left[\begin{array}{c|cc} 1 & 1 & 3 \\ \hline 3 & \textcircled{3} & -5 \\ 2 & 0 & -4 \end{array}\right] \Rightarrow \left[\begin{array}{c|c|c} {}_{1}\backslash{}^{1} & 1 & 3 \\ \hline 3 & {}_{1}\backslash{}^{3} & -5 \\ \hline 2 & 0 & {}_{1}\backslash{}^{-4} \end{array}\right]$$

OBSERVE

$$\underline{\underline{LU = \overset{\text{PERMUTED}}{A}}}$$

$\textcircled{4}$

Pivoting during **LU** factorization



- **Partial (row) pivoting** permutes the rows (equations) of **A** in order to ensure sufficiently large pivots and thus numerical stability:

$$\mathbf{PA} = \mathbf{LU}$$

- Here **P** is a **permutation matrix**, meaning a matrix obtained by permuting rows and/or columns of the identity matrix.
- **Complete pivoting** also permutes columns, $\mathbf{PAQ} = \mathbf{LU}$.

Gauss Elimination Method (GEM)

- GEM is a **general** method for **dense matrices** and is commonly used.
- Implementing GEM efficiently and stably is difficult and we will not discuss it here, since others have done it for you!
- The **LAPACK** public-domain library is the main repository for excellent implementations of dense linear solvers.
- MATLAB uses a highly-optimized variant of GEM by default, mostly based on LAPACK.
- MATLAB does have **specialized solvers** for special cases of matrices, so always look at the help pages!

Solving linear systems

- Once an LU factorization is available, solving a linear system is simple:

  $$\mathbf{A}\mathbf{x} = \mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{L}\left(\mathbf{U}\mathbf{x}\right) = \mathbf{L}\mathbf{y} = \mathbf{b}$$

  so solve for $\mathbf{y}$ using **forward substitution**.
  This was implicitly done in the example above by overwriting $\mathbf{b}$ to become $\mathbf{y}$ during the factorization.
- Then, solve for $\mathbf{x}$ using **backward substitution**

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

- If row pivoting is necessary, the same applies but $\mathbf{L}$ or $\mathbf{U}$ may be permuted upper/lower triangular matrices,

  $$\mathbf{A} = \widetilde{\mathbf{L}}\mathbf{U} = \left(\mathbf{P}^T\mathbf{L}\right)\mathbf{U}.$$

In MATLAB

- In MATLAB, the **backslash operator** (see help on *mldivide*)

$$x = A \backslash b \approx A^{-1} b,$$

  solves the linear system $\mathbf{Ax} = \mathbf{b}$ using the LAPACK library.
  Never use matrix inverse to do this, even if written as such on paper.
- Doing $x = A \backslash b$ is **equivalent** to performing an LU factorization and doing two **triangular solves** (backward and forward substitution):

$$\begin{aligned}[\tilde{L}, U] &= lu(A) \\ y &= \tilde{L} \backslash b \\ x &= U \backslash y\end{aligned}$$

- This is a carefully implemented **backward stable** pivoted LU factorization, meaning that the returned solution is as accurate as the conditioning number allows.

GEM Matlab example (1)

```
>> A = [ 1     2     3 ;  4     5     6 ;  7     8     0 ];
>> b=[2  1  -1]';

>> x=A^(-1)*b; x' % Don't do this!
ans =      -2.5556     2.1111     0.1111

>> x = A\b; x' % Do this instead
ans =     -2.5556     2.1111     0.1111

>> linsolve(A,b)' % Even more control
ans =      -2.5556     2.1111     0.1111
```

GEM Matlab example (2)

```
>> [L,U] = lu(A) % Even better if resolving

L =     0.1429    1.0000         0
        0.5714    0.5000    1.0000
        1.0000         0         0
U =     7.0000    8.0000         0
             0    0.8571    3.0000
             0         0    4.5000

>> norm(L*U-A,inf)
ans =      0

>> y = L\b;
>> x = U\y; x'
ans =   -2.5556    2.1111    0.1111
```

Backwards Stability

- Even though we cannot get $\mathbf{x}$ correctly for ill-conditioned linear systems, we can still get an (not *the* one!) $\mathbf{x}$ that is a solution of the equation to almost machine precision.
- This sort of **backward stability** means that there is a problem nearby the original problem such that the answer we compute $\hat{\mathbf{x}}$ is the solution of that "perturbed" problem,

$$(\mathbf{A} + \delta\mathbf{A})\,\hat{\mathbf{x}} = \mathbf{b} + \delta\mathbf{b}.$$

- A backwards stable method gives a **residual** $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$ that is zero to within the rounding unit $u \approx 10^{-16}$,

$$\frac{\|\mathbf{Ax} - \mathbf{b}\|}{\|\mathbf{b}\|} \sim \frac{\|\mathbf{Ax} - \mathbf{b}\|}{\|\mathbf{Ax}\|} \sim u,$$

- Observe that the conditioning number of the matrix does not enter here, it can be large!

Backwards Stability contd.

- Gaussian elimination with partial pivoting is almost always backwards stable in practice, but one can always check the residual after computing the answer (**always good practice** to confirm you solved the problem you thought you solved!)
- Specifically, if we compute the LU factorization we are guaranteed that

$$\mathbf{A} + \delta\mathbf{A} = \mathbf{LU} \quad \text{where} \quad \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \leq Cu$$

  where C is some modest constant that depends *polynomially* on the number of unknowns (not exponentially).
- Complete pivoting is rarely used in practice because it is expensive, even though it will give better guarantees.

Cost estimates for GEM

- For forward or backward substitution, at step k there are $\sim (n-k)$ multiplications and subtractions, plus a few divisions.
  The total over all n steps is

$$\sum_{k=1}^{n}(n-k) = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

  subtractions and multiplications, giving a total of $O(n^2)$ **floating-point operations** (FLOPs).
- The LU factorization itself costs a lot more, $O(n^3)$,

$$\text{FLOPS} \approx \frac{2n^3}{3},$$

  and the triangular solves are negligible for large systems.
- When many linear systems need to be solved with the same $\mathbf{A}$ the **factorization can be reused**.

Outline

Stability analysis

Perturbations on **right hand side** (rhs) only:

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

$$\delta\mathbf{x} = \mathbf{A}^{-1}\delta\mathbf{b} \quad \Rightarrow \|\delta\mathbf{x}\| \leq \left\|\mathbf{A}^{-1}\right\| \|\delta\mathbf{b}\|$$

Using the bounds

$$\|\mathbf{b}\| \leq \|\mathbf{A}\| \|\mathbf{x}\| \quad \Rightarrow \|\mathbf{x}\| \geq \|\mathbf{b}\| / \|\mathbf{A}\|$$

the relative error in the solution can be bounded by

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\left\|\mathbf{A}^{-1}\right\| \|\delta\mathbf{b}\|}{\|\mathbf{x}\|} \leq \frac{\left\|\mathbf{A}^{-1}\right\| \|\delta\mathbf{b}\|}{\|\mathbf{b}\| / \|\mathbf{A}\|} = \kappa(\mathbf{A})\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

where the **conditioning number** $\kappa(\mathbf{A})$ depends on the matrix norm used:

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \left\|\mathbf{A}^{-1}\right\| \geq 1.$$

Conditioning Number

- The full derivation, not given here, estimates the uncertainty or perturbation in the solution:

$$\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\kappa(\mathbf{A})}{1-\kappa(\mathbf{A})\frac{\|\delta \mathbf{A}\|}{\|\mathbf{A}\|}}\left(\frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|}+\frac{\|\delta \mathbf{A}\|}{\|\mathbf{A}\|}\right).$$

  The **worst-case conditioning** of the linear system is determined by $\kappa(\mathbf{A})$.
- Best possible error with rounding unit $u \approx 10^{-16}$:

$$\frac{\|\delta \mathbf{x}\|_\infty}{\|\mathbf{x}\|_\infty} \lesssim 2u\kappa(\mathbf{A}),$$

- Solving an ill-conditioned system, $\kappa(\mathbf{A}) \gg 1$ (e.g., $\kappa = 10^{15}$!) , should only be done if something special is known.
- The conditioning number can only be **estimated** in practice since $\mathbf{A}^{-1}$ is not available (see MATLAB's *rcond* function).

Matrix Rescaling and Reordering

- Pivoting is not always sufficient to ensure lack of roundoff problems. In particular, **large variations** among the entries in **A should be avoided**.
- This can usually be remedied by changing the physical units for $\mathbf{x}$ and $\mathbf{b}$ to be the **natural units** $\mathbf{x}_0$ and $\mathbf{b}_0$.
- **Rescaling** the unknowns and the equations is generally a good idea even if not necessary:

$$\mathbf{x} = \mathbf{D}_x \tilde{\mathbf{x}} = \text{Diag}\left\{\mathbf{x}_0\right\} \tilde{\mathbf{x}} \text{ and } \mathbf{b} = \mathbf{D}_b \tilde{\mathbf{b}} = \text{Diag}\left\{\mathbf{b}_0\right\} \tilde{\mathbf{b}}.$$

$$\mathbf{A}\mathbf{x} = \mathbf{A}\mathbf{D}_x \tilde{\mathbf{x}} = \mathbf{D}_b \tilde{\mathbf{b}} \quad \Rightarrow \quad \left(\mathbf{D}_b^{-1}\mathbf{A}\mathbf{D}_x\right) \tilde{\mathbf{x}} = \tilde{\mathbf{b}}$$

- The **rescaled matrix** $\widetilde{\mathbf{A}} = \mathbf{D}_b^{-1}\mathbf{A}\mathbf{D}_x$ should have a better conditioning.
- Also note that **reordering the variables** from most important to least important may also help.

Outline

Positive-Definite Matrices

- A real symmetric matrix $\mathbf{A}$ is positive definite iff (if and only if):
  1. All of its eigenvalues are real (follows from symmetry) and positive.
  2. $\forall x \neq \mathbf{0}$, $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$, i.e., the quadratic form defined by the matrix $\mathbf{A}$ is convex.
  3. There exists a *unique* lower triangular $\mathbf{L}$, $L_{ii} > 0$,

     $$\mathbf{A} = \mathbf{L}\mathbf{L}^T,$$

     termed the **Cholesky factorization** of $\mathbf{A}$ (symmetric LU factorization).

1. For Hermitian complex matrices just replace transposes with adjoints (conjugate transpose), e.g., $\mathbf{A}^T \rightarrow \mathbf{A}^\star$ (or $\mathbf{A}^H$ in the book).

Cholesky Factorization

- The MATLAB built in function

$$R = chol(A)$$

  gives the Cholesky factorization and is a good way to **test for positive-definiteness**.
- The cost of a Cholesky factorization is about half the cost of LU factorization, $n^3/3$ FLOPS.
- Solving linear systems is as for LU factorization, replacing $\mathbf{U}$ with $\mathbf{L}^T$.
- For Hermitian/symmetric matrices with positive diagonals MATLAB tries a Cholesky factorization first, *before* resorting to LU factorization with pivoting.

Special Matrices in MATLAB

- MATLAB recognizes (i.e., tests for) some special matrices automatically: banded, permuted lower/upper triangular, symmetric, Hessenberg, but **not** sparse.
- In MATLAB one may specify a matrix $\mathbf{B}$ instead of a single right-hand side vector $\mathbf{b}$.
- The MATLAB function

$$X = \mathit{linsolve}(A, B, \mathit{opts})$$

  allows one to specify certain properties that speed up the solution (triangular, upper Hessenberg, symmetric, positive definite, none), and also estimates the condition number along the way.
- Use *linsolve* instead of backslash if you know (for sure!) something about your matrix.

Outline

Non-Square Matrices

- In the case of **over-determined** (more equations than unknowns) or **under-determined** (more unknowns than equations), the solution to linear systems in general becomes **non-unique**.
- One must first define what is meant by a solution, and the common definition is to use a **least-squares formulation**:

$$\mathbf{x}^\star = \arg\min_{\mathbf{x}\in\mathbb{R}^n} \|\mathbf{Ax}-\mathbf{b}\| = \arg\min_{\mathbf{x}\in\mathbb{R}^n} \Phi(\mathbf{x})$$

  where the choice of the L_2 norm leads to:

$$\Phi(\mathbf{x}) = (\mathbf{Ax}-\mathbf{b})^T(\mathbf{Ax}-\mathbf{b}).$$

- Over-determined systems, $m > n$, can be thought of as **fitting a linear model (linear regression)**:
  The unknowns $\mathbf{x}$ are the coefficients in the fit, the input data is in $\mathbf{A}$ (one column per measurement), and the output data (observables) are in $\mathbf{b}$.

Normal Equations

- It can be shown that the least-squares solution satisfies:

$$\nabla \Phi(\mathbf{x}) = \mathbf{A}^T \left[2\left(\mathbf{A}\mathbf{x} - \mathbf{b}\right)\right] = \mathbf{0} \text{ (critical point)}$$

- This gives the square linear system of **normal equations**

$$\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x}^\star = \mathbf{A}^T\mathbf{b}.$$

- If $\mathbf{A}$ is of full rank, $\text{rank}\left(\mathbf{A}\right) = n$, it can be shown that $\mathbf{A}^T\mathbf{A}$ is positive definite, and Cholesky factorization can be used to solve the normal equations.
- Multiplying $\mathbf{A}^T$ $(n \times m)$ and $\mathbf{A}$ $(m \times n)$ takes n^2 dot-products of length m, so $O(mn^2)$ operations

Problems with the normal equations

$$\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x}^\star = \mathbf{A}^T\mathbf{b}.$$

- The conditioning number of the normal equations is

$$\kappa\left(\mathbf{A}^T\mathbf{A}\right) = \left[\kappa(\mathbf{A})\right]^2$$

- Furthermore, roundoff can cause $\mathbf{A}^T\mathbf{A}$ to no longer appear as positive-definite and the Cholesky factorization will fail.
- If the normal equations are ill-conditioned, another approach is needed.

The QR factorization

- For nonsquare or ill-conditioned matrices of **full-rank** $r = n \leq m$, the LU factorization can be replaced by the QR factorization:

$$\mathbf{A} = \mathbf{QR}$$
$$[m \times n] = [m \times n][n \times n]$$

where **Q** has **orthogonal columns**, $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}_n$, and **R** is a **non-singular upper triangular** matrix.

- Observe that orthogonal / unitary matrices are **well-conditioned** ($\kappa_2 = 1$), so the QR factorization is numerically better (but also more expensive!) than the LU factorization.
- For matrices **not of full rank** there are modified QR factorizations but **the SVD decomposition is better** (next class).
- In MATLAB, the QR factorization can be computed using *qr* (with column pivoting).

Solving Linear Systems via *QR* factorization

$$\left(\mathbf{A}^T\mathbf{A}\right)\mathbf{x}^\star = \mathbf{A}^T\mathbf{b} \text{ where } \mathbf{A} = \mathbf{QR}$$

- Observe that **R** is the Cholesky factor of the matrix in the normal equations:

$$\mathbf{A}^T\mathbf{A} = \mathbf{R}^T\left(\mathbf{Q}^T\mathbf{Q}\right)\mathbf{R} = \mathbf{R}^T\mathbf{R}$$

$$\left(\mathbf{R}^T\mathbf{R}\right)\mathbf{x}^\star = \left(\mathbf{R}^T\mathbf{Q}^T\right)\mathbf{b} \quad \Rightarrow \quad \mathbf{x}^\star = \mathbf{R}^{-1}\left(\mathbf{Q}^T\mathbf{b}\right)$$

  which amounts to solving a triangular system with matrix **R**.

- This calculation turns out to be much **more numerically stable** against roundoff than forming the normal equations (and has similar cost).

Computing the QR Factorization

- The QR factorization is closely-related to the **orthogonalization** of a set of n vectors (columns) $\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n\}$ in $\mathbb{R}^m$, which is a common problem in numerical computing.
- Classical approach is the **Gram-Schmidt method**: To make a vector $\mathbf{b}$ orthogonal to $\mathbf{a}$ do:

$$\tilde{\mathbf{b}} = \mathbf{b} - (\mathbf{b} \cdot \mathbf{a}) \frac{\mathbf{a}}{(\mathbf{a} \cdot \mathbf{a})}$$

- Repeat this in sequence: Start with $\tilde{\mathbf{a}}_1 = \mathbf{a}_1$, then make $\tilde{\mathbf{a}}_2$ orthogonal to $\tilde{\mathbf{a}}_1 = \mathbf{a}_1$, then make $\tilde{\mathbf{a}}_3$ orthogonal to $\operatorname{span}(\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2) = \operatorname{span}(\mathbf{a}_1, \mathbf{a}_2)$:

$$\begin{aligned}
\tilde{\mathbf{a}}_1 &= \mathbf{a}_1 \\
\tilde{\mathbf{a}}_2 &= \mathbf{a}_2 - (\mathbf{a}_2 \cdot \mathbf{a}_1) \frac{\mathbf{a}_1}{(\mathbf{a}_1 \cdot \mathbf{a}_1)} \\
\tilde{\mathbf{a}}_3 &= \mathbf{a}_3 - (\mathbf{a}_3 \cdot \mathbf{a}_1) \frac{\mathbf{a}_1}{(\mathbf{a}_1 \cdot \mathbf{a}_1)} - (\mathbf{a}_3 \cdot \mathbf{a}_2) \frac{\mathbf{a}_2}{(\mathbf{a}_2 \cdot \mathbf{a}_2)}
\end{aligned}$$

Gram-Schmidt Orthogonalization

- More efficient formula (**standard Gram-Schmidt**):

$$\tilde{\mathbf{a}}_{k+1} = \mathbf{a}_{k+1} - \sum_{j=1}^{k} \left(\mathbf{a}_{k+1} \cdot \mathbf{q}_j\right) \mathbf{q}_j, \quad \mathbf{q}_{k+1} = \frac{\tilde{\mathbf{a}}_{k+1}}{\left\|\tilde{\mathbf{a}}_{k+1}\right\|},$$

  with cost $\approx 2mn^2$ FLOPS but is **not numerically stable** against roundoff errors (**loss of orthogonality**).
- In the standard method we make each vector orthogonal to all previous vectors. A **numerically stable** alternative is the **modified Gram-Schmidt**, in which we take each vector and modify all following vectors (not previous ones) to be orthogonal to it (so the sum above becomes $\sum_{j=k+1}^{m}$).
- As we saw in previous lecture, a small rearrangement of mathematically-equivalent approaches can produce a much more robust numerical method.

Outline

Conclusions/Summary

- The conditioning of a linear system $\mathbf{Ax} = \mathbf{b}$ is determined by the condition number
$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \left\|\mathbf{A}^{-1}\right\| \geq 1$$
- Gauss elimination can be used to solve general square linear systems and also produces a factorization $\mathbf{A} = \mathbf{LU}$.
- Partial pivoting is often necessary to ensure numerical stability during GEM and leads to $\mathbf{PA} = \mathbf{LU}$ or $\mathbf{A} = \widetilde{\mathbf{L}}\mathbf{U}$.
- For symmetric positive definite matrices the Cholesky factorization $\mathbf{A} = \mathbf{LL}^T$ is preferred and does not require pivoting.
- The *QR* factorization is a numerically-stable method for solving **full-rank non-square systems**.