CS 475
Lecture Notes
Spring 2025

Collin Roberts

June 30, 2025

Contents

1 Lecture 01: Linear Algebra Review

Outline

1. Course Mechanics
2. Basic Theory of Linear Algebra

1.1 Course Mechanics

- See:
  - the unsecured course website: https://student.cs.uwaterloo.ca/~cs475, and
  - the course outline which will be linked there.

1.2 Basic Theory of Linear Algebra

Definition 1.2.1. *Let A be a matrix. The* ***range*** *of A defined as*

$$range(A) = \{y \mid Ax = y \text{ for some vector } x\}.$$

Theorem 1.2.2.

$$\begin{aligned} range(A) &= \text{the column space of } A \\ &= \text{the space spanned by the column vectors of } A = [a_1 \ \cdots \ a_n] \\ &= \{y = x_1 a_1 + \cdots + x_n a_n\} \\ &= \left\{ \sum_{j=1}^{n} x_j a_j \text{ for scalars } x_j \right\}. \end{aligned}$$

Q: What is the difference between $range(A)$ and the column space of A?
A: There is no difference. The $\vec{x}$ in the definition of $range(A)$ is precisely

$$\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

from the definition of the column space.

Remark: We are working over the field $\mathbb{R}$. Otherwise our computational approach would not make sense.

Definition 1.2.3. *1. column rank = dimension of the column space*
2. row rank = dimension of the row space

Theorem 1.2.4. *column rank = row rank.*

Thus we may simply refer to the rank of A.

Definition 1.2.5. *An $m \times n$ matrix A is of **full rank** if*

$$rank(A) = \min(m, n).$$

Thus, if $m \geq n$, then A is of full rank if it has n linearly independent column vectors.

Definition 1.2.6. *A set $S = \{\vec{v_1}, \ldots, \vec{v_n}\}$ of vectors is **linearly independent** of and only if, for any scalars $c_1, \ldots, c_n$, $c_1\vec{v_1} + \cdots + c_n\vec{v_n} = \vec{0}$ implies $c_1 = \cdots = c_n = 0$, i.e. the only linear combination of $\vec{v_1}, \ldots, \vec{v_n}$ which equals $\vec{0}$ is the trivial one.*

Definition 1.2.7. *A **nonsingular (invertible) matrix** is a square matrix, of full rank.*

Definition 1.2.8. *The **null space** of A is*

$$null(A) = \{x \mid Ax = \vec{0}\}.$$

Matrix Inverses

$$\begin{aligned} (AB)^{-1} &= B^{-1}A^{-1} \\ (A^{-1})^T &= (A^T)^{-1} \\ &\stackrel{def}{=} A^{-T}. \end{aligned}$$

Theorem 1.2.9. $B^{-1} = A^{-1} - B^{-1}(B - A)A^{-1}$.

Proof.

$$\begin{aligned} & B\left[A^{-1} - B^{-1}(B - A)A^{-1}\right] \\ & BA^{-1} - BB^{-1}(B - A)A^{-1} \\ & \underbrace{BA^{-1} - BA^{-1}}_{=0} + \underbrace{AA^{-1}}_{=I} \\ & I. \end{aligned}$$

It is an exercise for you to check multiplication on the other side. □

Remarks:

1. This Theorem is useful to establish the Sherman-Morrison-Woodbury formula, below.

$$(A + \underbrace{UV^T}_{\text{rank } k})^{-1} = A^{-1} - \underbrace{A^{-1}U(I + V^TA^{-1}U)^{-1}V^TA^{-1}}_{\text{rank } k},$$

where A is an invertible $n \times n$ matrix, and U, V are $n \times k$ matrices (usually $k \leq n$).

Thus a rank k correction to A (LHS) results in a rank k correction to the inverse (RHS):

- Assume UV^T (which is $(n \times n)$), has rank k.
- Further assume that $(I + V^TA^{-1}U)$ (which is $(k \times k)$) is invertible.
- Then $U(I + V^TA^{-1}U)^{-1}V^T$ also has rank k.

- A and A^{-1} have full rank, therefore $A^{-1}U(I + V^TA^{-1}U)^{-1}V^TA^{-1}$ also has rank k.

Example: Let A be $n \times n$, say $A = [a_{ij}]$. Let $k = 1$.

$$
\begin{aligned}
u &= \begin{bmatrix} u_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, v = \begin{bmatrix} v_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \\
uv^T &= \begin{bmatrix} u_1v_1 & 0 & \cdots & 0 \\ 0 & & & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \\
A + uv^T &= \begin{cases} a_{11} + u_1v_1 & \text{if } i = 1 \text{ and } j = 1 \\ a_{ij} & \text{otherwise} \end{cases} \\
(A + uv^T)^{-1} &= A^{-1} - \underbrace{A^{-1}u}_{n\times 1}(1 + \underbrace{v^TA^{-1}u}_{1\times 1})^{-1}\underbrace{v^TA^{-1}}_{1\times n}
\end{aligned}
$$

1.3 Q & A

1. Will we need to know MATLAB for this course?
   A: Yes.
2. How should we learn MATLAB?
   A: I will run a MATLAB tutorial, well before the first Crowdmark assignment is released. I will also post a "Quick Reference" guide.
3. Will the instructor post the URL for the unsecured website on LEARN?
   A: Yes! This is already done.
4. Can you talk about the optional textbooks?
   A: Yes. See the details posted on the course outline.
5. Can you be specific about which textbook to read for each topic?
   A: Yes, absolutely.
6. Will the mid-term be 100% written, or will there also be a MATLAB component?
   A: 100% written.
7. What types of questions will be on the marked quizzes on LEARN?
   A: The auto-marked types, e.g. MC, MS, MAT, TF, etc.
8. Will the Marked Quizzes and Crowdmark assignments be open book?
   A: Yes!

2 Lecture 02: Solving Linear Systems

Outline

1. Solving Linear Systems
   (a) LU factorizations
      i. Complexity
   (b) Symmetric Systems
   (c) Positive Definite Systems

2.1 Solving Linear Systems

How To Compute $x = A^{-1}b$: In numerical linear algebra, we never compute A^{-1} in order to compute $A^{-1}b$. Instead we compute x as the solution of $Ax = b$, via Gaussian elimination.

Big Picture of Gaussian Elimination

$$\underbrace{\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}}_{A} \rightarrow \underbrace{\begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix}}_{A^{(1)}} \rightarrow \underbrace{\begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & x & x \end{bmatrix}}_{A^{(2)}} \rightarrow \underbrace{\begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{bmatrix}}_{A^{(3)}}$$

GE Algorithm

for i = 1, 2, ..., n-1
  for k = i+1, ..., n
    mult = a_{ki} / a_{ii}
    $a_{ki} = 0$ — not needed, but helpful for intuition
    for j = i+1, ..., n
      $a_{kj} = a_{kj}$ − mult $*a_{ij}$ — update row k
    end
    $b_k = b_k$ − mult $*b_i$ — update RHS
  end
end

At the end, $A^{(n-1)}x = b^{(n-1)}$, is solved by back substitution.

2.1.1 LU Factorizations

Theorem 2.1.1. *If A can be reduced to RREF without interchanging rows, then there is a unique factorization $A = LU$, where L is lower triangular with 1s on its diagonal (i.e. unit diagonal), and U is upper triangular. Moreover, Moreover,*

$$U = A^{(n-1)}, L = \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ mult & & 1 \end{bmatrix}$$

Proof. See the proof, starting on p144 of *Matrix Analysis and Applied Linear Algebra*, by Carl D. Meyer. □

Important Remark: Not every non-singular $n \times n$ matrix A has an LU-decomposition. E.g. $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$.

Then solving $Ax = b$ is equivalent to solving $LUx = b$. Let $y = Ux$. Then $Ly = b$. So we

1. Solve $Ly = b$ by forward solving, then
2. Solve $Ux = y$ by back solving.

Forward Solve Algorithm

for $i = 1, 2, \ldots, n$
  $y_i = b_i$
  for $j = 1, 2, \ldots, i-1$
    $y_i = y_i - l_{ij} * y_j \qquad \left(y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j\right)$
  end
end

Backward Solve Algorithm

for $i = n, \ldots, 1$
  $x_i = y_i$
  for $j = i+1, \ldots, n$
    $x_i = x_i - u_{ij} * x_j \qquad \left(x_i = y_i - \sum_{j=i+1}^{n} u_{ij} x_j\right)$
  end
  $x_i = x_i / u_{ii}$ % diagonal entries not necessarily 1
end

Complexity

- 1 **flop** $= +/-/*/\div$.
- Consider the forward solve algorithm. For each i, the j-loop performs $2(i-1)$ flops.

$$\begin{aligned} \text{Total flops} &= \sum_{i=1}^{n} 2(i-1) \\ &= 2\sum_{i=1}^{n} i - \sum_{i=1}^{n} 2 \\ &= 2\frac{n(n+1)}{2} - 2n \\ &= n^2 + n - 2n \\ &= n^2 - n \\ &\in O(n^2). \end{aligned}$$

- flops(back-solve) = $O(n^2)$ (Exercise).
- flops(LU factorization) = $\frac{2}{3}n^3 + O(n^2)$ (Exercise).

For large n, the factorization is more expensive than forward and back solving.

Special Linear Systems

- Exploit special structures of linear systems
- More efficient LU factorization

2.1.2 Symmetric Systems

- LDM^T factorization, variant of LU.

We do NOT assume that A is symmetric yet; the next Theorem applies whether A is symmetric or not.

Definition 2.1.2. *A **principal submatrix** is a smaller matrix constructed by deleting rows and corresponding columns.*

Some examples are

$$\begin{bmatrix} \cancel{1} & \cancel{2} & \cancel{3} & \cancel{4} & \cancel{5} \\ \cancel{6} & 7 & 8 & 9 & \cancel{10} \\ \cancel{11} & 12 & 13 & 14 & \cancel{15} \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix} \rightarrow \begin{bmatrix} 19 & 20 \\ 24 & 25 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & \cancel{2} & 3 & 4 & 5 \\ \cancel{6} & 7 & 8 & 9 & \cancel{10} \\ 11 & 12 & 13 & 14 & 15 \\ \cancel{16} & 17 & 18 & 19 & \cancel{20} \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 3 & 5 \\ 11 & 13 & 15 \\ 21 & 23 & 25 \end{bmatrix}.$$

The following results are reproduced from Chapter 3 of *Matrix Analysis and Applied Linear Algebra*, by Carl D. Meyer.

Definition 2.1.3. *The **leading principal submatrices** of A are defined to be those submatrices taken from the upper-left-hand corner of A. That is*

$$\begin{aligned} A_1 &= \begin{bmatrix} a_{11} \end{bmatrix}, \\ A_2 &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \\ &\vdots \\ A_{n-1} &= \begin{bmatrix} a_{11} & \cdots & a_{1\ n-1} \\ \vdots & & \vdots \\ a_{n-1\ 1} & \cdots & a_{n-1\ n-1} \end{bmatrix}, \\ A_n &= A \end{aligned}$$

Theorem 2.1.4. *Each of the following statements is equivalent to saying that a non-singular $n \times n$ matrix A possesses an LU-factorization.*

1. *A zero pivot does not emerge during row-reduction to upper-triangular form with Type III operations.*

2. Each leading principal submatrix A_k is non-singular.

Proof.
- We will prove the statement concerning the leading principal submatrices and leave the proof concerning the nonzero pivots as an exercise.
- First, assume that A has an LU-factorization.
  - For any $1 \leq k \leq n$, partition A as
$$\begin{aligned} A &= LU \\ &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} \\ &= \begin{bmatrix} L_{11}U_{11} & * \\ * & * \end{bmatrix}, \end{aligned}$$
    where L_{11} and U_{11} are both $k \times k$.
  - Then $A_k = L_{11}U_{11}$ is non-singular, because both of L_{11} and U_{11} are non-singular (each is triangular, with non-zero diagonal entries).
  - Since k was arbitrary, this shows that all of the leading principal submatrices of A are nonsingular.
- Second, assume that the leading principal submatrices of A are all non-singular.
  - Let $1 \leq k \leq n$ be arbitrary.
  - We will prove by induction on k that each A_k has an LU-factorization.
  - Then, since $A = A_n$, it follows that A has an LU-factorization.
  - Base ($k = 1$):
    - $A_1 = \begin{bmatrix} a_{11} \end{bmatrix}$, so the assumption that A_1 is non-singular guarantees that $a_{11} \neq 0$.
    - Then $A_1 = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} a_{11} \end{bmatrix}$ is an LU-factorization of A_1.
    - This completes the base case.
  - Induction ($k > 1$):
    - The induction hypothesis is that all A_ℓ, for $1 \leq \ell < k$, have LU-factorizations.
    - In particular, A_{k-1} has an LU-factorization.
    - Write $A_{k-1} = L_{k-1}U_{k-1}$.
    - By assumption, A_{k-1} is non-singular.
    - Therefore $A_{k-1}^{-1} = U_{k-1}^{-1}L_{k-1}^{-1}$.
    - Define
$$\begin{aligned} c^T &= \text{the first } k-1 \text{ components of the } k^{th} \text{ row of } A_k, \\ b &= \text{the first } k-1 \text{ components of the } k^{th} \text{ column of } A_k, \\ \alpha_k &= \text{the } (k,k) \text{ entry of } A_k. \end{aligned}$$
    - With this notation, we can write
$$A_k = \begin{bmatrix} A_{k-1} & b \\ c^T & \alpha_k \end{bmatrix}.$$
    - I claim that the following is an LU-factorization of A_k:
$$\underbrace{\begin{bmatrix} L_{k-1} & 0 \\ c^T U_{k-1}^{-1} & 1 \end{bmatrix}}_{L_k} \underbrace{\begin{bmatrix} U_{k-1} & L_{k-1}^{-1} b \\ 0 & \alpha_k - c^T A_{k-1}^{-1} b \end{bmatrix}}_{U_k}$$

* We verify that this works in each of the 4 blocks of A_k.
  - top-left $k-1 \times k-1$ block: $L_{k-1}U_{k-1} = A_{k-1}$, by the induction hypothesis.
  - row k, first $k-1$ entries: $\underbrace{\begin{bmatrix} c^T U_{k-1}^{-1} & 1 \end{bmatrix}}_{1\times k} \underbrace{\begin{bmatrix} U_{k-1} \\ 0 \end{bmatrix}}_{k\times k-1} = c^T$.
  - column k, first $k-1$ entries: $\underbrace{\begin{bmatrix} L_{k-1} & 0 \end{bmatrix}}_{k-1\times k} \underbrace{\begin{bmatrix} L_{k-1}^{-1} b \\ \alpha_k - c^T A_{k-1}^{-1} b \end{bmatrix}}_{k\times 1} = b$.
  - k,k entry: $\underbrace{\begin{bmatrix} c^T U_{k-1}^{-1} & 1 \end{bmatrix}}_{1\times k} \underbrace{\begin{bmatrix} L_{k-1}^{-1} b \\ \alpha_k - c^T A_{k-1}^{-1} b \end{bmatrix}}_{k\times 1} = c^T \underbrace{U_{k-1}^{-1} L_{k-1}^{-1}}_{=A_{k-1}^{-1}} b + \alpha_k - c^T A_{k-1}^{-1} b = \alpha_k$.
* Observe that
  - L_k has 1s on its diagonal, and
  - U_k has non-zeros on its diagonal. The fact that $\alpha_k - c^T A_{k-1}^{-1} b \neq 0$ follows, because A_k and L_k are both non-singular, hence $U_k = L_k^{-1} A_k$ must also be non-singular.
* This completes the induction step, and hence the proof.

□

Theorem 2.1.5. *If all the leading principal submatrices of A are nonsingular, then there exist unique unit lower diagonal matrices L and M, and a unique diagonal matrix D such that*

$$A = LDM^T$$

Proof.
- Uniquely factor $A = LU$, with L unit lower triangular.
- Define $D = diag(d_1, \ldots, d_n), d_i = u_{ii},\ 1 \leq i \leq n$.
- Note, all $d_i \neq 0$, by the hypothesis that A's leading principal submatrices are all non-singular.
- Hence D^{-1} exists (its diagonal entries are the reciprocals of D's diagonal entries).
- Let $M^T = D^{-1}U$.
- Observe that $D^{-1}U$ is upper triangular, and moreover, it has 1s on its diagonal (by the construction of D^{-1}).
- This says that M^T is unit upper triangular.
- Therefore M is unit lower triangular.
- Thus $A = LU = LD(D^{-1}U) = LDM^T$.

□

Remarks:

1. LU-factorization, and hence LDM-factorization, lie in $O(n^3)$.

Theorem 2.1.6. *Keep the hypotheses on A from Theorem 2.1.5. If A is symmetric, then $A = LDL^T$.*

Proof.
- By Theorem 2.1.5, there is a unique factorization $A = LDM^T$.
- Since A is symmetric, we have

$$\begin{aligned} A &= A^T \\ &= (LDM^T)^T \\ &= (M^T)^T D^T L^T \\ &\underbrace{=}_{D \text{ is diagonal, hence symmetric}} MDL^T. \end{aligned}$$

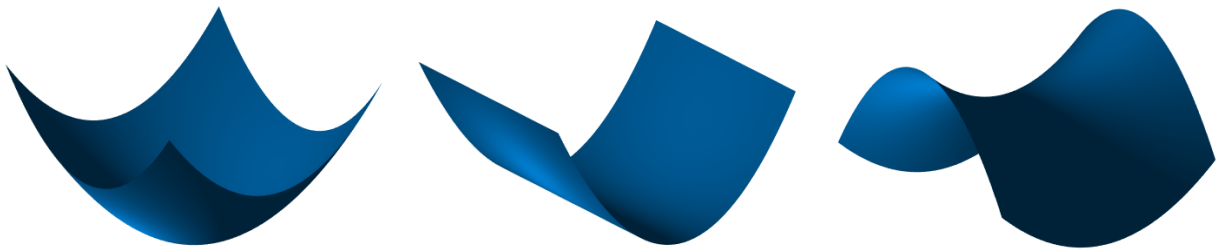
- By the uniqueness of the LDM-factorization, we have $M = L$.

□

2.1.3 Positive Definite Systems

Definition 2.1.7. *An $n \times n$ symmetric matrix A is **positive definite** if $x^T Ax > 0$, for all non-zero $n \times 1$ matrices x.*

Roughly speaking, Definition 2.1.7 generalizes a definition for positive scalars, i.e., $a \in \mathbb{R}$ is positive if $xax > 0, \forall x \in \mathbb{R}, x \neq 0$. Consider the function $f(x) = x^T Ax$, which is quadratic and A contains the coefficients. Positive definiteness essentially asks if f is convex. Figure 2.1 shows examples of f with different A matrices.



positive definite | positive semidefinite | indefinite

Figure 2.1: Plotting of $f = x^T Ax$ as a height function with $x \in \mathbb{R}^2$ and different $A \in \mathbb{R}^{2\times 2}$.

Remarks:

1. There is not universal agreement among mathematicians that we should only ask if a matrix is positive definite if we already know that it is symmetric.
2. E.g. $A = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$ satisfies $x^T Ax > 0$, for all $0 \neq x \in \mathbb{R}^2$, but is clearly not symmetric.
3. In this course, we will only care if a matrix is positive definite when we already know that it is symmetric.
4. This is why we make being symmetric part of the definition of being positive definite.
5. Keep in mind that the definition of positive definiteness may be different in other contexts, outside of this course.

Equivalent Characterizations of Being Positive Definite

1. All eigenvalues are strictly positive. (This makes sense because the eigenvalues of a symmetric matrix are all real.)
2. All pivots are strictly positive. (using the fact that for a symmetric matrix, the signs of the pivots are the signs of the eigenvalues.)
3. The k^{th} pivot of a matrix is
$$d_k = \frac{\det(A_k)}{\det(A_{k-l})},$$
where A_k is the k^{th} leading principal submatrix. All the pivots will be positive if and only if $det(A_k) > 0$ for all $1 \leq k \leq n$. So, if the determinants of the leading principal submatrices are positive, then the matrix is positive definite.
4. A matrix A is positive definite if and only if it can be written as $A = R^T R$ for some possibly rectangular matrix R with independent columns.

Remarks:

1. I will not prove these equivalences in class.
2. If any student specifically asks to see the proofs, then I will type them up and post them.
3. You can safely infer from the above two comments that you are not responsible for knowing these proofs for this course.

Theorem 2.1.8. *If A is positive definite, then A^{-1} exists.*

A useful result for PD matrices is given in Theorem 2.1.9.

Theorem 2.1.9. *If $A \in \mathbb{R}^{n\times n}$ is PD and $X \in \mathbb{R}^{n\times k}$ has rank $k \leq n$, then $B = X^T AX$ is also PD (i.e., $z^T Bz > 0, \forall z \in \mathbb{R}^k, z \neq 0$).*

$$\underset{n\times n}{\begin{bmatrix} & A & \\ & & \\ & & \end{bmatrix}} \quad \underset{n\times k}{\begin{bmatrix} & X & \\ & & \\ & & \end{bmatrix}} \quad \underset{k\times k}{\begin{bmatrix} & B & \end{bmatrix}} \quad \begin{bmatrix} z_1 \\ \vdots \\ z_k \end{bmatrix} \quad \begin{bmatrix} x_1 \\ \\ \vdots \\ \\ x_n \end{bmatrix}$$

The above diagram shows the sizes of all the matrices/vectors in Theorem 2.1.9.

Proof.

- Consider any $0 \neq z \in \mathbb{R}^k$, then
$$z^T Bz = z^T X^T AXz = (Xz)^T A(Xz).$$
- Let $x = Xz$, which is a vector in $\mathbb{R}^n$.
- If $x \neq 0$ then we are finished because $(Xz)^T A(Xz) = x^T Ax > 0$ since A is PD.
- When can $x = 0$? This is equivalent to asking what the null space of X is.
- Since X has rank k it is full rank.

- By the rank-nullity theorem $\dim(\text{null}(X)) = \text{nullity}(X) = 0$.
- Hence, the null space of X contains only the zero-vector.
- Thus, $x = 0$ only if $z = 0$. So $z^T Bz = (Xz)^T A(Xz) > 0$ for all $z \neq 0 \Rightarrow B$ is PD.

□

Corollary 2.1.10. *If A is PD, then all its principal submatrices are PD. In particular, all diagonal entries are positive.*

Proof. Each diagonal entry is a principal submatrix with all other rows/columns deleted. You can design (identity-like) matrices X to "pick out" arbitrary principal submatrices using $X^T AX$ (which is PD by Theorem 2.1.9), e.g.,

$$A = \begin{bmatrix} -1 & 2 & 5 \\ 2 & 4 & -4 \\ 5 & -4 & 7 \end{bmatrix}, \qquad X = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \Rightarrow X^T AX = \begin{bmatrix} -1 & 5 \\ 5 & 7 \end{bmatrix}.$$

□

Remarks:

1. The converse of the Corollary 2.1.10 holds, because A is a principal submatrix of itself.

Corollary 2.1.11. *If A is PD, so that $A = LDL^T$, then the diagonal matrix D has strictly positive entries.*

Proof.
- Since L is unit lower triangular, therefore it is invertible. Hence we have

$$\begin{aligned} A &= LDL^T \\ L^{-1}AL^{-T} &= D. \end{aligned}$$

- By Theorem 2.1.9, $D = L^{-1}AL^{-T}$ is PD.
- By Corollary 2.1.10, D's diagonal entries are all positive.

□

3 Lecture 03: Solving Linear Systems

Outline

1. Solving Linear Systems
   (a) Symmetric Positive Definite (SPD) Systems
      i. Constructing the Cholesky Factor
   (b) Banded Systems
   (c) General Sparse Matrices

3.1 Solving Linear Systems

3.1.1 Symmetric Positive Definite (SPD) Systems

Theorem 3.1.1. *If A is SPD, then there exists unique lower triangular G, with strictly positive entries on its diagonal, such that*

$$A = GG^T.$$

Proof. By Theorem 2.1.6 and Corollary 2.1.11, write $A = LDL^T$, for some lower triangular L and $D = diag(d_1, \ldots, d_n), d_i > 0$.
Define $D^{\frac{1}{2}}$, one co-ordinate (i), at a time, as follows:

- If the i^{th} diagonal entry of L is positive, then the i^{th} diagonal entry of $D^{\frac{1}{2}}$ is $\sqrt{d_i}$.
- Otherwise, if the i^{th} diagonal entry of L is negative, then the i^{th} diagonal entry of $D^{\frac{1}{2}}$ is $-\sqrt{d_i}$.

Let $G = LD^{\frac{1}{2}}$. Then G is lower triangular, with strictly positive entries on its diagonal. Further, $GG^T = LD^{\frac{1}{2}}(LD^{\frac{1}{2}})^T = LD^{\frac{1}{2}}D^{\frac{1}{2}}L^T = LDL^T = A$.

Explanation for why the Cholesky Factorization is Unique:

- Let $A = GG^T = HH^T$, for some lower triangular H with strictly positive entries on its diagonal.
- First, note that $I = H^{-1}GG^TH^{-T}$:

$$\begin{aligned} HH^T &= A \\ &= GG^T, \text{ so that} \\ H^{-1}(GG^T)H^{-T} &= H^{-1}(HH^T)H^{-T} \\ &= (H^{-1}H)(H^TH^{-T}) \\ &= I \end{aligned}$$

- Then we have

$$\begin{aligned} I &= H^{-1}GG^TH^{-T} \\ &= H^{-1}G(H^{-1}G)^T, \text{ and therefore} \qquad (3.1) \\ H^{-1}G &= (H^{-1}G)^{-T}. \qquad (3.2) \end{aligned}$$

- The LHS of (3.2) is a product of lower triangular matrices, hence it is lower triangular.
- The RHS of (3.2) is upper triangular.
- Let $E = H^{-1}G$.
- Then since E is both upper and lower triangular, therefore E is diagonal. In particular $E^T = E$.
- Therefore (3.1) implies that $E^2 = I$, in other words E is diagonal, with ± 1 entries on its diagonal.
- Because $G = HE$, therefore the two factorizations can differ only by the signs of their columns.
- But since the diagonal entries of both G and H must be strictly positive, therefore they must be equal.

□

The factorization $A = GG^T$ is called the **Cholesky factorization**. The matrix G is referred to as the **Cholesky factor**.

Q & A

1. Will we need to reproduce proofs on Quizzes / Assignments / Exams?
   A: Not on Quizzes / Assignments, since they will be open book. I may ask for short, new proofs, on Assignments. It is also possible that I will ask for proofs on Exams. My idea about Quizzes, so far, is to have you work out some examples of the results we are proving in class.

Constructing the Cholesky Factor We now discuss how to construct the Cholesky factor. Lecture 23 of Trefethen and Bau or Section 4.2.5 of Golub and Van Loan are good supplemental reads. First, view A as

$$A = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix},$$

where $\alpha = a_{11} \in \mathbb{R}$, $v = a_{2:n,1} \in \mathbb{R}^{n-1}$, and $B = a_{2:n,2:n} \in \mathbb{R}^{(n-1)\times(n-1)}$.

The colon notation follows from MATLAB.

- For a matrix A, $a_{i,:}$ and $a_{:,j}$ denote the i-th row and the j-th column, respectively.
- Moreover, $a_{i:j,p:q}$ denotes the submatrix with rows from i to j and columns from p to q.
- For example, $a_{2:n,1}$ corresponds to entries in the 1^{st} column of A, from the 2^{nd} to the n^{th} row.

Cholesky factorization can intuitively be thought of as applying Gaussian elimination in a "symmetric way". The goal of Gaussian elimination was to zero-out the column below by subtracting multiples of the current row. The "work-in-progress" LU-factorization of A after the first step of Gaussian elimination can be viewed as

$$A = \begin{bmatrix} 1 & 0 \\ \frac{v}{\alpha} & I \end{bmatrix} \begin{bmatrix} \alpha & v^T \\ 0 & B - \frac{vv^T}{\alpha} \end{bmatrix},$$

where the first column of L becomes

$$\begin{bmatrix} 1 \\ \frac{v}{\alpha} \end{bmatrix}.$$

Reminder/Explanation about LU-factorization:

1. To eliminate the entries of v below α in column 1, the multipliers are $-\frac{v}{\alpha}$.
2. Thus, per our LU-factorization procedure, we record the negatives of these multipliers, namely $\frac{v}{\alpha}$, in column 1 of our L matrix.
3. Why the $B - \frac{vv^T}{\alpha}$ block is correct:
   (a) As above, the multipliers $-\frac{v}{\alpha}$ are required by the entries below the pivot in column 1.
   (b) Because of symmetry, the entries that get applied to the lower rows are v^T.
   This explains where the term $-\frac{vv^T}{\alpha}$ comes from.
4. The provided multiplication creates a $\frac{vv^T}{\alpha}$ term, which cancels with $-\frac{vv^T}{\alpha}$, to leave the required B in the bottom right of the product matrix.

However, applying just Gaussian elimination to get an LU factorization does not take advantage of symmetry. Cholesky factorization therefore aims to zero out the corresponding row also to remain symmetric. The first stage of Cholesky factorization is

$$A = \begin{bmatrix} \sqrt{\alpha} & 0 \\ \frac{v}{\sqrt{\alpha}} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & B - \frac{vv^T}{\alpha} \end{bmatrix} \begin{bmatrix} \sqrt{\alpha} & \frac{v^T}{\sqrt{\alpha}} \\ 0 & I \end{bmatrix},$$

which gives the first column of G as

$$\begin{bmatrix} \sqrt{\alpha} \\ \frac{v}{\sqrt{\alpha}} \end{bmatrix}.$$

Brief Explanation of the Change From LU to Cholesky

1. It is an exercise to verify that this "work-in-progress" factorization of A is correct.
   (a) The job of the LH factor is to restore the entries below the pivot.
   (b) The job of the RH factor is to restore the entries to the right of the pivot.
2. In LU, we produce the L-factor on the left, and the U factor by modifying the original A, on the right.
3. In Cholesky, we will produce by the end,

$$G \quad \underbrace{I}_{\text{where } A \text{ was, originally}} \quad G^T$$

This first step is derived by considering that the final form must be

$$A = GG^T = \begin{bmatrix} g_{11} & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} g_{11} & G_{21}^T \\ 0 & G_{22}^T \end{bmatrix} = \begin{bmatrix} g_{11}^2 & g_{11}G_{21}^T \\ g_{11}G_{21} & G_{21}G_{21}^T + G_{22}G_{22}^T \end{bmatrix}.$$

Therefore,

$$A = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix} = \begin{bmatrix} g_{11}^2 & g_{11}G_{21}^T \\ g_{11}G_{21} & G_{22}G_{22}^T + G_{21}G_{21}^T \end{bmatrix},$$

implies

$$\begin{aligned} \alpha &= (g_{11})^2 \\ \Rightarrow g_{11} &= \sqrt{\alpha}, \\ v &= g_{11} G_{21} \\ \Rightarrow G_{21} &= \frac{v}{g_{11}} \\ &= \frac{v}{\sqrt{\alpha}}. \end{aligned}$$

This provides the ingredients (namely α and v) needed to compute the matrix to be processed at the next step (namely $B - \frac{vv^T}{\alpha}$).

The Cholesky factorization algorithm then works recursively on the lower block $B - \frac{vv^T}{\alpha}$ since it is also SPD. To see that $B - \frac{vv^T}{\alpha}$ is SPD consider multiplying by the full rank matrix

$$X = \begin{bmatrix} 1 & -\frac{v^T}{\alpha} \\ 0 & I \end{bmatrix}.$$

We have that

$$X^T A X = \begin{bmatrix} \alpha & 0 \\ 0 & B - \frac{vv^T}{\alpha} \end{bmatrix},$$

hence by Theorem 2.1.9 and Corollary 2.1.10 the principal submatrix $B - \frac{vv^T}{\alpha}$ is PD. It is also symmetric since $X^T A X = (X^T A X)^T$ and A is symmetric.

So we can Cholesky factor $B - \frac{vv^T}{\alpha}$ as $B - \frac{vv^T}{\alpha} = G_1 G_1^T$. The recursion continues eliminating one row/column at a time. The Cholesky factor of A itself will have the form

$$G = \begin{bmatrix} \sqrt{\alpha} & 0 \\ \frac{v}{\sqrt{\alpha}} & G_1 \end{bmatrix}.$$

Cholesky Example:

- So that we will know what our goal is, we first choose a G. Then we compute our starting point, namely the matrix A, via $A = GG^T$.
- With the help of a student, we selected

$$\begin{aligned} G &= \begin{bmatrix} 2 & 0 & 0 \\ 3 & 3 & 0 \\ -4 & -6 & 5 \end{bmatrix}, \text{ so that} \\ A &= GG^T \\ &= \begin{bmatrix} 2 & 0 & 0 \\ 3 & 3 & 0 \\ -4 & -6 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & -4 \\ 0 & 3 & -6 \\ 0 & 0 & 5 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 6 & -8 \\ 6 & 18 & -30 \\ -8 & -30 & 77 \end{bmatrix}. \end{aligned}$$

- A is clearly symmetric.
- To make absolutely certain that A has a Cholesky factorization, we verify that A is also positive definite. It is an exercise to verify that row reducing A yields $\begin{bmatrix} 4 & 6 & -8 \\ 0 & 9 & -18 \\ 0 & 0 & 25 \end{bmatrix}$. From this matrix, we can see that the pivots of A are $4, 9$ and 25. They are all positive, and hence A is positive definite.
- We carry out the algorithm to compute the Cholesky factor G.
  1. Writing $\begin{bmatrix} 4 & 6 & -8 \\ 6 & 18 & -30 \\ -8 & -30 & 77 \end{bmatrix} = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix}$, we have

$$\begin{aligned} \alpha &= 4 \\ v &= \begin{bmatrix} 6 \\ -8 \end{bmatrix} \\ B &= \begin{bmatrix} 18 & -30 \\ -30 & 77 \end{bmatrix}. \end{aligned}$$

  and therefore we obtain

$$\begin{aligned} g_{11} &= \sqrt{\alpha} \\ &= \sqrt{4} \\ &= 2 \\ G_{21} &= \frac{v}{g_{11}} \\ &= \frac{\begin{bmatrix} 6 \\ -8 \end{bmatrix}}{2} \\ &= \begin{bmatrix} 3 \\ -4 \end{bmatrix}, \text{ so that} \\ B - \frac{vv^T}{\alpha} &= \begin{bmatrix} 18 & -30 \\ -30 & 77 \end{bmatrix} - \frac{\begin{bmatrix} 6 \\ -8 \end{bmatrix} \begin{bmatrix} 6 & -8 \end{bmatrix}}{4} \\ &= \begin{bmatrix} 18 & -30 \\ -30 & 77 \end{bmatrix} - \frac{\begin{bmatrix} 36 & -48 \\ -48 & 64 \end{bmatrix}}{4} \\ &= \begin{bmatrix} 18 & -30 \\ -30 & 77 \end{bmatrix} - \begin{bmatrix} 9 & -12 \\ -12 & 16 \end{bmatrix} \\ &= \begin{bmatrix} 9 & -18 \\ -18 & 61 \end{bmatrix} \end{aligned}$$

2. Writing $\begin{bmatrix} 9 & -18 \\ -18 & 61 \end{bmatrix} = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix}$, we have

$$\begin{aligned} \alpha &= 9 \\ v &= \begin{bmatrix} -18 \end{bmatrix} \\ B &= \begin{bmatrix} 61 \end{bmatrix}. \end{aligned}$$

and therefore we obtain

$$\begin{aligned} g_{11} &= \sqrt{\alpha} \\ &= \sqrt{9} \\ &= 3 \\ G_{21} &= \frac{v}{g_{11}} \\ &= \frac{\begin{bmatrix} -18 \end{bmatrix}}{3} \\ &= \begin{bmatrix} -6 \end{bmatrix}, \text{ so that} \\ B - \frac{vv^T}{\alpha} &= \begin{bmatrix} 61 \end{bmatrix} - \frac{\begin{bmatrix} -18 \end{bmatrix}\begin{bmatrix} -18 \end{bmatrix}}{9} \\ &= \begin{bmatrix} 61 \end{bmatrix} - \begin{bmatrix} 36 \end{bmatrix} \\ &= \begin{bmatrix} 25 \end{bmatrix} \end{aligned}$$

3. Writing $\begin{bmatrix} 25 \end{bmatrix} = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix}$, we have

$$\alpha = 25$$

and therefore we obtain

$$\begin{aligned} g_{11} &= \sqrt{\alpha} \\ &= \sqrt{25} \\ &= 5, \end{aligned}$$

at which point, we are finished.

- We have now recovered all the diagonal and subdiagonal entries from the original choice of G.

Cost of Cholesky Factorization Algorithm 9.5 gives pseudocode for the in-place Cholesky factorization. This implementation exploits symmetry by working only on sub-diagonal entries. In the end, the lower triangle is the Cholesky factor G.

Algorithm 3.1 : Cholesky Factorization

```
for k = 1, ..., n                                   ▷ iterate down rows
    a_kk = √a_kk                                    ▷ factor diagonal element (√α)
    for i = k + 1, ..., n                           ▷ go over rows
        a_ik = a_ik / a_kk                          ▷ update current column entries below diagonal (v/√α)
    end for
    for j = k + 1, ..., n                           ▷ go over columns
        for i = j, ..., n                           ▷ go over rows
            a_ij = a_ij − a_ik ∗ a_jk               ▷ update lower right block B − vv^T/α
        end for
    end for
end for
```

Note that Algorithm 9.5 assumes that the diagonal entries a_{kk} are non-zero. Problems arise when these entries are zero or close to zero. We will address this issue through **pivoting**, when we discuss the stability of factorizations.

The cost of Cholesky factorization can be estimated by considering just the inner most loop. For that loop there is one subtraction and one multiplication. So the FLOP count is

$$\sum_{k=1}^{n} \sum_{j=k+1}^{n} \sum_{i=j}^{n} 2 = \frac{n^3}{3} + O\left(n^2\right),$$

which is calculated as displayed below. As promised Cholesky factorization is half that of LU factorization, which had a cost of $\frac{2n^3}{3} + O\left(n^2\right)$ FLOPs.

Detailed Computation:

$$
\begin{aligned}
& \sum_{k=1}^{n}\sum_{j=k+1}^{n}\sum_{i=j}^{n} 2 \\
= \; & 2\sum_{k=1}^{n}\sum_{j=k+1}^{n}\sum_{i=j}^{n} 1 \\
= \; & 2\sum_{k=1}^{n}\sum_{j=k+1}^{n} (n-(j-1)) \\
= \; & 2\sum_{k=1}^{n}\sum_{j=k+1}^{n} ((n+1)-j) \\
= \; & 2\sum_{k=1}^{n}\left[(n-(k+1-1))(n+1) - \sum_{j=k+1}^{n} j\right] \\
= \; & 2\sum_{k=1}^{n}\left[(n-k)(n+1) - \left(\frac{n(n+1)}{2} - \frac{k(k+1)}{2}\right)\right] \\
= \; & 2\sum_{k=1}^{n}\left[n^2+n-kn-k-\frac{n^2}{2}-\frac{n}{2}+\frac{k^2}{2}+\frac{k}{2}\right] \\
= \; & 2\sum_{k=1}^{n}\left[\frac{k^2}{2}+k\left(-n-\frac{1}{2}\right)+\left(\frac{n^2}{2}+\frac{n}{2}\right)\right] \\
= \; & 2\left[\left(\frac{1}{2}\right)\frac{n(n+1)(2n+1)}{6}+\left(-n-\frac{1}{2}\right)\frac{n(n+1)}{2}+\left(\frac{n^2}{2}+\frac{n}{2}\right)n\right] \\
= \; & 2n(n+1)\left[\frac{(2n+1)}{12}-\left(n+\frac{1}{2}\right)\frac{1}{2}+\frac{n}{2}\right] \\
= \; & n(n+1)\left[\frac{(2n+1)}{6}-\left(n+\frac{1}{2}\right)+n\right] \\
= \; & \frac{n(n+1)}{6}[2n+1-3] \\
= \; & \frac{n(n+1)}{6}(2n-2) \\
= \; & \frac{n(n+1)(n-1)}{3} \\
= \; & \frac{n^3}{3}+O\left(n^2\right).
\end{aligned}
$$

3.1.2 Banded Matrices

Banded matrices have nonzero entries only in "bands" adjacent to the main diagonal. Some example banded matrices are (empty entries are zero)

$$\begin{bmatrix} 9 & 5 & & \\ & 3 & 8 & \\ & & 2 & 7 \\ & & & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 5 & 6 & & & & \\ 7 & 7 & 1 & & & \\ 4 & 2 & 9 & 2 & & \\ & 7 & 5 & 3 & 8 & \\ & & 3 & 8 & 2 & 7 \\ & & & 4 & 7 & 1 \end{bmatrix}. \tag{3.3}$$

Definition 3.1. *The matrix $A = [a_{ij}]$ has*

1. ***upper bandwidth**, q, if $a_{ij} = 0$ for $j > i + q$, and*
2. ***lower bandwidth**, p, if $a_{ij} = 0$ for $i > j + p$.*

The general form of a banded matrix given in Definition 3.1 is

$$\begin{bmatrix} \times & \cdots & \times & & & & \\ \vdots & \ddots & & \ddots & & & \\ \times & & & & & & \\ & \ddots & & \ddots & & \ddots & \\ & & & & & & \times \\ & & & \ddots & & \ddots & \vdots \\ & & & & \times & \cdots & \times \end{bmatrix}.$$

In the examples in (3.3) the first matrix has $q = 1, p = 0$ and the second has $q = 1, p = 2$. Exercise: how might you store these banded matrices efficiently?

Q & A

1. Can there be a piece, inside the band (i.e. parallel to the band), with all 0 entries?
   A: Yes! Definition 3.3 says nothing about what happens within the band. In the most extreme case, the zero matrix trivially satisfies the definition of a banded matrix!
2. Do upper lower triangular matrices satisfy Definition 3.3?
   A: Yes!
   (a) An upper triangular matrix satisfies $p = 0$.
   (b) A lower triangular matrix satisfies $q = 0$.

Factoring Banded Matrices If A is banded then so are the factorizations LU, LDM^T, GG^T.

Theorem 3.1. *Let $A = LU$. If A has upper bandwidth q and lower bandwidth p, then U has upper bandwidth q and L has lower bandwidth p.*

Matrix Type	Lower Bandwidth p	Upper Bandwidth q
Diagonal	0	0
Upper Triangular	0	$n-1$
Lower Triangular	$m-1$	0
Tridiagonal	1	1
Upper Bidiagonal	0	1
Lower Bidiagonal	1	0
Upper Hessenberg	1	$n-1$
Lower Hessenberg	$m-1$	1

Table 3.1: Common types of matrices $A \in \mathbb{R}^{m \times n}$ in this course and their bandwidths.

$$\underset{A}{\begin{bmatrix} \times & \cdots & \times & & & \\ \vdots & \ddots & & \ddots & & \\ \times & & & & & \\ & \ddots & & \ddots & & \ddots \\ & & & & & \times \\ & & & \ddots & & \ddots \\ & & & & \times & \cdots \end{bmatrix}} = \underset{L}{\begin{bmatrix} \times & & & & & \\ \vdots & \ddots & & & & \\ \times & & & & & \\ & \ddots & & \ddots & & \\ & & & & & \\ & & & \ddots & & \ddots \\ & & & & \times & \cdots \end{bmatrix}} \underset{U}{\begin{bmatrix} \times & \cdots & \times & & & \\ & \ddots & & \ddots & & \\ & & & & & \\ & & & \ddots & & \ddots \\ & & & & & \times \\ & & & & \ddots & \vdots \\ & & & & & \times \end{bmatrix}}.$$

A special case of banded matrices is **tridiagonal** matrices, which have $p = q = 1$. One can show that the flop count for LU factorization of tridiagonal matrices is $O(n)$. Table 3.1 gives other examples of important matrices we will see in this course.

Cost of Banded LU Factorization

Algorithm 3.2 : Banded LU Factorization

```
for k = 1, ..., n − 1                                   ▷ iterate over rows
    for i = k + 1, ..., min(k + p, n)                   ▷ go over rows within band
        a_ik = a_ik / a_kk                              ▷ determine multiplicative factors
    end for
    for i = k + 1, ..., min(k + p, n)                   ▷ go over rows within band
        for j = k + 1, ..., min(k + q, n)               ▷ go over columns within band
            a_ij = a_ij − a_ik ∗ a_kj                   ▷ subtract scaled row data only in non-zero bands
        end for
    end for
end for
```

- Algorithm 3.1.2 gives the banded version of LU factorization.
- We assume diagonal entries $a_{kk} \neq 0$ for now in Algorithm 3.1.2.
- Banded LU factorization is most beneficial when there are many zeros in A.

- In other words, when the upper/lower bandwidths (q/p) are small relative to the size of the matrix (n).
- If $n \gg p$ and $n \gg q$, then banded LU is $\sim 2npq$ flops.
- This is much faster than naïve implementation of LU factorization $\sim \frac{2n^3}{3}$ flops that would operate on zero entries.
- For example, with $n = 300, p = 2, q = 2$, basic LU takes $\sim 18,000,000$ flops but banded LU takes only ~ 2400 flops.

Exercises:

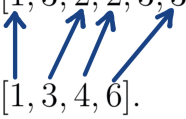
1. Verify flop count of banded LU factorization (leading order term).
2. Work out **efficient** algorithms for banded forward/backward triangular solves. That is, modify the standard forward/backward solve algorithms to avoid touching entries you **know** are always zero (based on bandwidth).

3.1.3 General Sparse Matrices

- Patterns other than just simple bands are also common.
- General **sparse matrices** (i.e. matrices having few non-zero entries) contain mostly zero entries, but non-zeros can occur on more than the diagonal bands.
- For many problems the (max) number of non-zeros per row is constant, i.e., the total number of non-zeros is $O(n)$.
- So we still only want to store the non-zero entries.
- Various storage formats (data structures) exist for sparse matrices.
- We will not be coding our own in this course, but it is useful to be aware of them.
- The simplest approach is to have a vector of one (i, j, value) triplet per non-zero, but this is inefficient.
- A more common storage structure is the Compressed Row Storage (CRS) (or Compressed Sparse Row (CSR)):
  - array of non-zero entries ("val") with length = number of non-zeros (nnz),
  - array of column indices ("colInd") with length = nnz, and
  - array of indices where each row starts ("rowPtr") with length = number of rows.

For example, consider the CSR structure for the following matrix

$$\begin{bmatrix} 2 & & 5 & \\ & 3 & & \\ & 6 & -3 & \\ & & 10 & 2 \end{bmatrix} \quad \Rightarrow \quad \begin{aligned} \text{val} &= [2, 5, 3, 6, -3, 10, 2], \\ \text{colInd} &= [1, 3, 2, 2, 3, 3, 4], \\ \text{rowPtr} &= [1, 3, 4, 6]. \end{aligned}$$



(We could include 0 for an empty row.)

Remark: We have **not** made a rigourous definition of a sparse matrix.

Factorization

- For LU factorization the main cost is the row subtraction step:

$$a_{ij} = a_{ij} - a_{ik}a_{kj}/a_{kk}.$$

- Since most entries are zero, our algorithms should skip operating on them.
- However, an **important point** to realize is that even if A is sparse, its factorization **may not** be!
- This happens because row subtractions can turn zeros into non-zero entries, which is referred to as "fill-in".
- A classic example is the "arrow matrix", which has fully dense (triangular) L and U factors

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

Exercises:

1. Can you see why the L and U are dense (in corresponding triangles)?
2. Therefore, what is the storage cost of the factors and what is the complexity of the factorization?

The key intuition for general sparse matrices is that we must reorder the system of equations. A matrix **reordering** permutes rows/columns to yield a matrix whose LU factorization suffers no fill-in (i.e., no new non-zeros). With the arrow matrix for example we can solve the same system, but reorder the equations so that

$$A = \begin{bmatrix} \times & & & & \times \\ & \times & & & \times \\ & & \times & & \times \\ & & & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ \times & \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & & & & \times \\ & \times & & & \times \\ & & \times & & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix} = LU.$$

Matrix reorderings will be discussed in more detail later.

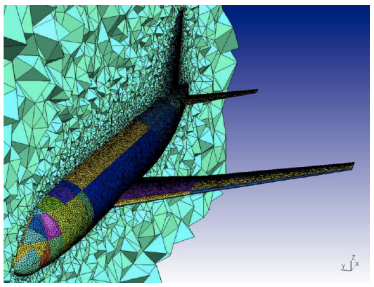
4 Lecture 04: Finite Differences for Modelling Heat Conduction

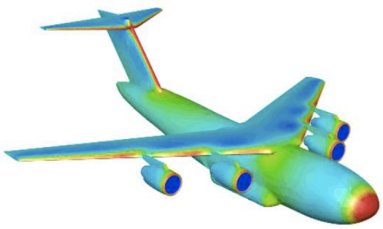
Outline

1. Finite Differences for Modelling Heat Conduction

This lecture covers an application of solving linear systems. Partial differential equations (PDEs) involve multivariable functions and (partial) derivatives. They describe numerous phenomena:

- Electromagnetism,
- Fluid flow,
- Sound propagation,
- Financial problems,
- Solid mechanics (engineering),
- Quantum mechanics,
- ...





The numerical solution of PDEs are a common source of sparse linear systems (e.g., finite difference/finite volume/finite element methods). This lecture introduces finite differences for a PDE describing heat conduction.

4.1 Finite Differences for Modelling Heat Conduction

Setup:

1. Suppose that we want to approximate the (unknown) **temperature function**, $T(x, y, z)$ (where x, y, z are spatial coordinates) in some 3-D solid object, **at equilibrium** (i.e. T does not vary with respect to time).
2. Suppose further that we have a given (known) **heat source function**, $f(x, y, z)$.

Then the heat distribution may be modelled using the **Poisson equation**:

$$\begin{aligned} f + \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) &\underbrace{=}_{\text{at equilibrium}} 0 \\ \Leftrightarrow -\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) &= f \\ \Leftrightarrow -\Delta T &= f. \end{aligned}$$

The differential operator

$$\Delta = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right)$$

is called the **Laplacian operator**.

1D Example Boundary conditions are also necessary to fully define the problem. Consider

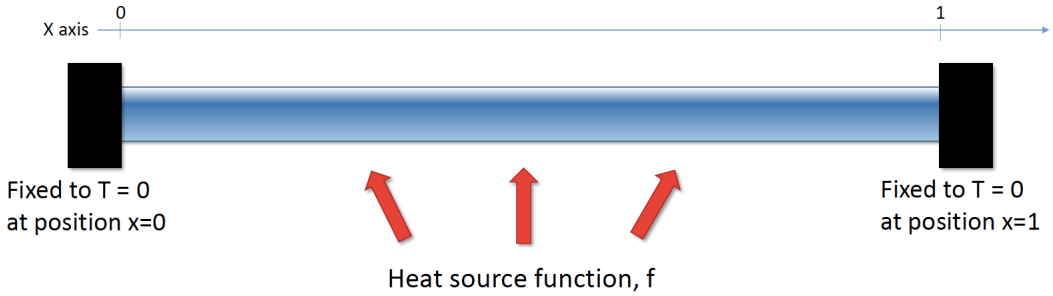
a 1D example where

$$-\frac{\partial^2 T}{\partial x^2} = f \quad \text{on } (0,1), \tag{4.4}$$

$$T(0) = 0, \tag{4.5}$$

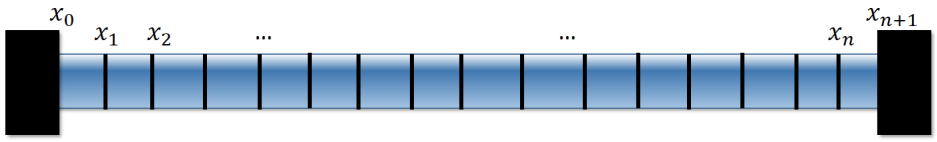
$$T(1) = 0. \tag{4.6}$$

Lines (4.5) and (4.6) are the boundary conditions. In this case the temperature T is zero at both $x = 0$ and $x = 1$. The figure below shows the domain pictorially.



We want to find an approximate numerical solution, given the source f and the boundary temperatures. Our approach here is to first discretize (subdivide) the material into finite subintervals. Then approximate the spatial derivatives with finite differences.

Discretizing the domain is done by chopping the length of the 1D bar into chunks. That is, define discrete points on the bar $0 = x_0 < x_1 < x_2 < \cdots < x_n < x_{n+1} = 1$, which are referred to as **gridpoints** x_i.



We let T_i denote the numerical approximation of the exact solution $T(x_i)$ for $i = 0, \ldots, n$.

Here we assume evenly spaced intervals. Define the grid spacing h as

$$\begin{aligned} h = x_i - x_{i-1} &= \frac{1}{n+1}, \\ &= \frac{\text{domain length}}{\#\text{ of intervals}}. \end{aligned}$$

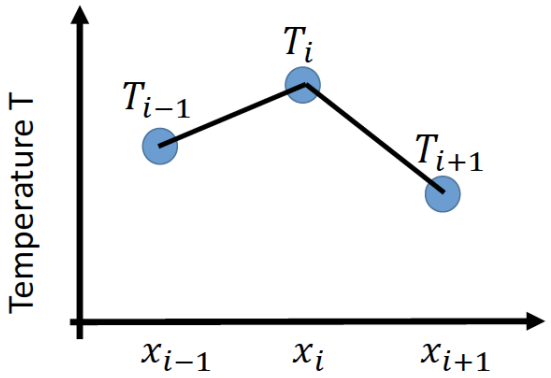
Due to the boundary conditions we know $T_0 = 0$ and $T_{n+1} = 0$. Therefore, the unknowns we must solve for are the n temperature values $T_1, T_2, \ldots, T_n$ (i.e., at non-boundary gridpoints). These gridpoints are referred to as the **active** gridpoints.

Now that the discrete domain is defined we discretize the partial derivatives. Recall, **finite differences** are one approach to obtain discrete approximations of derivatives. For example,

$$\frac{\partial T}{\partial x}(x_i) \approx \frac{T(x_i) - T(x_{i-1})}{x_i - x_{i-1}} \approx \frac{T_i - T_{i-1}}{h}. \tag{4.7}$$

We will use the **centered finite difference** approximation of $\frac{\partial^2 T}{\partial x^2}$, specifically

$$\begin{aligned} & \frac{\partial^2 T}{\partial x^2}(x_i) \\ \approx & \frac{\frac{T_{i+1}-T_i}{h} - \frac{T_i - T_{i-1}}{h}}{h} \\ = & \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2} \end{aligned} \tag{4.8}$$



It can be seen from (4.8), and the above figure, that T_i depends on its neighbours T_{i+1} and T_{i-1}. The resulting relationships between gridpoints determines T_i, for $i = 1, 2, \ldots, n$. Each gridpoint $i = 1, 2, \ldots, n$ gives one equation relating its value to its two neighbours:

$$-\left(\frac{T_{i+1} - 2T_i + T_{i-1}}{h^2}\right) = f_i, \qquad \text{for} \qquad i = 1, \ldots, n. \tag{4.9}$$

Equation (4.9) is our **discrete** equation approximating the **continuous** equation $-\frac{\partial^2 T}{\partial x^2} = f$.

The general form of the matrix equation from (4.9) is

$$\frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}\begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_{n-1} \\ T_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix}.$$

How The Boundary Conditions Determine The First And Last Rows

$$
\begin{aligned}
f_1 &= -\left(\frac{T_2 - 2T_1 + \underbrace{T_0}_{=0}}{h^2}\right) \\
&= \frac{2T_1 - T_2}{h^2} \\
f_n &= -\left(\frac{\underbrace{T_{n+1}}_{=0} - 2T_n + T_{n-1}}{h^2}\right) \\
&= \frac{-T_{n-1} + 2T_n}{h^2}
\end{aligned}
$$

What can we say about the matrix structure? It is a banded matrix, but more specifically symmetric and tridiagonal.

Heat Conduction in 2D Plate Consider the 2D domain of a square plate with zero temperature boundaries. We want to determine the heat distribution $T(x, y)$ on the interior given a heat source function $f(x, y)$. Figure 4.2 shows an example of the 2D plate and a heat distribution for an example f.

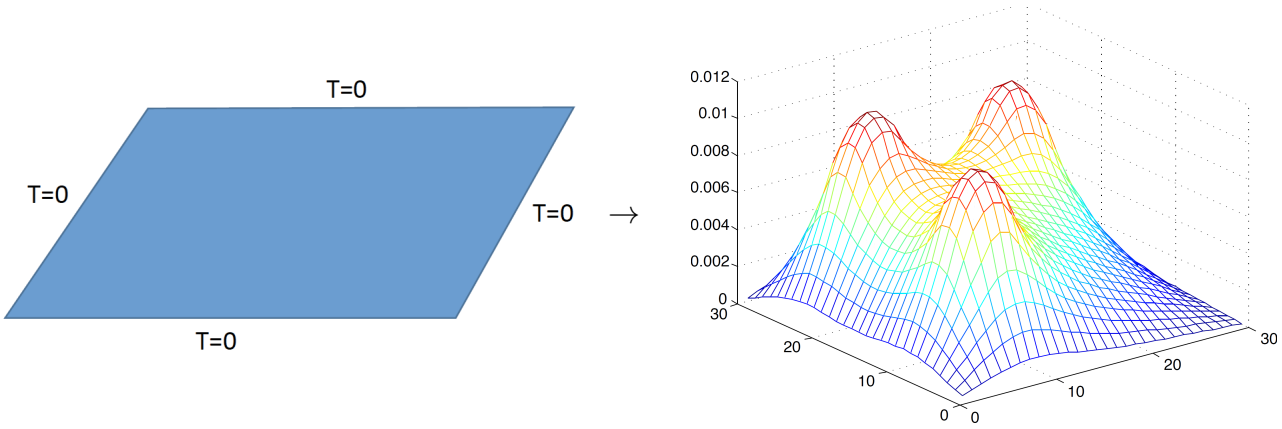


Figure 4.2: Two-dimensional plate domain (left) and heat distribution (right).

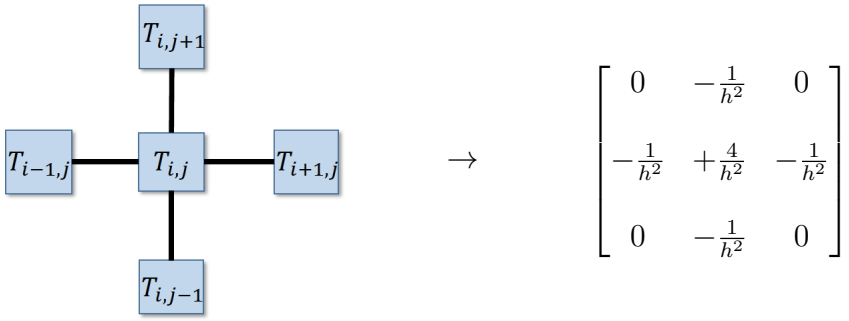
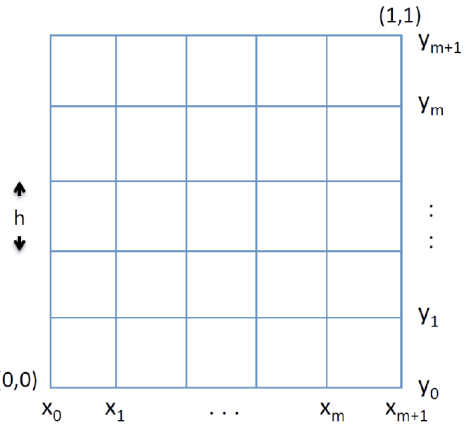


Figure 4.3: The finite difference stencil for the left hand side of (4.10), i.e., the **negative** of the 2D discrete Laplacian.

Gridpoints are now indexed by i, j so that (x_i, y_j) defines a discrete location on the 2D plate. The inset shows an example grid. The approximate temperature at (x_i, y_j) is denoted $T_{i,j}$ such that $T_{i,j} \approx T(x_i, y_j)$. We now need to approximate the 2D continuous Poisson equation

$$-\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = f,$$

at each gridpoint. The discrete Poisson equation in 2D is obtained by approximating the 2nd derivative in each axis separately, then summing them together. That is,



$$\begin{aligned} -\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) &= f \\ -\left(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{h^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{h^2}\right) &= f_{i,j} \\ \frac{4T_{i,j} - T_{i-1,j} - T_{i+1,j} - T_{i,j-1} - T_{i,j+1}}{h^2} &= f_{i,j}. \end{aligned} \tag{4.10}$$

The finite difference **stencil** is a convenient visual notation for (4.10) centered at each gridpoint (see Figure 4.3). The nonzeros in the stencil will be the nonzeros in a row of the matrix.

To put (4.10) into matrix form we need to "flatten" the indices from 2D (i, j) to 1D (k). The 2D computational domain is indexed from 0 to $m+1$ in each dimension (see Figure 4.4 left). Since the boundary values are known to be 0, we only need to solve for unknowns $T_{i,j}$ for all $i, j \in [1, m]$ (a total of m^2 unknowns).

How do we index the unknowns into a 1D array? A natural rowwise ordering numbers gridpoints along the x-axis first, then along the y-axis (see Figure 4.4 right).

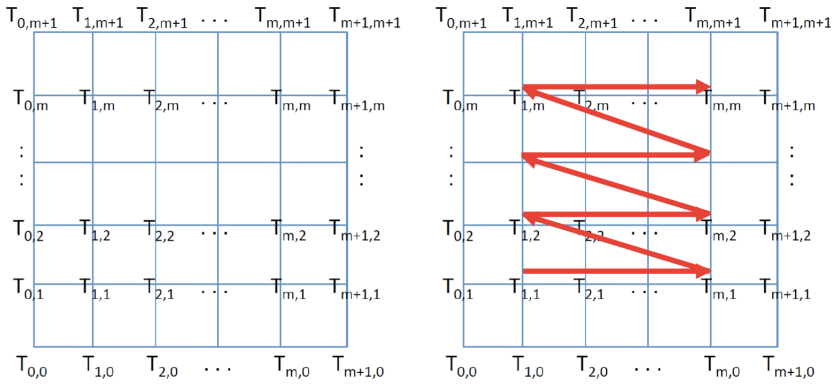


Figure 4.4: Two-dimensional indexing (left) for a discrete plate and a possible 1D ordering/flattening (right).

Specifically,

$$\begin{aligned} T_{1,1} &\to T_1, \\ T_{2,1} &\to T_2, \\ &\vdots \\ T_{m,1} &\to T_m, \\ T_{1,2} &\to T_{m+1}, \\ T_{2,2} &\to T_{m+2}, \\ &\vdots \\ T_{m,m} &\to T_{m^2}. \end{aligned}$$

That is, we convert from 2D indices (i, j) to 1D indices k using $k = i + (j - 1) \times m$.

The general form of the Laplacian matrix in 2D with this natural rowwise ordering is given in Figure 4.5. Notice that their are 5 bands:

- 1 diagonal band,
- 2 bands immediately above/below the diagonal,
- 2 bands separated horizontally by m entries.

Explanation:

- Let (i, j) be arbitrary, $1 \le i \le m, 1 \le j \le m$.
- Consider the equation

$$\frac{1}{h^2}\left(4T_{i,j} - T_{i-1,j} - T_{i+1,j} - T_{i,j-1} - T_{i,j+1}\right) = f_{i,j}.$$

- The 4 coefficient appears on the diagonal, since i, j on the LHS agrees with i, j on the RHS.

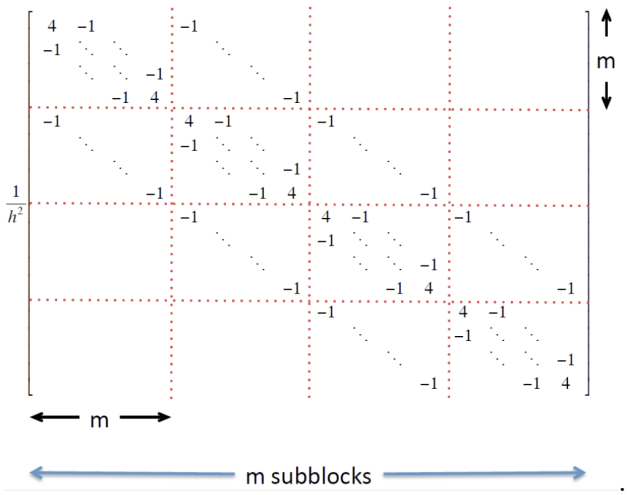


Figure 4.5: General matrix structure for discrete Laplacian with natural rowwise ordering.

- The $-T_{i-1,j}$ term contributes
  - nothing, if $i = 1$, by boundary conditions, and
  - -1 immediately to the left of the diagonal, otherwise.
- The $-T_{i+1,j}$ term contributes
  - nothing, if $i = m$, by boundary conditions, and
  - -1 immediately to the right of the diagonal, otherwise.
- The $-T_{i,j-1}$ term contributes
  - nothing, if $j = 1$, by boundary conditions, and
  - -1, m positions to the left of the diagonal, otherwise.
- The $-T_{i,j+1}$ term contributes
  - nothing, if $j = m$, by boundary conditions, and
  - -1, m positions to the right of the diagonal, otherwise.

Remarks:

1. The 2-D setup quickly becomes more complicated than the 1-D setup.
2. Adding more dimensions would further increase the complexity.
3. If we relax our assumption about not allowing changes in temperature over time, then we would need to add a time dimension. E.g. adding a time dimension to the 1-D setup would make it 2-D to start. We would have to be careful to clearly define our boundary conditions in this case.

Other types of PDE problems, discretizations, and geometries give rise to different matrix structures and properties. For example, a triangular mesh can be used with a finite volume discretization to study the flow around an airfoil (see Figure 4.6).

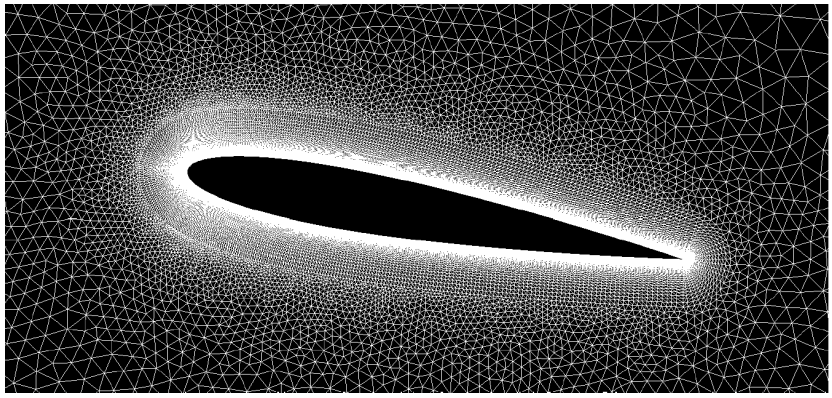


Figure 4.6: Example discretization using triangles for an airfoil.

This lecture only considered modelling heat in an equilibrium using the Poisson equation. The time-dependent heat equation considers non-equilibrium situations, i.e., how temperature evolves over time. The finite difference equations are similar and lead to another linear system to solve.

5 Lecture 05: Graph Structure of Matrices; Matrix Re-Ordering

Outline

1. Graph Structure of Matrices
   (a) Graph Structure
   (b) Fill-in During Factorization
2. Matrix Re-Ordering
   (a) Key Idea

5.1 Graph Structure of Matrices

This lecture considers the graph representation of (symmetric) matrices. We will discuss the effect of factorization with respect to fill-in and the graph itself. Remember that fill-in during factorizations increases storage and flop costs, so lower is better! Common matrix reordering methods that reduce fill-in will be discussed in following lectures.

5.1.1 Graph Structure

Given a square matrix A we can create a directed graph $G(A)$. The graph has one node i for each row i in A and an edge connecting $i \to j$ if the matrix entry $a_{ij} \neq 0$. If A is symmetric we can take an undirected graph, i.e., edges $i \leftrightarrow j$. Definition 5.1 gives a more formal definition of the graph structure from a matrix.

Definition 5.1. *Let $A \in \mathbb{R}^{n \times n}$. We associate A with a (directed) graph $G = (V, E)$ that has n nodes: one node per row of A. When $i \neq j$, an edge $(i, j) \in E$ connects nodes i and j iff $a_{ij} \neq 0$. Mathematically,*

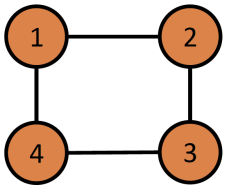
$$i \in V, \forall 1 \leq i \leq n \text{ and } \forall i \neq j, (i, j) \in E \text{ iff } a_{ij} \neq 0.$$

Remarks:

1. This is most useful when A is sparse.
2. Note that we exclude drawing self-cycles (i.e. when $i = j$) in the graph, even though $a_{ii} \neq 0$ is possible.
3. We will consider mostly undirected (symmetric) graphs in this course, unless otherwise noted.

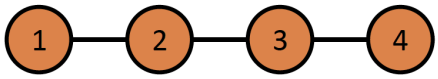
An example graph from a matrix is give below:

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

$\rightarrow G(A) =$ 

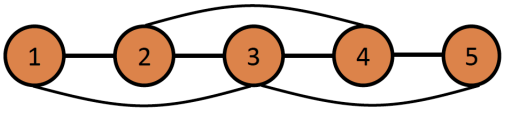
The graph structure often has a physical/geometric interpretation. For example, the Laplacian matrix from the Poisson equation recovers the underlying grid structure. For the 1D Laplacian matrix, which is tidiagonal, we obtain a graph consists of nodes connected consecutively:

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

$\rightarrow G(A) =$ 

Another 1D example is shown below. Exercise: what shape of finite difference "stencil" would produce the graph below?

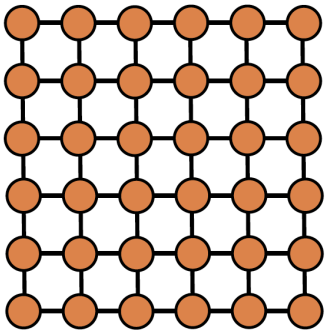
$$A = \begin{bmatrix} \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \end{bmatrix}$$

$\rightarrow G(A) =$ 

The graph of the 2D Laplacian matrix also recovers the original grid structure. With

$$A = \begin{bmatrix}
4 & -1 & & & -1 & & & & & & & & & & \\
-1 & \ddots & \ddots & & & \ddots & & & & & & & & & \\
 & \ddots & \ddots & -1 & & & \ddots & & & & & & & & \\
 & & -1 & 4 & & & & -1 & & & & & & & \\
-1 & & & & 4 & -1 & & & -1 & & & & & & \\
 & \ddots & & & -1 & \ddots & \ddots & & & \ddots & & & & & \\
 & & \ddots & & & \ddots & \ddots & -1 & & & \ddots & & & & \\
 & & & -1 & & & -1 & 4 & & & & -1 & & & \\
 & & & & -1 & & & & 4 & -1 & & & -1 & & \\
 & & & & & \ddots & & & -1 & \ddots & \ddots & & & \ddots & \\
 & & & & & & \ddots & & & \ddots & \ddots & -1 & & & \ddots \\
 & & & & & & & -1 & & & -1 & 4 & & & -1 \\
 & & & & & & & & -1 & & & & 4 & -1 & \\
 & & & & & & & & & \ddots & & & -1 & \ddots & \ddots \\
 & & & & & & & & & & \ddots & & & \ddots & \ddots \\
 & & & & & & & & & & & -1 & & & -1 \\
 & & & & & & & & & & & & & -1 & 4
\end{bmatrix},$$

we have

$G(A) =$ 

Explanation:

1. By symmetry, we can analyze the rows and know that the columns will behave the same way.
2. There are m^2 rows. Hence the graph has m^2 nodes, i.e. it is a grid with m nodes on each of its rows and columns.
3. For the block of the first m rows, and the block of the last m rows (2 such blocks):
   (a) 2 rows with 2 off-diagonal non-zero entries - these are the "corner" nodes of the graph.
   (b) $m-2$ rows with 3 off-diagonal non-zero entries - these are "outside, non-corner" nodes of the graph.
4. For each of the inner blocks of m rows ($m-2$ such blocks):

(a) 2 rows with 3 off-diagonal non-zero entries - these are "outside, non-corner" nodes of the graph.
(b) $m-2$ rows with 4 off-diagonal non-zero entries - these are "inside" nodes of the graph.

5. From here, it is just a matter of mapping the rows of the matrix to the nodes of the graph and checking that the structure is as described.
6. It is an exercise to verify that this numbering of the graph vertices agrees with the structure decribed:

$(m-1)m+1$	$(m-1)m+2$	$\cdots$	$(m-1)m+(m-1)$	m^2
$\vdots$				$\vdots$
$m+1$	$m+2$	$\cdots$	$2m-1$	$2m$
1	2	$\cdots$	$m-1$	m

Q & A

1. What if the matrix A is not symmetric?
   A: We must write the graph in a directed way.
2. Can we still write the graph directed, even of A is symmetric?
   A: Yes! In this case, each edge has an arrowhead on both ends.

5.1.2 Fill-in During Factorization

Recall that factorization can destroy the nice sparsity pattern of a matrix. For example, consider the LU factorization of this arrow matrix

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

$$= \underbrace{\begin{bmatrix} 1 & & & & \\ \times & 1 & & & \\ \times & \times & 1 & & \\ \times & \times & \times & 1 & \\ \times & \times & \times & \times & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix}}_{U}.$$

In this section we consider the relationship between factorization of sparse matrices, fill-in, and the graph structure.

Suppose we want to compute the Cholesky factorization of the matrix

$$A = \begin{bmatrix} \times & \times & & \times \\ \times & \times & \times & \\ & \times & \times & \times \\ \times & & \times & \times \end{bmatrix}. \tag{5.11}$$

Fill-in occurs if we compute the Cholesky factorization of A in (5.11). For Cholesky factorization we have

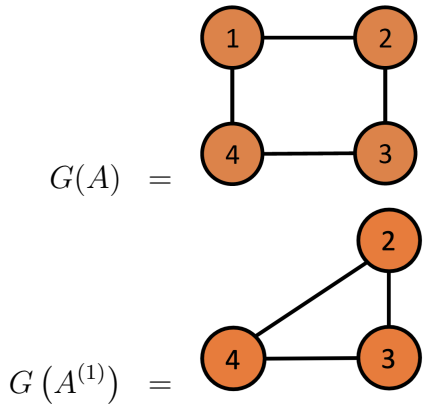
$$v = \begin{bmatrix} \times \\ 0 \\ \times \end{bmatrix}, \qquad \Rightarrow \quad \frac{vv^T}{\alpha} = \begin{bmatrix} \times & 0 & \times \\ 0 & 0 & 0 \\ \times & 0 & \times \end{bmatrix},$$

$$\Rightarrow \quad B - \frac{vv^T}{\alpha} = \begin{bmatrix} \times & \times & \\ \times & \times & \times \\ & \times & \times \end{bmatrix} - \begin{bmatrix} \times & 0 & \times \\ 0 & 0 & 0 \\ \times & 0 & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}.$$

Therefore, non-zero entries are introduced in the same places as with LU factorization. This is because we are deleting the same node $i = (1)$, which causes nodes (2) and (4) to connect with an edge.

For the 1st iteration, we add a multiple of the 1st row to the rows below it. This reduces $a_{2:n,1}$ to zeros, indicated in green. While introducing the zeros in $a_{2:n,1}$ we unfortunately introduce some new nonzeros (indicated in blue, called "fill-in"). Note that some nonzero entries may also become zero, but we do not take these into account. These new zeros will depend on the actual values in A, which we would like to abstract away for greater generality.

Now consider what happens to the graph structure of (5.11) after one step of Cholesky factorization. The new graph deletes node (1) and connects nodes (2) and (4) together.



However, nodes (2) and (4) were not connected before, which corresponds to the fill-in. In general, elimination of node i yields a new graph with:

1. Node i and all its edges deleted,
2. New edges $j \leftrightarrow k$ added if there were edges (j, i) and (i, k), i.e., new edges between all node pairs connected to i in the old graph (corresponds to fill-in!).

Convention: Don't display diagonal entries, if they are not connected to anything else. A complete graph would display node #1, with no edges connecting it to anything else.

Exercise: what are the graphs of the LU-factorizations of the arrow matrices below? From the graph, can you see why A_1 produces dense LU factors, while A_2 suffers no fill-in?

$$A_1 = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{bmatrix}$$

$$A_2 = \begin{bmatrix} \times & & & & \times \\ & \times & & & \times \\ & & \times & & \times \\ & & & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

5.2 Matrix Reordering

Earlier, we saw that general sparse matrices may have **dense LU factors**, e.g.,

$$\begin{aligned} A &= \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{bmatrix} \\ &= \begin{bmatrix} 1 & & & & \\ \times & 1 & & & \\ \times & \times & 1 & & \\ \times & \times & \times & 1 & \\ \times & \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix} \\ &= LU. \end{aligned}$$

We will now start discussing matrix reorderings, which can produce LU factors without fill-in. For example, reordering the same arrow matrix above can give

$$\begin{aligned} A &= \begin{bmatrix} \times & & & & \times \\ & \times & & & \times \\ & & \times & & \times \\ & & & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \\ &= \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ \times & \times & \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & & & & \times \\ & \times & & & \times \\ & & \times & & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix} \\ &= LU. \end{aligned}$$

5.2.1 Key Idea

We saw earlier that the graph structure of a matrix expresses the underlying relationships among variables. The ordering (numbering) of the nodes/variables impacts the matrix layout, but **not** its graph or the solution. The graph structure of a symmetric matrix is clearly unchanged by just renumbering its nodes. However, different matrices with the same graph can suffer vastly different levels of fill-in during factorization.

Goal of matrix reordering: Renumber the graph nodes to produce a matrix that minimizes fill-in during factorization.

Reordering a matrix can be written mathematically in terms of **permutation matrices**.

A permutation matrix is the identity matrix I with (some) rows/columns swapped.

1. Permuting the rows is equivalent to multiplying A by a permutation matrix P on the left: PA. For example,

$$\begin{bmatrix} & 1 & \\ & & 1 \\ 1 & & \end{bmatrix} \begin{bmatrix} 3 & 2 & 5 \\ 2 & 4 & 1 \\ 5 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 1 \\ 5 & 1 & 3 \\ 3 & 2 & 5 \end{bmatrix}.$$

   Note that this multiplication is only conceptual. In implementations one never multiplies or stores permutation matrices explicitly.
2. Similarly, permuting the columns is equivalent as multiplying by a permutation matrix Q on the right: AQ. For example,

$$\begin{bmatrix} 3 & 2 & 5 \\ 2 & 4 & 1 \\ 5 & 1 & 3 \end{bmatrix} \begin{bmatrix} & 1 & \\ & & 1 \\ 1 & & \end{bmatrix} = \begin{bmatrix} 5 & 3 & 2 \\ 1 & 2 & 4 \\ 3 & 5 & 1 \end{bmatrix}.$$

3. Of course, we can permute the rows and the columns simultaneously: PAQ.

A Nice Fact About Any Permutation Matrix, Q: $QQ^T = I$.

The effect of permutation matrices on solving linear systems is as follows. Suppose we are interested in solving the linear system

$$Ax = b.$$

Key Question: How can we correctly keep track of permutations of the rows/columns of A?

- If we permute A to $\tilde{A} = PAQ$, then we need to reorder entries of x and b to match the changes applied to A. Hence

$$
\begin{aligned}
Ax &= b && (5.12)\\
A(QQ^T)x &= b && (5.13)\\
AQ(Q^Tx) &= b && (5.14)\\
PAQ(Q^Tx) &= Pb && (5.15)\\
\tilde{A}\tilde{x} &= \tilde{b} && (5.16)
\end{aligned}
$$

where $\tilde{x} = Q^Tx$ and $\tilde{b} = Pb$.

- After solving (5.16) for $\tilde{x}$, we can recover the original solution $x = Q\tilde{x}$.
- We are "unpermuting" $\tilde{x}$ to recover x.

Q & A

1. Why is Q needed?
   A: To permute the columns of A, if needed.
2. Can we have $P = I$?
   A: Yes, if no row permutations are required for A. The setup simplifies considerably in this case.
3. In the 3×3 case, can P swap two rows, leaving the 3^{rd} row untouched?
   A: Yes!
4. Is I a permutation matrix?
   A: Yes! It's the matrix of the identity permutation.

6 Lecture 06: Matrix Re-Ordering

Outline

1. Matrix Re-Ordering
   (a) Key Idea
   (b) Example with Natural Ordering
   (c) Envelope Reordering
   (d) Level sets
   (e) Cuthill-McKee

6.1 Matrix Reordering

6.1.1 Key Idea

Symmetric Permutations

- The special case of symmetric permutation is of particular importance.
- A symmetric permutation is when we replace A with PAP^T, i.e., we permute the rows and columns in the same way ($Q = P^T$).
- Symmetric permutations naturally preserve symmetry for symmetric matrices (more generally when the sparsity pattern is symmetric).

Q: What does a symmetric permutation do to the graph of A?

A: The structure will be unchanged; the nodes will be re-numbered.

Given a particular reordering, what is P for a symmetric permutation? Reordering gives a list of before/after labels, e.g.,

$$\begin{aligned} 1 &\to 2, \\ 2 &\to 3, \\ 3 &\to 1, \\ 4 &\to 4. \end{aligned}$$

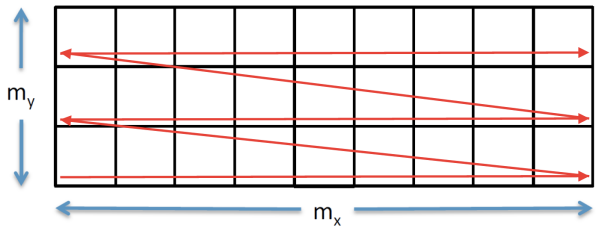
This says which old row moves to which new row. We apply the desired row swaps to the identity matrix I to get the permutation P. In the example above,

$$I = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix} \quad \Longrightarrow \quad P = \begin{bmatrix} & 1 & & \\ 1 & & & \\ & & 1 & \\ & & & 1 \end{bmatrix}.$$

6.1.2 Example with Natural Ordering

Idea: We saw earlier that bandwidths are preserved when we factorize A. Thus, when we have some choice of how to write A, making a choice which affords **minimum bandwidth** is desirable.

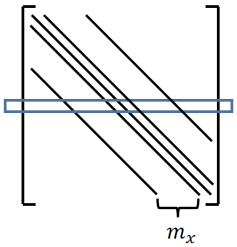
Consider the 2D Laplacian matrix on a non square domain, with $m_x \gg m_y$, and natural rowwise ordering. This domain and rowwise ordering (first along x-axis, then y-axis) is depicted below.



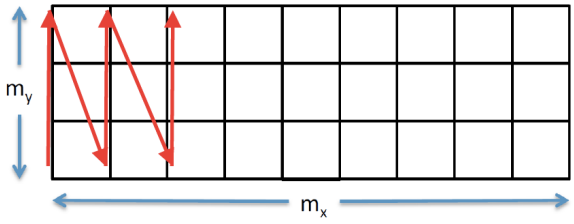
What is the bandwidth of the 2D Laplacian matrix with natural rowwise ordering? Consider row i, which has entries at columns

- i (diagonal),
- $i-1, i+1$ (inner bands),
- $i-m_x, i+m_x$ (outer bands).

Hence, the bandwidth is m_x.



Consider instead columnwise ordering along y-axis first, and then x-axis. In this case, the bandwidth becomes m_y instead!

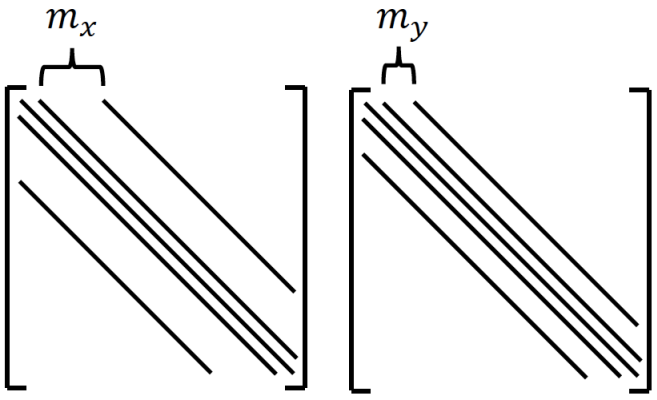


Consider row j, which has entries at columns

- j (diagonal),
- $j-1, j+1$ (inner bands),
- $j-m_y, j+m_y$ (outer bands).

Hence, the bandwidth is m_y.

Ordering along the y-axis first gives narrower bandwidth (in this case) since $m_x \gg m_y$.



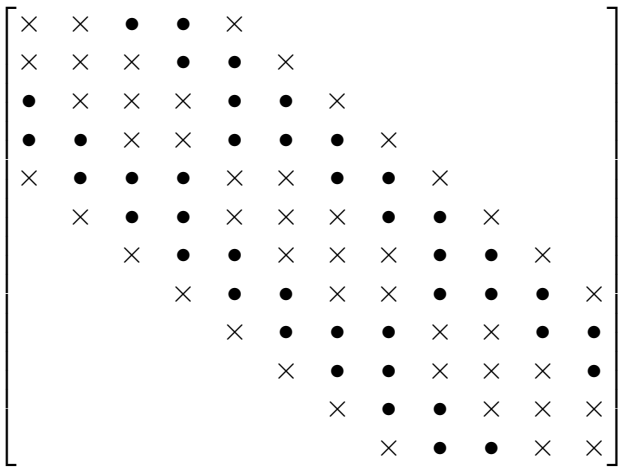


Figure 6.7: Example of where fill can occur for banded matrices. The possible fill occurs where • are depicted.

The columnwise ordering of the 2D Laplacian matrix produces less fill.

Recall when factoring banded matrices the L and U factors of a band matrix have the same lower and upper bandwidths, respectively, as the input A. That is, the widths of the bands are preserved. Therefore, fill can only occur between the outermost bands (e.g., see Figure 6.7).

- One can verify that flops(banded GE) $\approx O(npq)$ for bandwidths p, q and n gridpoints.
- Therefore, the cost is $O(m^2 n)$ for bandwidth m.
- For the rowwise ordering we have flops $= O(m_x^2 n)$, whereas for the columnwise ordering we only have flops $= O(m_y^2 n)$.
- But what can we do for more general sparsity patterns?
  - Finding the true optimum ordering of the graph is NP complete (i.e. it is hard).
  - There do exist many ordering algorithms based on good heuristics.
- We will look at some common algorithms next:
  - Envelope/Level set methods,
  - (Reverse) Cuthill McKee,
  - Markowitz,
  - Minimum Degree.

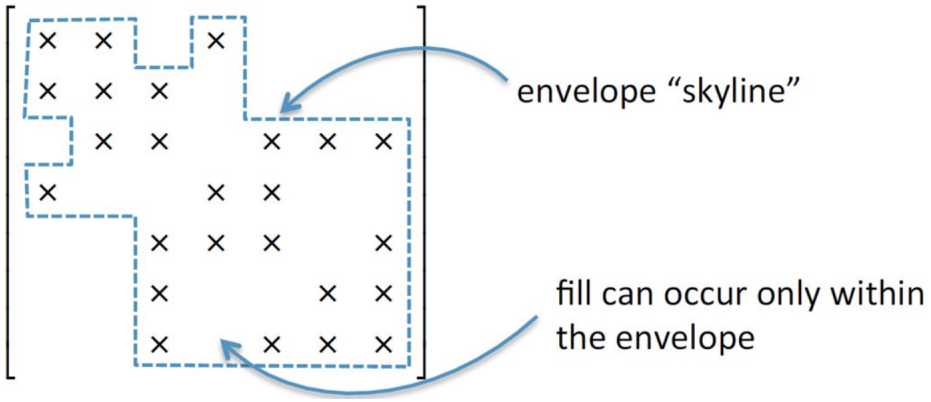
Q & A

1. Last week we discussed upper- and lower-bandwidths. What does "bandwidth" mean, unqualified?
   A: If upper and lower bandwidths are equal, then writing "bandwidth", unqualified, is clear enough.

6.1.3 Envelope Reordering

- In this lecture we will look at the first two types of methods.
- The next lecture will discuss Markowitz and Minimum degree reorderings.

- In practice, bandwidth may vary a lot between individual rows.



The **envelope** is the contiguous part of the matrix containing all non-zero entries, indicated by dotted lines in the picture.

Note, we are not asserting that all entries inside the envelope are non-zero; we are asserting that all entries outside the envelope are zero.

Remarks:

1. Recall that fill-in during factorization is bad, hence it is to be minimized.
2. In each row of L, fill can only occur between 1st non-zero entry (from the left) and the diagonal entry.
3. This motivates us to limit fill by keeping the envelope as close to the diagonal as possible.
4. This further motivates the next section, on **Level Sets**.

Q & A:

1. Does our matrix have to be symmetric?
   A: All graphs coming up are undirected, which requires A to be symmetric. So although our setup does not require A to be symmetric, we will demand that A be symmetric to agree with the convention in the notes.
2. Can the envelope sit both above and below the diagonal?
   A: Yes, as in the above picture.

Q: What does this imply about a **good** numbering of nodes?

A: The graph neighbours should have numbers as close together as possible.

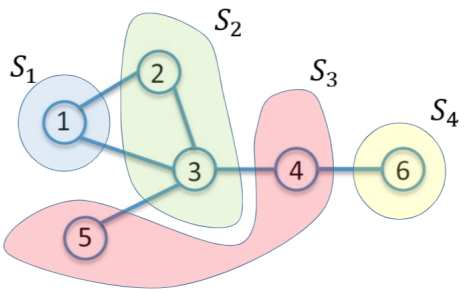
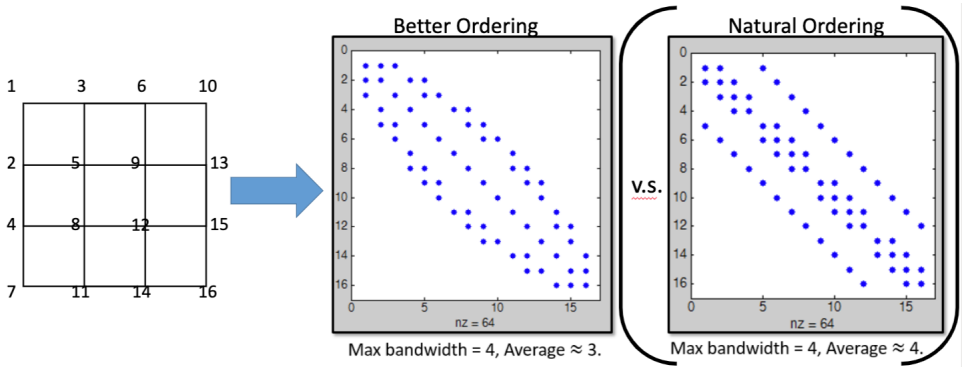


Figure 6.8: Level sets of an example graph.



6.1.4 Level sets

We assume the sparsity pattern of the matrix is symmetric, i.e., the underlying graph representation is undirected. Envelope methods are based on **graph level sets** S_i defined below.

Definition 6.1. *Let A be a symmetric matrix. Let $\{s_j\}$ be the nodes of the graph of A. Then the **level sets** S_i are the sets of nodes that are the same graph distance from some starting point. That is,*

$$
\begin{aligned}
S_1 &= \{\textit{the single starting node}\},\\
S_2 &= \{\textit{all immediate neighbours of the node in } S_1\},\\
S_3 &= \{\textit{all immediate neighbours of nodes in } S_2, \textit{not in } S_1 \textit{ or } S_2\},\\
&\vdots\\
S_i &= \{\textit{all immediate neighbours of nodes in } S_{i-1}, \textit{not in } S_1, S_2, \ldots, S_{i-1}\}.
\end{aligned}
$$

Figure 6.8 shows an example of the level sets of a graph.

Q: Why do we care about level sets?

A: Envelope methods:

1. order the nodes in $S_2, S_3, \ldots, S_k$, and

2. "do something" with the ordered list of nodes.

- Envelope methods order nodes in S_2, then nodes in S_3, and so on.
- This is similar to a breadth first traversal (BFS).

6.1.5 Cuthill-McKee

How do we order nodes **within** each level set? In the **Cuthill-McKee (CM)** algorithm a heuristic is used based on the **degree** of a node.

Definition 6.2. *The degree of a node v, denoted* $\deg(v)$*, is the number of adjacent nodes (i.e. the number of incident edges).*

For example, in the graph in Figure 6.8 we have $\deg(3) = 4$ and $\deg(5) = 1$.

The Cuthill-McKee ordering heuristic is as follows. When visiting a node during traversal, order its neighbors (yet to be visited) in increasing order of degree and add them to the queue in this order. Cuthill-McKee algorithm is given in Algorithm 6.3, which consists of the following:

1. pick an arbitrary starting node and number it 1,
2. find all un-numbered neighbours of node 1 and number them in increasing order of degree,
3. for each of node 1's neighbours, order their neighbours in increasing order of degree,
4. continue recursively until all nodes have been numbered.

Ties are broken in an arbitrary manner, e.g., based on the initial node ordering.

Remarks:

1. This assumes that the graph is connected.
2. If the graph is not connected, then the corresponding matrix (possibly under some permutation of rows/columns) can be decomposed into diagonal blocks, each of which corresponds with a connected subgraph.
3. Then we could solve each subsystem independently of the others.
4. Thus we lose no generality by assuming that our graph is connected.

Algorithm 6.3 : Cuthill-McKee Ordering

Input: undirected graph $G = (V, E)$
Output: ordered level sets S_i
choose starting node s ▷ for each connected component
$S_1 \leftarrow \{s\}$, mark s
$i = 1$
while $S_i \neq \emptyset$
    $S_{i+1} \leftarrow \emptyset$
    for each $u \in S_i$ ▷ in order of increasing degree
        for each unmarked v adjacent to u ▷ in order of increasing degree
            $S_{i+1} \leftarrow S_{i+1} \cup \{v\}$
            mark v
        end for
    end for
    $i = i + 1$ ▷ move on to the next level set
end while

Cuthill-McKee - Q & A

1. **Q:** What is a good choice of starting node for CM?
   A: A vertex with as high a degree as possible.

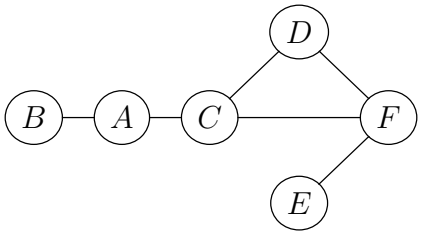
Reverse Cuthill-McKee (RCM)

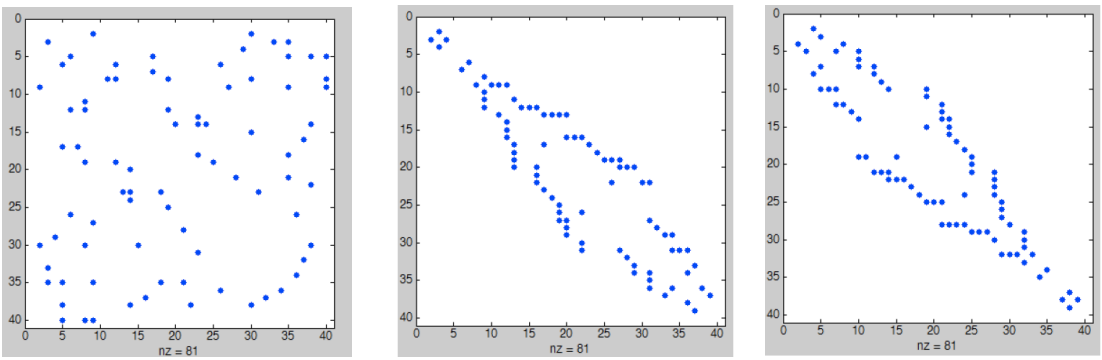
- The **reverse Cuthill-McKee (RCM)** algorithm is exactly what it sounds like!
- You compute the CM numbering then reverse it, i.e.,

$$\text{node}_i^{\text{RCM}} = \text{node}_{n-i+1}^{\text{CM}} \quad \text{for} \quad i = 1, \ldots, n.$$

- This is simple, but why perform the reversal? John Alan George ("Computer Implementation of the finite element method", 1971) observed by simply reversing the Cuthill-McKee ordering, we can reduce the amount of fill-in for many graphs (matrices).
- The RCM has the same envelope of CM but better observed behavior in practice.
- The patterns produced by RCM are more like the low fill downward arrow matrix, rather than the upward arrow.
- Figure 6.9 shows an example on a random symmetric matrix.
- As expected the CM algorithm produces a matrix with a smaller bandwidth.
- The RCM algorithm has the same bandwidth but is more "down-arrow" like.

Example: In this example, we will compute the CM ordering, and the RCM ordering, for this graph:





Random Symmetric Matrix Cuthill-McKee Reverse Cuthill-McKee

Figure 6.9: Comparison of CM (middle) vs RCM (right) for a symmetric matrix (left).

Assumptions:

1. Select node A as your starting node.
2. Break any ties with respect to the degrees of nodes in a level set according to the usual alphabetic order of the nodes.

Computing the CM and RCM Orderings:

1. Start by numbering:
   1. A
2. Number the un-numbered neighbours of node A in ascending order of their degrees:
   2. B (degree 1)
   3. C (degree 3)
3. B has no un-numbered neighbours. Number the un-numbered neighbours of node C in ascending order of their degrees:
   4. D (degree 2)
   5. F (degree 3)
4. D has no un-numbered neighbours. Number the un-numbered neighbours of node F in ascending order of their degrees:
   6. E (degree 1)
5. This yields the Cuthill-Mckee ordering:

1	2	3	4	5	6
A	B	C	D	F	E

6. This yields the Reverse Cuthill-Mckee ordering:

1	2	3	4	5	6
E	F	D	C	B	A

Detailed Explanation of Fig 6.10:

1. Start by numbering:
   1. A
2. Number the un-numbered neighbours of node A in ascending order of their degrees:
   2. G (degree 6)
3. Number the un-numbered neighbours of node G in ascending order of their degrees:

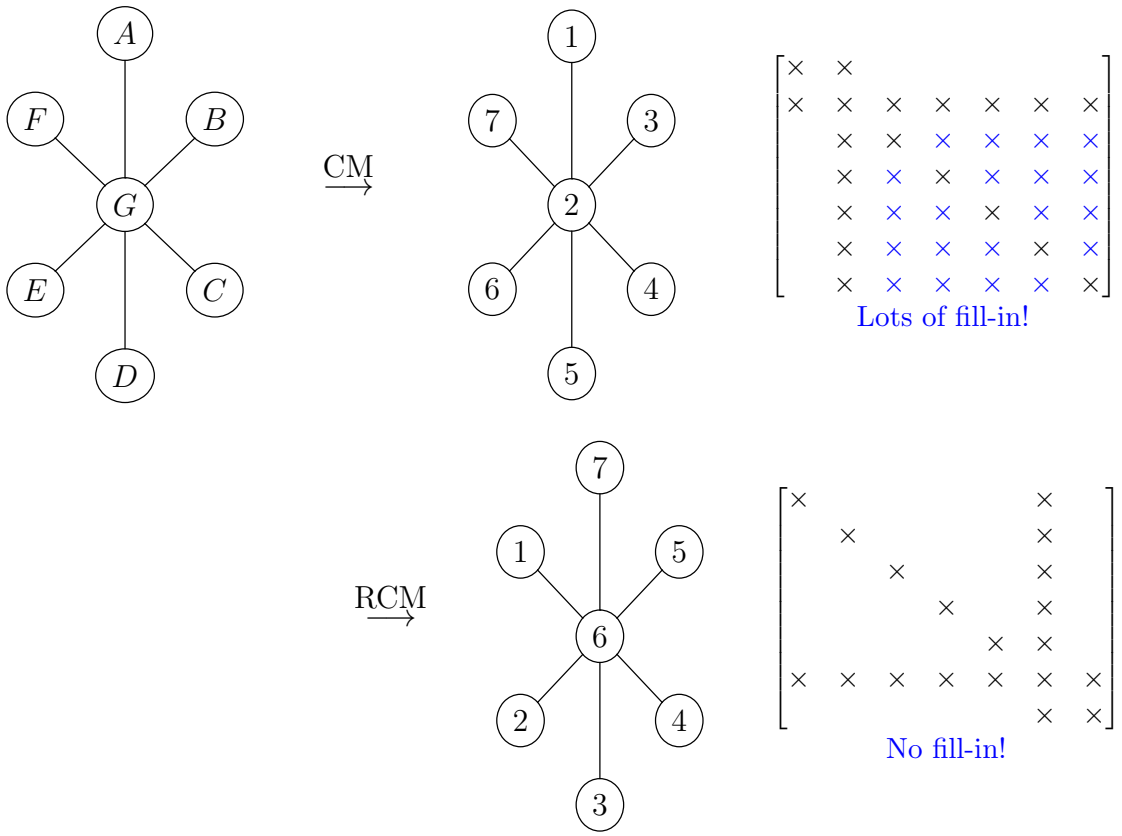


Figure 6.10: CM and RCM orderings for the initial graph (top left). The CM algorithm is started at node A and ties are broken alphabetically.

3. B (degree 1)
4. C (degree 1)
5. D (degree 1)
6. E (degree 1)
7. F (degree 1)

4. This yields the Cuthill-Mckee ordering:

1	2	3	4	5	6	7
A	G	B	C	D	E	F

5. This yields the Reverse Cuthill-Mckee ordering:

1	2	3	4	5	6	7
F	E	D	C	B	G	A

Theorem 6.1. (RCM is optimal on trees.) *On a tree, no matter which node we start with, RCM ordering produces no fill.*

Proof. We argue that the first node in the RCM ordering has to be a leaf (degree 1).

- Towards a contradiction, suppose the starting node is s and the first node in RCM is a non-leaf node u.
- Then u is connected to at least two nodes v and w.
- In constructing the CM ordering we may have reached (from s) at most one of v and w before reaching u (otherwise we have a cycle $v \cdots s \cdots wuv$, which can not happen for a tree).
- But then after we have reached u we need to continue the CM ordering to the other (or both) child node, so u cannot be the last node in CM (i.e. u cannot be the first note in the RCM ordering).
- This is a contradiction, so u must be a leaf node.
- Now note that if we remove the node u, and rerun CM, we would get the same ordering (without u).
- Thus, recursively, we see that in the elimination of the graph by following RCM, we are always removing leaf nodes hence will not introduce any new edges.

□

- Note that Theorem 6.1 does not hold in general for the CM ordering.
- Further, RCM does not necessarily produce the optimal ordering for general graphs.
- Determining the optimal ordering that introduces the least amount of fill-in is NP-complete [Yannakakis 1981].
- The example in Figure 6.11 shows the optimality of RCM on a small tree.

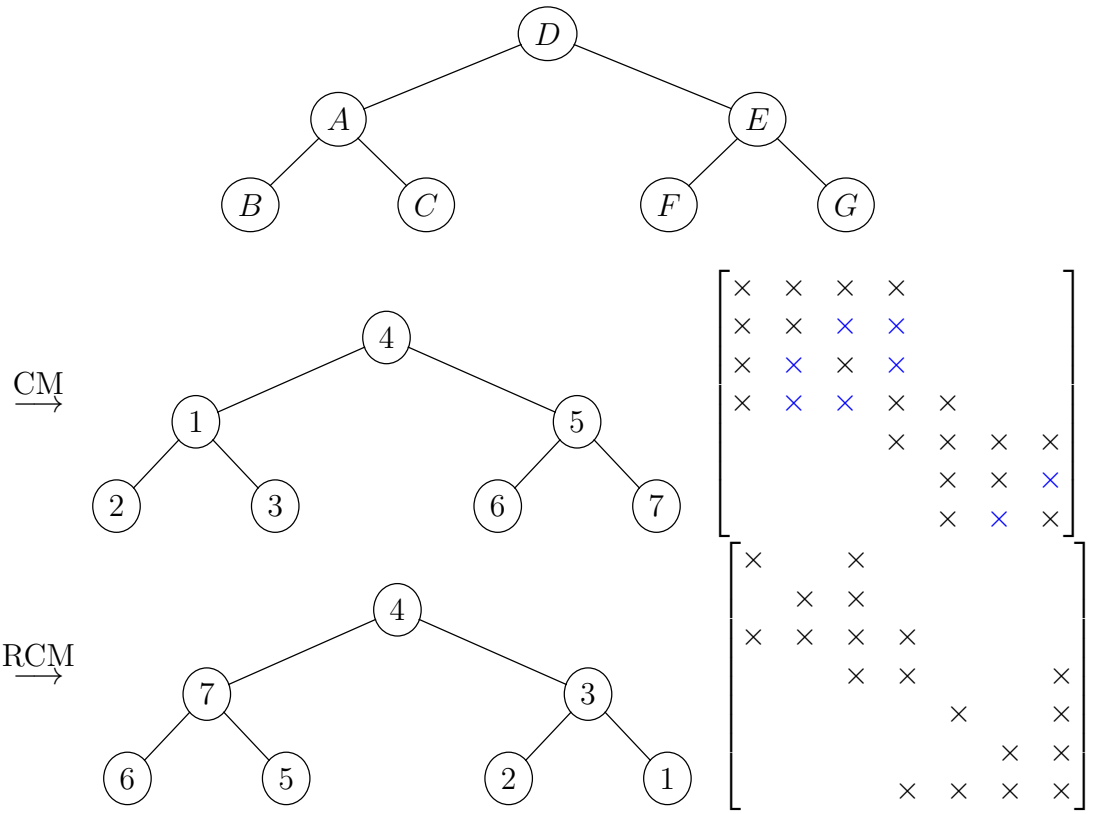


Figure 6.11: CM and RCM ordering for a tree graph. Notice that RCM produces a reordering with no fill (shown as ×).

7 Lecture 07: Matrix Re-ordering; Image De-Noising

Outline

1. Matrix Re-ordering
   (a) Markowitz Reordering
   (b) Minimum Degree Reordering
   (c) Stability (Optional)
   (d) Pivoting (Optional)
      i. Unnecessary Pivoting
2. Image De-Noising
   (a) Inverse Problems
   (b) Regularization Models
      i. Tikhonov Regularization
      ii. Laplacian Regularization
      iii. Total Variation Regularization

7.1 Matrix Re-ordering

Q & A:

1. Does Minimum Degree Re-Ordering have a problem, if the graph contains cycles?
   A: No. The algorithm still works, with enough care.
2. Is it correct that a tree is a graph without cycles?
   A: Yes!
3. Is it correct that the weak version of diagonal dominance uses $\geq$, while the usual version uses $>$?
   A: Yes.

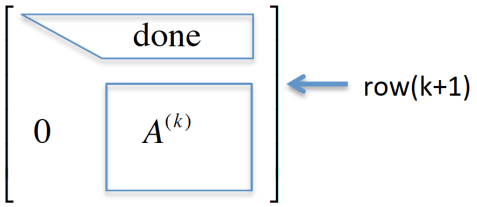
We have seen so far that a graph provides a useful abstraction of the structure of symmetric matrices. The graph offers insight into where fill-in can occur during factorization. In the previous lecture we discussed envelope methods, such as (reverse) Cuthill-McKee, which are a family of reorderings that can reduce fill by minimizing the envelope.

- Envelope/Level set methods,
- (Reverse) Cuthill McKee,
- Markowitz,
- Minimum Degree.

In this lecture we will look at the last two reordering schemes above.

7.1.1 Markowitz Reordering

Markowitz [Markowitz 1957] is a local rule that tries to (approximately) minimize fill on the current step only. In other words, it greedily minimizes fill-in for the current step. After k steps of LU factorization we have:



where $A^{(k)}$ indicates the lower right block matrix after k steps of LU. Consider the fill that can occur during one step of LU factorization from this point. Normally, we subtract multiples of current row $(k+1)$ from rows below $(k+2, \ldots, n)$, if there is a non-zero in the column to be zeroed out.

Specially, fill-in does not occur in rows that already have a zero in the column, e.g.,

$$A^{(k)} = \begin{bmatrix} \times & \times & & \times \\ \times & \times & & \times \\ 0 & & & \\ \times & \times & & \times \end{bmatrix}, \tag{7.17}$$

where blue $\times$ denotes fill-in that has occurred. The worst case fill-in at this step (using this pivot $\times$) is

$$\begin{aligned} & \text{4 entries} \\ = \; & \text{(2 other non-zeros in row)} \times \text{(2 other non-zeros in column)}. \end{aligned}$$

We can **swap rows and columns on the fly** to reduce the resulting fill-in. Consider all entries $a_{ij}^{(k)}$ in the lower right block $A^{(k)}$. The idea is to determine the entry that would minimize the (worst-case) fill. Then, swap it into the top-left (pivot $\times$) position of $A^{(k)}$. The row that produces the least fill on this current step is determined using the **Markowitz product**.

Let $r_i^{(k)} = \texttt{nnz}$ (number of nonzeros) in row i of $A^{(k)}$ and $c_j^{(k)} = \texttt{nnz}$ in column j of $A^{(k)}$. The maximum possible fill using $a_{ij}^{(k)}$ as the pivot is

$$(r_i^{(k)} - 1)(c_j^{(k)} - 1),$$

which is called the **Markowitz product**.

Brief Explanation: Consider the rows below the pivot row first (i.e. the count of non-zero entries in the chosen column). Zero entries below the pivot cannot cause fill-in, because we don't have to use the pivot row to eliminate a zero entry.

The Markowitz reordering algorithm swaps rows/columns to choose the pivot $a_{\ell m}^{(k)}$ that minimizes the Markowitz product, i.e.,

$$(\ell, m) = \arg\min_{k \le i,j} (r_i^{(k)} - 1)(c_j^{(k)} - 1).$$

As a concrete example take

$$A^{(k)} = \begin{bmatrix} 3 & 1 & 0 & 3 \\ 0 & 2 & 3 & 6 \\ 2 & 1 & 0 & 1 \\ 1 & 6 & 3 & 3 \end{bmatrix}.$$

The Markowitz product for $a_{11}^{(k)}$ is 4, but only 2 for $a_{23}^{(k)}$.

Exercise: compare $r_i^{(k)} = 1, c_j^{(k)} = 5$ with $r_i^{(k)} = 2, c_j^{(k)} = 2$, which should we prefer?

Note that the Markowitz product is just an approximation of the fill. It estimates the worst case fill since generally some of the entries may already be non-zero. Therefore, no new non-zeros would be created, e.g., in (7.17) the blue $\times$ may already be non-zero.

If A is symmetric, then we select $a_{\ell\ell}^{(k)}$ with

$$\ell = \arg\min_{k \le i} (r_i^{(k)} - 1),$$

since $\arg\min_{k \le i} r_i^{(k)} = \arg\min_{k \le j} c_j^{(k)}$. So we only need to consider diagonal entries for symmetric matrices. By symmetrically swapping both rows and columns, $a_{\ell\ell}^{(k)}$ becomes the new pivot. This approach has the following features:

- preserves symmetry and **diagonal dominance**,
- corresponds to node reordering.

Aside: A matrix is (weakly) **diagonally dominant** if for every row, the magnitude of the diagonal entry is larger than or equal to the sum of the magnitudes of all the other entries in that row. That is,

$$|a_{ii}| \ge \sum_{i \ne j} |a_{ij}|, \text{ for all } i.$$

A matrix is (strongly) **diagonally dominant** if for every row, the magnitude of the diagonal entry is strictly larger than the sum of the magnitudes of all the other entries in that row. That is,

$$|a_{ii}| > \sum_{i \ne j} |a_{ij}|, \text{ for all } i.$$

We will see more on this when we discuss **iterative methods**.

7.1.2 Minimum Degree Reordering

The symmetric case of Markowitz reordering inspires an algorithm called **minimum degree reordering**. Consider the $(r_i^{(k)} - 1)$ for diagonal entries of this symmetric matrix

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & \times & \\ \times & & \times & \times & \times \\ \times & & & \times & \times \end{bmatrix}. \tag{7.18}$$

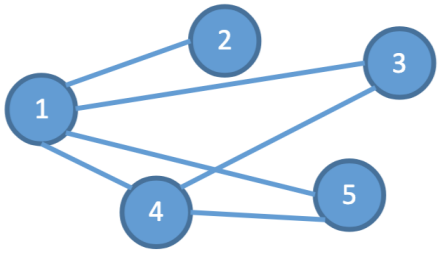
Because the matrix is symmetric, therefore we only need to consider diagonal entries. We have that

$$
\begin{aligned}
a_{11} &\mapsto 4, \\
a_{22} &\mapsto 1, \\
a_{33} &\mapsto 2, \\
a_{44} &\mapsto 3, \\
a_{55} &\mapsto 2,
\end{aligned}
$$

so we would swap to use a_{22} as the pivot.

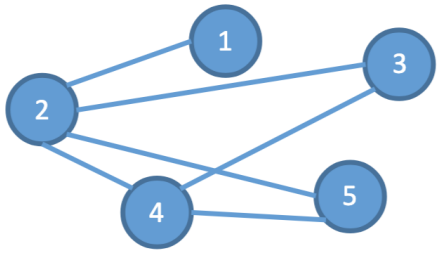
But what do these values correspond to in the graph view? The value of $(r_i^{(k)} - 1)$ is number of off-diagonal non-zero entries in the row, which is the same as the **degree** of the corresponding node! The original matrix in (7.18) gives the following graph:

$$
\begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & & & \\
\times & & \times & \times & \\
\times & & \times & \times & \times \\
\times & & & \times & \times
\end{bmatrix}
\quad \longleftrightarrow \quad
$$



While the reordering with a_{22} as the pivot gives:

$$
\begin{bmatrix}
\times & \times & & & \\
\times & \times & \times & \times & \times \\
 & \times & \times & \times & \\
 & \times & \times & \times & \times \\
 & \times & & \times & \times
\end{bmatrix}
\quad \longleftrightarrow \quad
$$



Minimum degree ordering **chooses the node with (current) minimum degree as the pivot element**, at each step of factorization.

Recall that a step of Cholesky factorization corresponds (in the graph) to

1. deleting the chosen node and all its edges, then
2. connecting its incident neighbours together with a new edge (this corresponds to fill).

A specific example of Cholesky and the resulting fill is given below. The initial matrix and corresponding graph are

$$
\begin{bmatrix}
2 & -1 & -1 \\
-1 & 2 & \\
-1 & & 2
\end{bmatrix}
\quad \longleftrightarrow \quad
$$

Matrix

Elimination Graph $G(A^{(0)})$

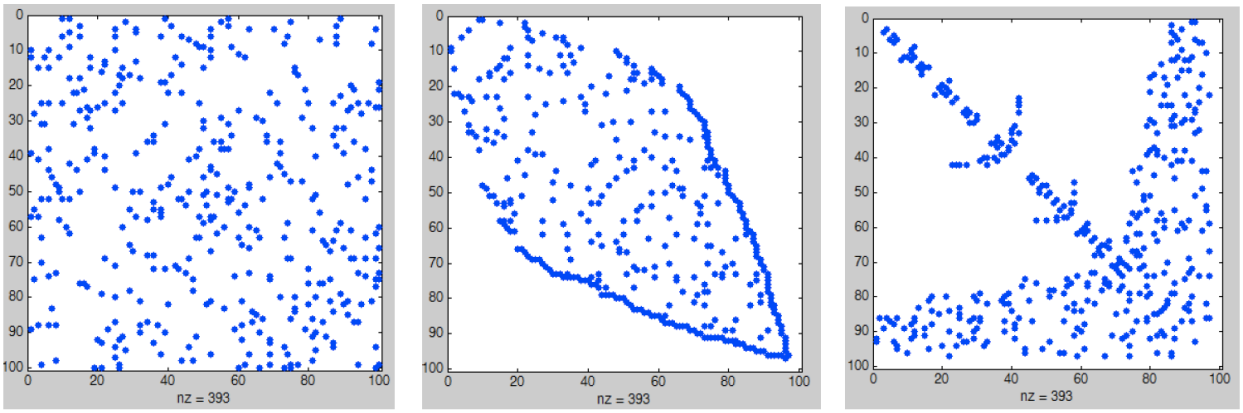


Figure 7.12: Comparison of reordering techniques RCM and AMD for a random sparse matrix.

The first step of Cholesky gives

$$\alpha = 2, \qquad v = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \qquad B - \frac{vv^T}{\alpha} = \begin{bmatrix} 2 & \\ & 2 \end{bmatrix} - \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 3/2 & -1/2 \\ -1/2 & 3/2 \end{bmatrix}.$$

Hence the result after one step of Cholesky is:

$$\begin{bmatrix} 2 & & \\ & 3/2 & -1/2 \\ & -1/2 & 3/2 \end{bmatrix} \quad \longleftrightarrow \quad$$

Matrix

Elimination Graph $G(A^{(1)})$

Note that the zeros have filled in corresponding to nodes 2 and 3 now connecting.

When multiple nodes have same degree we need a strategy to break ties. Some possible strategies are:

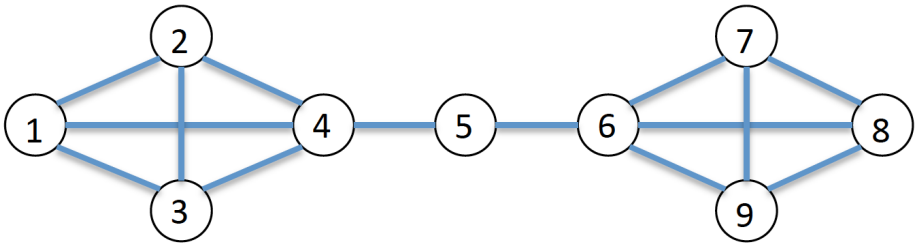
1. select the node with smallest node number in the original ordering,
2. pre-order with RCM, then select the node that is numbered earlier according to an RCM ordering (computed in advance),
3. Various others, e.g. "multiple minimum degree" chooses multiple nodes that don't interact and eliminate them at once.

In practice, tie breaking may have a significant impact on the order. Figure 7.12 also shows a comparison of Matlab (more advanced) variant of MD called symamd (symmetric approximate minimum degree) with RCM. Unlike CM and RCM, MD does not try to minimize bandwidth.

Algorithm 7.4 : Minimum Degree Ordering

Input: undirected graph $G = (V, E)$
Output: permuted node set S
$S \leftarrow \emptyset$
while $V \neq \emptyset$
    find minimum degree node $v \in V$ ▷ use chosen tie breaking rule
    $S \leftarrow S \cup \{v\}$ ▷ append to sorted list S
    $V \leftarrow V \setminus \{v\}$
    for $u \in v$.adj
        for $w \in v$.adj
            if $u \neq w$ and $\{u, w\} \notin E$ **then**
                $E \leftarrow E \cup \{u, w\}$
            end if
        end for
        $E \leftarrow E \setminus \{u, v\}$
    end for
end while

Minimum degree reordering is optimal for trees, i.e., MD produces no fills on trees. To see this think about the elimination graphs for a tree: each time we are removing leaves (with degree 1), hence will never introduce new edges. However, MD is still a local strategy with no guarantee of absolute minimum total fill (which is NP-complete). Consider the following counterexample of optimality. The graph



suffers no fill-in. This can be seen from the matrix of this graph

$$\begin{bmatrix} \times & \times & \times & \times & & & & & \\ \times & \times & \times & \times & & & & & \\ \times & \times & \times & \times & & & & & \\ \times & \times & \times & \times & \times & & & & \\ & & & \times & \times & \times & & & \\ & & & & \times & \times & \times & \times & \times \\ & & & & & \times & \times & \times & \times \\ & & & & & \times & \times & \times & \times \\ & & & & & \times & \times & \times & \times \end{bmatrix}.$$

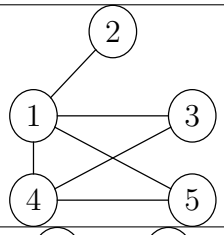
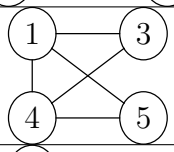
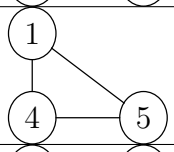
Factorization can only fill in the envelope, but its envelope is already totally filled! Unfortu-

nately, the minimum degree ordering would create fill in this example. MD ordering would eliminate node 5 first since it has the minimum degree of 2. This would immediately connect nodes 4 and 6, corresponding to fill (denoted $\times$)

$$\begin{bmatrix} \times & \times & \times & \times & & & & & & \\ \times & \times & \times & \times & & & & & & \\ \times & \times & \times & \times & & & & & & \\ \times & \times & \times & \times & \times & \times & & & & \\ & & & \times & \times & \times & & & & \\ & & & \times & \times & \times & \times & \times & \times & \\ & & & & & \times & \times & \times & \times & \\ & & & & & \times & \times & \times & \times & \\ & & & & & \times & \times & \times & \times & \end{bmatrix}.$$

A Fully Worked Out Example of Minimum Degree Re-Ordering Here we fully work out the example which is started above.

We follow Algorithm 7.1.2 (i.e. we don't re-label as we go: we work only with the graph, ignoring what is happening in the matrix)

After Step	Remaining Graph	S (sorted)
0		$\emptyset$
1		2
2		2, 3
3		2, 3, 1
4		2, 3, 1, 4
5	$\emptyset$	2, 3, 1, 4, 5

This yields the Minimum degree ordering:

Sort Order	1	2	3	4	5
Node Number	2	3	1	4	5

Finally, we mention some of the many possible improvements of MD:

- "supervariables"/indistinguishable nodes: nodes with identical adjacency structure (neighbours) can be eliminated simultaneously,
- multiple elimination: non adjacent nodes of same degree can also be safely eliminated simultaneously,

- approximate minimum degree: use an approximation to the degree updates of neighbours, which improves the run time,
- quotient graph: smarter graph representation to reduce storage.

Remember that our overall goal of reordering is to **minimize computation and storage costs** of factorization on sparse matrices by limiting fill. Minimum degree ordering tries to greedily minimize fill at each step by eliminating the node with least degree. MD often outperforms RCM but is still just a heuristic!

7.1.3 Stability (Optional)

The story so far in this course has been:

- Direct methods for solving linear systems rely on matrix factorization,
- Matrices often have useful properties (e.g., sparse, banded, symmetric, PD),
- We can design efficient algorithms by exploiting these properties,
- Matrix reordering can reduce cost for sparse matrices by (heuristically) reducing fill-in,
- Application arises from finite difference methods for PDE problems in heat conduction.

- Here we will discuss some stability issues for factorization. We consider another use for row/column swaps to now help with stability.
- How do small error/changes in a matrix problem $Ax = b$ affect the (exact) solution?
- The **matrix condition number**, $\kappa(A) = \|A\|\|A^{-1}\|$ (where $\|A\| = \max \frac{\|Ax\|}{\|x\|}$), provides a measure for this.
- Note that $\kappa \geq 1$.
- The condition number $\kappa(A)$ can provide an upper bound on the change in x due to the relative change δ in b and/or A.
- Specifically, if
$$\max\left(\frac{\|\Delta A\|}{\|A\|}, \frac{\|\Delta b\|}{\|b\|}\right) \leq \delta,$$
then
$$\frac{\|\Delta x\|}{\|x\|} \leq 2\kappa(A)\delta + O\left(\delta^2\right).$$

Stability is a property of the numerical algorithm, which is distinct from conditioning of the problem. Essentially, stability is concerned with how errors or changes in input to the numerical algorithm affect the output. For example, do small errors magnify or shrink during computation? It is important to note that a highly stable algorithm cannot prevent issues due to a poorly conditioned problem. Furthermore, an unstable algorithm can give useless results, even for a well-conditioned problem.

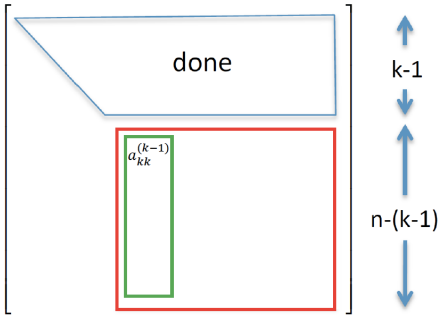
Here we will discuss the stability of LU factorization. The basic goal is to find L and U whose "size" remains under control. Huge entries will also inflate round off error and produce useless results. For example, we do not want re-multiplying LU together to give a new $\hat{A}$ that is far from the input A.

7.1.4 Pivoting (Optional)

Factorizations considered so far assume diagonal entry $a_{kk}^{(k-1)}$ (i.e. the **pivot**) at each step is non-zero. The notation $A^{(k-1)}$ denotes the remaining unfactored lower right matrix block during LU. Recall that when doing row subtraction, we scale the active row by a column entry a_{ik} **divided by the pivot value** a_{kk}. Problems arise when pivot is zero ($a_{kk}^{(k-1)} = 0$) or close to zero ($a_{kk}^{(k-1)} \approx 0$). That is, the problem of division by zero occurs when $a_{kk}^{(k-1)} = 0$. When $a_{kk}^{(k-1)} \approx 0$ we introduce large round off errors. These issues are usually addressed through **pivoting**.

Pivoting for stability is done by permuting rows/columns to get a larger magnitude element as the pivot, $a_{kk}^{(k-1)}$. There are two forms of pivoting

1. Complete pivoting swaps rows and columns to use the single largest magnitude element of $A^{(k-1)}$ (red square) as the pivot.
2. Partial pivoting swaps only rows to put largest magnitude element of column k (green rectangle) into the pivot position (row k).



The LU factorization algorithm with partial pivoting satisfies

$$\hat{L}\hat{U} = \hat{P}A + \Delta A, \qquad \frac{\|\Delta A\|}{\|A\|} = O(\rho \epsilon_{\text{machine}}),$$

where the hat notation indicates the numerical solution. The scalar ρ is called the growth factor and can be approximated as

$$\rho \approx \frac{\|U\|}{\|A\|}.$$

Lecture 22 of Trefethen & Bau is a good reference for more information.

So, is partial pivoting enough? There are examples where partial pivoting is inadequate. Consider

$$\underbrace{\begin{bmatrix} 1 & & & & 1 \\ -1 & 1 & & & 1 \\ -1 & -1 & 1 & & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}}_{\|A\| \approx 3.145} = \underbrace{\begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ -1 & -1 & 1 & & \\ -1 & -1 & -1 & 1 & \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}}_{\|L\| \approx 2.736} \underbrace{\begin{bmatrix} 1 & & & & 1 \\ & 1 & & & 2 \\ & & 1 & & 4 \\ & & & 1 & 8 \\ & & & & 16 \end{bmatrix}}_{\|U\| \approx 18.473}.$$

Each step of LU induces magnification so the entries of U grow exponentially. Notice, eventhough $A = LU$ we see that $\|U\| \gg \|A\|$. The growth factor in this example is is $\rho \approx 2^{n-1}$. For larger and larger matrices we would easily start to run out of precision!

Fortunately, the constructed example above is not found in practical situations. In practice, partial pivoting is essentially always sufficient. Hence, the cost of complete pivoting is usually not justified. A quote from Trefethen & Bau states: "In fifty years of computing, no matrix problems that excite an explosive instability are known to have arisen **under natural**

circumstances." Partial pivoting yields a modified factorization $PA = LU$, where P is the permutation matrix due to row swaps.

Unnecessary Pivoting

Pivoting improves stability of the factorization, but tends to reduce sparsity. There is a trade-off that needs to be considered. But for certain matrices pivoting is never necessary.

Theorem 7.1. *If A is symmetric positive definite, then during LU factorization the pivot $a_{kk}^{(k-1)} > 0$ for all k.*

Proof. Assume A is SPD. We want to show that the pivot in the remaining submatrix $A^{(k-1)}$ is positive. For $n = 1$, the theorem is clearly true since a 1×1 SPD matrix is a positive scalar. We continue with an inductive argument. For $n > 1$, write

$$A = \begin{bmatrix} a_{11} & v^T \\ v & A_{22} \end{bmatrix},$$

where $a_{11} \in \mathbb{R}$, v is a column vector and A_{22} is a submatrix. Since A is SPD, $a_{11} > 0$ because it is a principle submatrix of A. Now consider eliminating v using a_{11} as the pivot for LU (careful not to confuse with Cholesky)

$$A = \begin{bmatrix} a_{11} & v^T \\ v & A_{22} \end{bmatrix} \to \begin{bmatrix} a_{11} & v^T \\ 0 & A_{22} - \frac{vv^T}{a_{11}} \end{bmatrix}.$$

Let $A_{22}^{(1)} = A_{22} - \frac{vv^T}{a_{11}}$. Note that $A_{22}^{(1)}$ is symmetric since vv^T is symmetric. We must also show that $A_{22}^{(1)}$ is PD, i.e., $x^T A_{22}^{(1)} x > 0, \forall x \neq 0$. (Because SPD matrices have positive diagonal entries, then $a_{22}^{(1)} > 0$.)

Since A is SPD, $y^T A y > 0, \forall y \neq 0$ by definition. Let $x \in \mathbb{R}^{n-1}$, $x \neq 0$. Define

$$y = \begin{bmatrix} -\frac{x^T v}{a_{11}} \\ x \end{bmatrix} \in \mathbb{R}^n.$$

Then,

$$\begin{aligned} 0 &< y^T A y, \\ &= \begin{bmatrix} -\frac{x^T v}{a_{11}} & x^T \end{bmatrix} \begin{bmatrix} a_{11} & v^T \\ v & A_{22} \end{bmatrix} \begin{bmatrix} -\frac{x^T v}{a_{11}} \\ x \end{bmatrix}, \\ &= x^T A_{22} x - \frac{1}{a_{11}} x^T v v^T x, \\ &= x^T \left(A_{22} - \frac{vv^T}{a_{11}} \right) x, \\ \Rightarrow 0 &< x^T \left(A_{22}^{(1)} \right) x. \end{aligned}$$

Hence, $A_{22}^{(1)}$ is SPD so $a_{22}^{(1)} > 0$. This process can be repeated, so $a_{kk}^{(k-1)} > 0$ for all k (and for all n). □

Pivoting is also unnecessary for:

1. Row strictly diagonally-dominant matrices

$$|a_{kk}| > \sum_{j \neq k} |a_{kj}| \qquad \text{for} \qquad k = 1, \ldots, n,$$

2. Column strictly diagonally-dominant matrices

$$|a_{kk}| > \sum_{j \neq k} |a_{jk}| \qquad \text{for} \qquad k = 1, \ldots, n.$$

That is, matrices whose diagonal entry is bigger in magnitude than the sum of remaining entries in the row/column (respectively).

7.2 Image De-Noising

Images often contain random "noise" (small errors), arising from the sensors, capture method, or (lighting) conditions. See for example Figure 7.13.

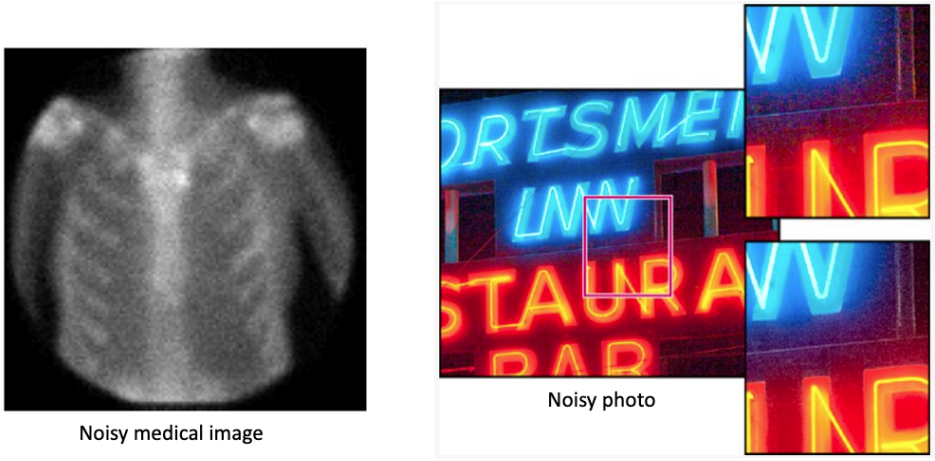


Figure 7.13: Example noisy images.

Synthetic images generated by raytracing have noise unless you run them for a very long time. An alternative approach is to raytrace for a short time, then clean up with some de-noising (see Figure 7.14).

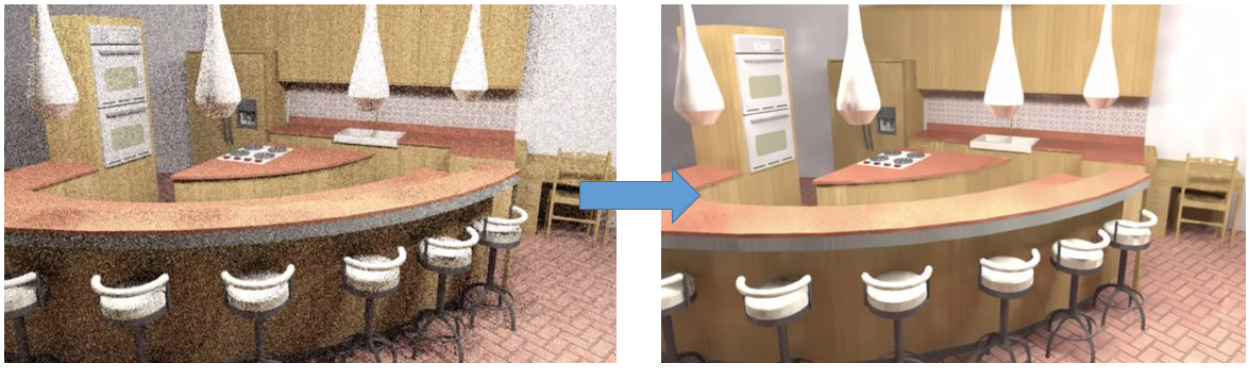


Figure 7.14: Example noisy synthetic image (left) and denoised image (right) from [Kalantari et al. SIGGRAPH 2015].

Often there is enough "signal" amidst the noise that we can try to recover a version with the noise removed/reduced.

7.2.1 Inverse Problems

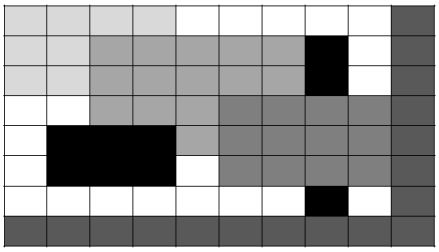
Image denoising is an **inverse problem**. That is, given some observations we want to reconstruct the source/factors that generated them. Given some (noisy) observation u^0 of some signal u^* we want to recover the clean signal u^*, i.e.,

$$u^0 = u^* + n,$$

where n is the noise. Thus, we want to decompose the observation u^0 into the sum of two components: the clean signal u^* and the noise n.

The observed image is u^0 is given. The goal is to find an approximation of u^*. We treat grayscale images as 2D scalar functions

u_{ij} = pixel intensity value at row i, column j.



Two key assumptions enabling us to solve the inverse problem:

1. noise is not too large, i.e., observation u^0 is "close" to signal u^*
2. signal u^* has some **structure** that we can exploit.

7.2.2 Regularization Models

We seek u satisfying

$$\min_u \alpha R(u) + \|u - u^0\|_2^2,$$

where $R(u)$ is the **regularization model**. In this form, the $\|u - u^0\|_2^2$ term can be thought of as a measure of the discrepancy between the observation u^0 and the numerical solution

u. (The notation $\|\cdot\|_2^2$ should remind us of the square of the 2-norm, i.e. the square of the Euclidean distance, i.e. the sum of the squares of the distances along each axis.)

The parameter $\alpha > 0$ is called the **regularization constant**, which controls the trade-off between

- **regularity** ("smoothness") and
- **fit** (fidelity to data u^0).

The regularization constant balances the two goals:

- $\alpha \to 0$: ignores the first term (regularization) implying $u \approx u^0$, so this basically outputs the observation u^0,
- $\alpha \to \infty$: ignores the second term (observation) implying $u \approx$ (minimizer of $R(u)$) giving a perfectly "regular" image.

Good recovery of a denoised image relies on

- an appropriate tuning of α, and on
- the regularization model $R(u)$.

We will discuss three options here:

1. **Tikhonov**,
2. **Laplacian**, and
3. **Total Variation** regularizations.

Tikhonov Regularization

For **Tikhonov regularization** $R(u)$ is a measure of the total sum of pixel intensity

$$R(u) = \|u\|_2^2.$$

Therefore our optimization becomes

$$\min_u \alpha\|u\|_2^2 + \|u - u^0\|_2^2.$$

Solving this quadratic optimization (e.g., via Euler-Lagrange equations - all details skipped) leads to

$$\begin{aligned} \alpha u + (u - u^0) &= 0, \text{ so} \\ (\alpha + 1)u &= u^0. \end{aligned}$$

Hence, the new pixel intensities are given by

$$u = \frac{u^0}{\alpha + 1}. \tag{7.19}$$

From (7.19) we see that the solution with Tikhonov regularization gives

- $u \to u^0$ as $\alpha \to 0$ and

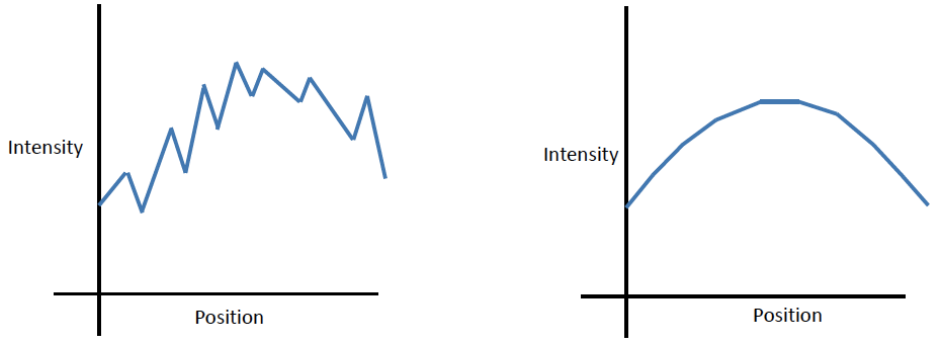


Figure 7.15: Example of a noisy signal (left) and smoother signal (right).

- $u \to 0$ when $\alpha \to \infty$.

Thus, α indeed balances matching the input data and being close to a perfectly regular image (image of all zeros). However, this is **really not what we want** from our regularization since it is pushing us towards a zero intensity image.

Laplacian Regularization

Consider a noisy "image" (signal) in 1D shown in Figure 7.15. It can be seen from the noisy 1D image that there is drastic change in slope throughout the image. If one compared to the smoother 1D image the slope changes more continuously. Therefore, we should try to penalize changes in slopes/derivatives, ∇u, instead of pixel values u.

The **Laplacian regularization model** $R(u)$ is a measure of the total sum of intensity **gradients**

$$R(u) = \|\nabla u\|_2^2.$$

So the optimization problem becomes

$$\min_u \alpha\|\nabla u\|_2^2 + \|u - u^0\|_2^2.$$

The optimal solution (minimizer) satisfies the linear PDE

$$\begin{aligned} -\alpha\nabla\cdot\nabla u + (u - u^0) &= 0, \\ -\alpha\Delta u + u &= u^0, \end{aligned} \tag{7.20}$$

where $\nabla \cdot \nabla = \Delta$ is the Laplace operator (from our PDE application).

We can apply finite differences to (7.20) to compute a numerical approximation of the minimizer u_{ij} at each pixel (i, j). Using the finite difference approximation of the Laplacian Δ, previously discussed in Lecture 4, we have

$$\frac{\alpha}{h^2}(4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}) + u_{ij} = u^0_{ij}.$$

This gives a matrix equation of the form $(\alpha A + I)u = u^0$. If the solution remains too noisy we can try iterating as

$$(\alpha A + I)u^{k+1} = u^k, \qquad \text{for} \qquad k = 1, 2, \ldots, K.$$

Figure 7.16: Laplacian regularization for image denoising of the noisy image (left). The result on the right is smeared out throughout the image.

However, **the drawback of Laplacian regularization is that it tends to smear out edges** as shown in Figure 7.16.

Total Variation Regularization

To avoid smearing edges the **total variation regularization** takes

$$R(u) = \|\nabla u\|_1.$$

This still minimizes the slopes but with a different measure that does not punish them too much (i.e., 1-norm, without squaring). So our optimization roughly becomes

$$\min_u \alpha\|\nabla u\|_1 + \|u - u^0\|_2^2.$$

The minimization problem leads to another PDE to solve, namely

$$-\alpha\nabla \cdot \left(\frac{1}{\|\nabla u\|_1}\right)\nabla u + u = u^0. \tag{7.21}$$

The PDE (7.21) is similar to the Laplacian regularization PDE, but with 1 instead of $\frac{1}{\|\nabla u\|_1}$. The effect is that the matrix coefficients (amount of smoothing) depends on gradients in the image themselves.

1. Near big intensity jumps (edges of objects in an image)

$$\|\nabla u_{ij}\| \text{ is large } \Rightarrow \frac{1}{\|\nabla u_{ij}\|} \text{ is small!}$$

   Therefore the 1st term becomes negligible giving $u \approx u^0$, which leaves edges nearly unchanged staying close to data u^0.

2. However, in "flat" regions, where intensity is roughly constant,

$$\|\nabla u_{ij}\| \text{ is small } \Rightarrow \frac{1}{\|\nabla u_{ij}\|} \text{ is large!}$$

This implies more diffusion at pixel (i, j) since effectively we have

$$-C\nabla \cdot \nabla u_{ij} + u_{ij} = u^0_{ij}, \qquad \text{where } C \text{ is some large value.}$$

The increase in diffusion makes these regions flatter/smoother.

To summarize, the behaviour of total variation regularization is

1. Edge-like regions are smoothed less, and
2. flatter regions are smoothed more.

So we get smoothing that roughly "stays within the lines". Figure 7.17 shows results from the original paper of a noisy image (top) and total variation denoised image (bottom).

We will now discuss how to compute a numerical approximation to (7.21). We can apply a finite difference discretization again of the form

$$\alpha A(u)u + u = u^0,$$

but this equation is **nonlinear**. The coefficients in the matrix $A(u)$ depend on the solution u itself. We need to solve this equation numerically: we cannot solve the PDE directly, as we did for the earlier techniques. A simple approach to solve nonlinear equations is the **fixed point iteration**. We freeze the coefficients to make the equations linear, solve, update, and repeat. That is, solve

$$\begin{aligned} & \alpha A(u^k)u^{k+1} + u^{k+1} &=& \ u^0, \\ \Rightarrow \quad & (\alpha A(u^k) + I)u^{k+1} &=& \ u^0, \qquad \text{for } k = 0, 1, \ldots, K. \end{aligned}$$

We pick an initial guess and compute an approximate solution by solving the system. The matrix $A(u^k)$ is then recomputed for the next iteration.

Note that such an iteration **does not always converge to the solution** in general. Fortunately, in this case the fixed point iteration does converge. There are different approaches to determine when to stop iterating, i.e., what is K? One approach is to stop iterating when the approximation is not changing much anymore, i.e.

$$\|u^{k+1} - u^k\| < \text{tol},$$

for some small tolerance.

Figure 8.20 compares the Laplacian and total variation regularization. The Laplacian regularization is unable to remove as much noise as the total variation regularization. Increasing the regularization parameter α for the Laplacian regularization will just blur the image instead of removing noise. The effect of increasing α for total variation regularization is shown in Figure 7.19. Edges are still well preserved as α increases and the image becomes smoother.

Remarks:

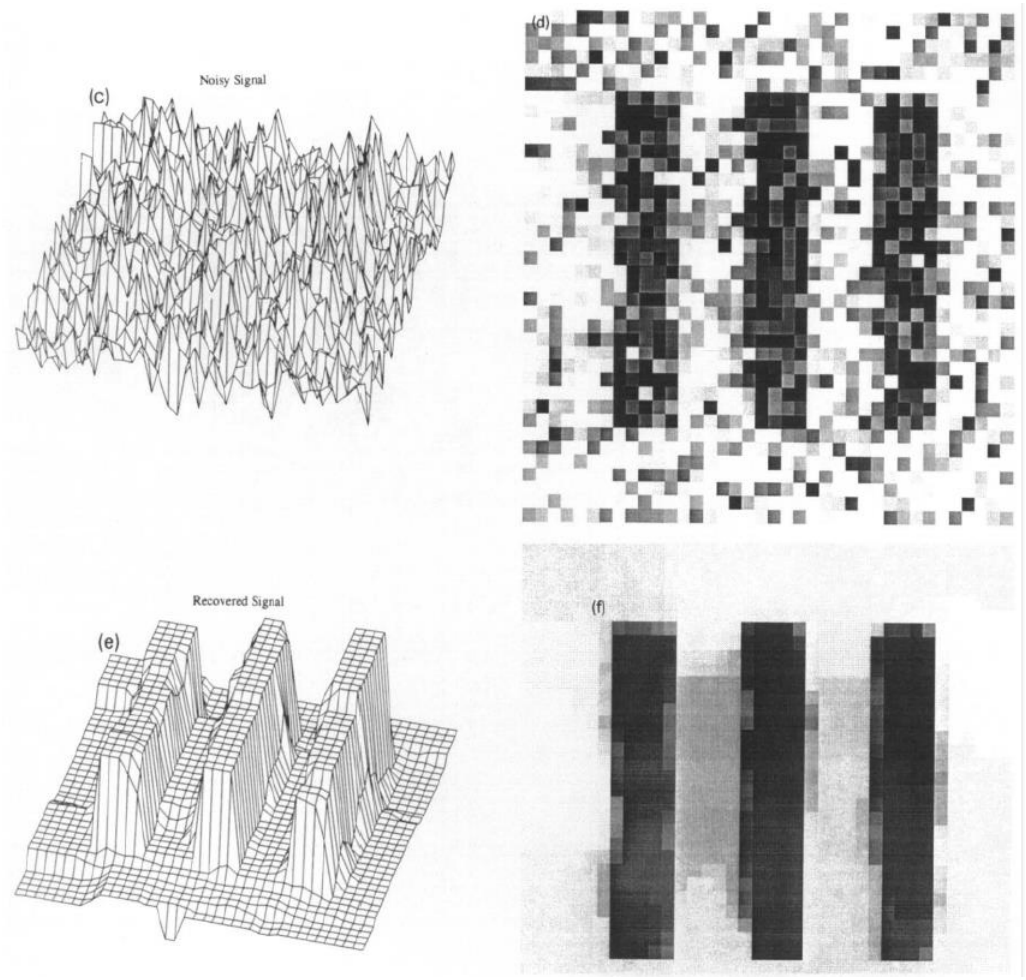


Figure 7.17: Noisy image (top) and denoised image (bottom) using total variation regularization [Rudin et al. 1992].

Figure 7.18: Comparison denoising an image with Laplacian and total variation regularization (images from Mathworks Matlab manual).

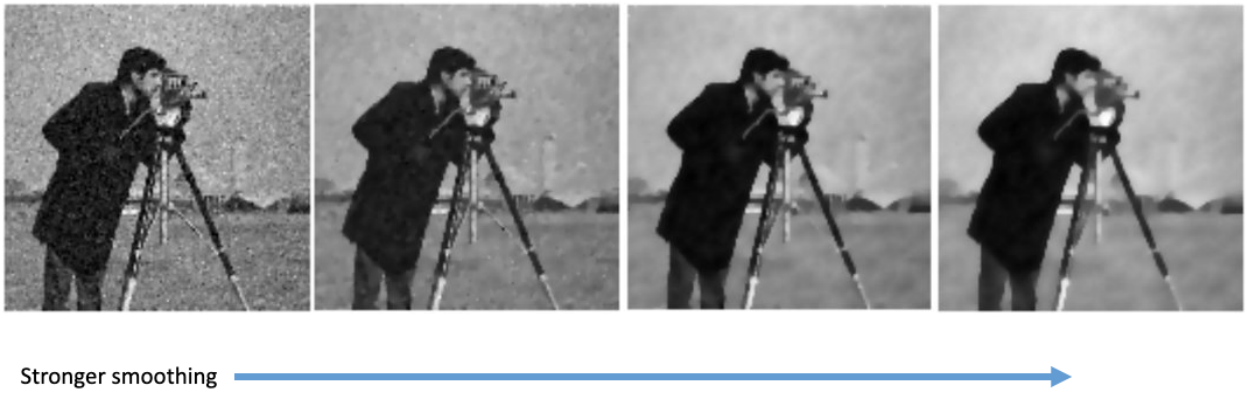


Figure 7.19: Effect of increasing the regularization parameter α with total variation regularization.

1. Many questions about how to complete these computations have been left unanswered so far. When we work out such examples on the Crowdmark assignments, all the required details will be specified.

8 Lecture 08: Iterative Methods

Outline

1. Iterative Methods
   (a) Stationary Iterative Methods
2. Splitting Methods
   (a) Richardson
   (b) Jacobi
   (c) Gauss-Seidel
   (d) Successive Over Relaxation (SOR)
3. Convergence of Splitting Methods

8.1 Iterative Methods

The previous lecture concluded our look at direct methods for linear systems. These methods are based on factoring the matrix A. They solve the system in a known finite sequence of steps, then return the solution. In this lecture we begin looking at **iterative** methods for linear systems. These methods gradually and iteratively refine a solution. They repeat the same steps over and over, then stop only when a desired tolerance is achieved.

Possible benefits of iterative methods compared to direct methods:

1. They may be faster and tend require less memory.
2. They may be faster for typically large, sparse, higher-dimensional problems, since they are usually less memory-intensive since no fill-in occurs. LU factorization was $O(n^3)$ in the worst (fully dense) case for $A \in \mathbb{R}^{n \times n}$. For iterative methods the operation count depends on number of non-zeros (nnz), as well as, how many iterations it takes.
3. Another benefit for applications needing only approximate solutions is that one can **"quit early"**. With iterative methods you can increase your error tolerance to obtain a less accurate approximate solution. Ideally, iterative approaches make gradual progress in the solution quality up to the tolerance (or limits of floating point arithmetic). Direct approaches give no solution at all until they complete all operations. Depending on the problem one or the other may get to a (satisfactory) solution first as shown in Figure 8.20.
4. Exact algorithms such as Gaussian elimination need to alter the matrix A. The splitting algorithms discussed below do not. If we cannot find a good ordering to (significantly) reduce the amount of fill-in, then iterative algorithms should be used (for large problems). The downside of course is that we give up computing an exact (up to machine precision) solution.

Termination Criterion The termination criterion is based on the error $e = x - \hat{x}$ between

1. the (current, approximate) numerical solution $\hat{x}$ and
2. the true solution x.

We terminate computation when $e \approx 0$. However, we do not know the true solution x since

that is what we are trying to compute. So a more practical indicator of the error is the norm of the **residual**

$$r = b - A\hat{x}.$$

The residual measures how much the current approximation fails to satisfy $A\hat{x} = b$.

The residual and error satisfy $Ae = r$ since

$$\begin{aligned} Ax &= b, \\ \Rightarrow Ax - A\hat{x} &= b - A\hat{x}, \\ \Rightarrow A\underbrace{(x - \hat{x})}_{e} &= \underbrace{b - A\hat{x}}_{r}, \\ \Rightarrow Ae &= r. \end{aligned}$$

Therefore, the matrix condition number (See Stability in Lecture 07) can be used to bound the (relative) size of error e and residual r as

$$\frac{\|e\|}{\|x\|} \leq \kappa(A)\frac{\|r\|}{\|b\|}.$$

From this we see that a small residual can imply a small error, but only if A is **well conditioned** (i.e., κ is small).

As an example, consider the 1×1 linear system $5x = 300$.

- The true solution is obviously $x = 60$.
- If our current best estimate is $\hat{x} \approx 50$, then the error is $e = x - \hat{x} = 60 - 50 = 10$.
- This is the exact error of how "wrong" $\hat{x}$ is.
- The residual in this case is $r = b - A\hat{x} = 300 - 5(50) = 50$ (i.e., how far is $b - A\hat{x}$ from zero?).
- Notice also that the claim of $Ae = r$ is satisfied since $5(10) = 50$.

8.1.1 Stationary Iterative Methods

Iterative methods start from some (perhaps arbitrary or zero) guess at the solution. Increasingly accurate approximations to the solution are generated by iterating a basic procedure repeatedly.

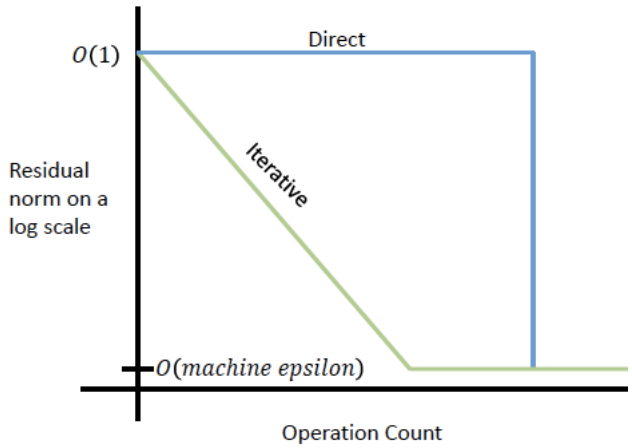


Figure 8.20: Comparison of the path to solutions of iterative versus direct methods.

Q & A

1. Do we always know that we will converge towards a solution?
   A: No. We will need to be careful about this point, because convergence is not guaranteed without additional assumptions. We will discuss these convergence assumptions, soon.
2. If an iterative method converges, must it produce a "close enough" answer more quickly than the direct methods that we have studied?
   A: No: Not all iterative methods are created equal, and the meaning of "close enough" completely depends on the choice of tolerance.

8.2 Splitting Methods

This section gives describes splitting methods for iteratively solving linear equations. We can rewrite the linear system $Ax = b$ as

$$(M - N)x = b \Leftrightarrow Mx = Nx + b, \qquad \text{where } A = M - N.$$

Important Notes About Notation:

1. We assume that M must be **invertible**.
2. (In each method described below, you should think about what extra constraints this would place on the coefficient matrix, A.)
3. However, we do **not** necessarily compute M^{-1} to solve a system involving M.
4. We abuse notation slightly in what follows, by writing $M^{-1}b$ as shorthand for the solution, x, to the system $Mx = b$.

Then, starting with some initial guess x^0, we can **iteratively** find x by repeatedly solving

$$Mx^{k+1} = (Nx^k + b). \tag{8.22}$$

Then we have:

$$\begin{aligned} Mx^{k+1} &= Nx^k + b \\ &= (M-A)x^k + b \\ &= Mx^k - Ax^k + b \\ &= Mx^k + (b - Ax^k) \\ x^{k+1} &= x^k + M^{-1} \underbrace{(b - Ax^k)}_{r^k,\text{ at step } k\ (\hat{x}=x^k)} \end{aligned} \tag{8.23}$$

For the splitting method (8.22) to be effective, we need to choose M (hence N) so that

1. it is easy to solve the linear system (8.22), i.e., $My = z$ should be easy to solve.
2. M is close to A, in the sense of having small norm $\|I - M^{-1}A\|$.

At one extreme we could choose $M = A$, and the iterative procedure (8.22) will "converge" in one iteration since

$$x^{k+1} = x^k + A^{-1}(b - Ax^k) = x^k + x - x^k = x.$$

However, using the actual A^{-1} is too expensive (and defeats the purpose). The cost would be solving a general linear system $Ax = b$. There is a trade-off between the two goals, so we want to take $M \approx A$.

For different choices of M we get the following splitting methods to discuss in this lecture:

1. Richardson,
2. Jacobi,
3. Gauss-Seidel,
4. Successive Over Relaxation (SOR).

Remark: splitting methods are related to **fixed-point iterations**. Suppose we want to solve the (nonlinear) equation

$$f(x) = 0,\ x \in X.$$

We consider the iteration

$$x^{k+1} = T(x^k) = T^{k+1}(x_0),$$

where the mapping $T : X \to X$ is chosen so that any fixed-point satisfying $x = T(x)$ implies $f(x) = 0$. Many iterative algorithms can be derived as above. For our splitting algorithm in (8.23), the function $f(x) = Ax - b$ and $T(x) = x + M^{-1}(b - Ax)$. Indeed, $x = Tx \Rightarrow f(x) = 0$.

8.2.1 Richardson Iteration

Richardson iteration is perhaps the simplest method. The choice of M the scaled identity matrix

$$M = \frac{1}{\theta} I,$$

where $\theta > 0$ is some appropriately chosen constant. How to choose θ will be detailed when we discuss the convergence of iterative methods in Section 8.3. We have from (8.23) that the Richardson iteration is

$$x^{k+1} = x^k + \theta(b - Ax^k).$$

Or, for a particular i^{th} entry we have $x_i^{k+1} = x_i^k + \theta\left(b_i - \sum_{j=1}^n a_{ij}x_j^k\right)$. Note that the new value x^{k+1} is a weighted sum of old value x^k and the residual $b - Ax^k$. Clearly, each iteration costs $O(\text{nnz}(A))$. Note that we need to store 2 separate vectors, x^k and x^{k+1}.

8.2.2 Jacobi Iteration

The next three methods will rely on the following labelled submatrices of A

- D = main diagonal (all diagonal entries non-zero),
- $-L$ = (strictly) below diagonal,
- $-U$ = (strictly) above diagonal.

$$A = \begin{bmatrix} \ddots & & -U \\ & D & \\ -L & & \ddots \end{bmatrix}$$

With the choice

$$M = D := \text{diag}(A),$$

we have the Jacobi iteration from (8.23):

$$x^{k+1} = x^k + D^{-1}(b - Ax^k).$$

Intuitively, we "exactly" solve each row equation independently for the corresponding entry, using the current estimate of vector x^k. For example, consider if row 7 looks like

$$2x_5 - 5x_6 + 10x_7 + 3x_9 = 14.$$

Then to find x^{k+1} for row index $i = 7$ we set

$$x_7^{k+1} = \frac{1}{10}\left(14 - 2x_5^k + 5x_6^k - 3x_9^k\right),$$

using the other known (step k) values of x. Again, each iteration costs $O(\text{nnz}(A))$ and we must store 2 vectors.

Let us write the Jacobi iteration more explicitly for the ith entry as

$$\begin{aligned} x_i^{k+1} &= x_i^k + \frac{1}{a_{ii}}\left(b_i - \sum_j a_{ij}x_j^k\right), \\ &= \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}x_j^k\right). \end{aligned}$$

Motivation for the Jacobi iteration: Recall the residual vector $r = b - Ax$. Clearly, x is a solution iff $r = 0$. Given the current iterate x^k, the Jacobi iteration tries to zero out the

i-th residual r_i, **in turn**:

$$\begin{aligned} r_i &= 0 \\ \iff b_i - \sum_{j \neq i} a_{ij} x_j^k - a_{ii} x_i^{k+1} &= 0 \\ \Rightarrow x_i^{k+1} &= \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right). \end{aligned}$$

The Jacobi iteration is easy to implement but unfortunately quite slow. However, it is trivially **parallelizable**. Given x^k, we can update the components in the next iteration x^{k+1} in parallel.

8.2.3 Gauss-Siedel Iteration

The Gauss-Siedel iteration is very similar to the Jacobi iteration. The difference is instead of using "old" data, x^k, use "new" x^{k+1} data for entries that have already been updated so far on this pass. That is,

$$\begin{aligned} x_i^{k+1} &= x_i^k + \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j \geq i} a_{ij} x_j^k \right) \\ &= \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j>i} a_{ij} x_j^k \right). \end{aligned}$$

Recall we are zeroing out the residual at ith step - this gives the first equation.

If we rearrange the updates we have

$$\sum_{j \leq i} a_{ij} x_j^{k+1} = b - \sum_{j>i} a_{ij} x_j^k,$$

or in matrix form

$$\begin{aligned} (D-L)x^{k+1} &= b + Ux^k \\ \Rightarrow \quad x^{k+1} &= x^k + (D-L)^{-1}(b - Ax^k), \end{aligned}$$

so

$$M = D - L.$$

Again, each iteration costs $O(\text{nnz}(A))$. Gauss-Seidel however only needs to store one vector since you can update/overwrite x^k entries as you go!

There exist variants of Gauss-Siedel (GS). There is nothing special about proceeding to update x^k from top to bottom (i.e ., GS runs over rows from $i = 1, \ldots, n$). The reverse ordering gives **backward** Gauss-Seidel. This swaps the role of L and U giving

$$x^{k+1} = x^k + (D-U)^{-1}(b - Ax^k),$$

which corresponds to update the elements of x in the reverse order (i.e., $i = n, n-1, \ldots, 1$). Thus

$$M = D - U.$$

Of course, we can also combine (forward) Gauss-Siedel with backward Gauss-Siedel to construct **symmetric** Gauss-Seidel

$$\begin{aligned} x^{k+\frac{1}{2}} &= x^k + (D-L)^{-1}(b - Ax^k), \\ x^{k+1} &= x^{k+\frac{1}{2}} + (D-U)^{-1}(b - Ax^{k+\frac{1}{2}}) \\ &= \left(x^k + (D-L)^{-1}(b - Ax^k)\right) + (D-U)^{-1}(b - A(x^k + (D-L)^{-1}(b - Ax^k))) \end{aligned}$$

We simplify the second term first.

$$\begin{aligned} & (D-U)^{-1}(b - A(x^k + (D-L)^{-1}(b - Ax^k)) \\ = \; & (D-U)^{-1}(b - Ax^k - A(D-L)^{-1}(b - Ax^k)) \\ = \; & (D-U)^{-1}(I - A(D-L)^{-1})(b - Ax^k) \\ = \; & (D-U)^{-1}((D-L)(D-L)^{-1} - A(D-L)^{-1})(b - Ax^k) \\ = \; & (D-U)^{-1}(\underbrace{D - L - A}_{=U})(D-L)^{-1}(b - Ax^k) \\ = \; & (D-U)^{-1}U(D-L)^{-1}(b - Ax^k), \end{aligned}$$

so that we finally get

$$\begin{aligned} x^{k+1} &= \left(x^k + (D-L)^{-1}(b - Ax^k)\right) + (D-U)^{-1}U(D-L)^{-1}(b - Ax^k) \\ &= x^k + (D-L)^{-1}(b - Ax^k) + (D-U)^{-1}U(D-L)^{-1}(b - Ax^k) \\ &= x^k + \left[(D-L)^{-1} + (D-U)^{-1}U(D-L)^{-1}\right](b - Ax^k) \\ &= x^k + \left[I + (D-U)^{-1}U\right](D-L)^{-1}(b - Ax^k) \\ &= x^k + \left[(D-U)^{-1}(D-U) + (D-U)^{-1}U\right](D-L)^{-1}(b - Ax^k) \\ &= x^k + (D-U)^{-1}\left[D - U + U\right](D-L)^{-1}(b - Ax^k) \\ &= x^k + (D-U)^{-1}D(D-L)^{-1}(b - Ax^k) \end{aligned}$$

So that for symmetric Gauss-Seidel, the matrix M is

$$M = (D-L)D^{-1}(D-U).$$

Gauss-Siedel usually converges faster than the Jacobi iteration. However, GS is an inherently **sequential** algorithm and is harder to parallelize. **Red-black Gauss-Seidel** is an update ordering that allows for some parallelization (see Figure 8.21 left). It alternates between sweeps of updating (1) only red nodes and (2) only black nodes. We can update all red nodes in parallel since they only use (old) black data, and vice versa. One can generalize the red-black GS idea to non-grid structured problems by coloring a graph. You ensure no neighbours have same color and then update all same color neighbours simultaneously (see Figure 8.21 right). See for example the application of real time cloth simulation by [Fratarcangeli et al. 2016] in this video.

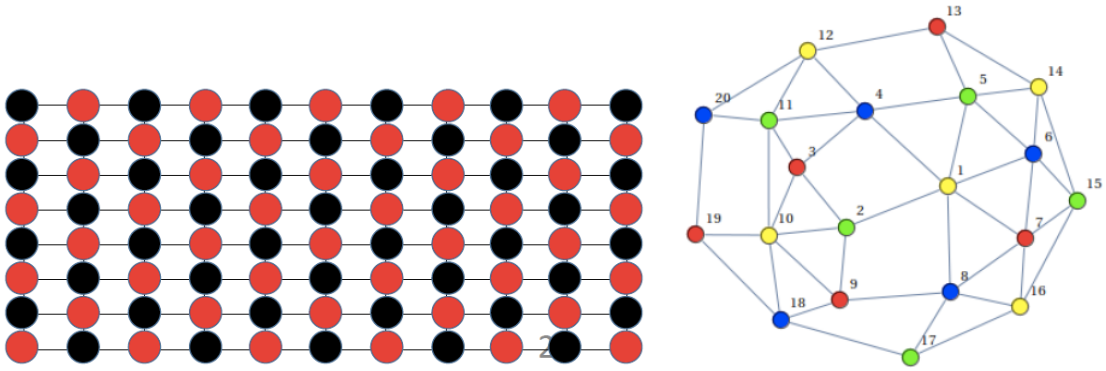


Figure 8.21: Example colorings for parallelizing the Gauss-Seidel iteration.

8.2.4 Successive Over-Relaxation (SOR)

A common strategy to accelerate fixed-point iterations is averaging. For instance, by averaging the current iterate x^k with the GS update according to a **relaxation factor** $\omega > 0$, we obtain SOR:

$$\begin{aligned} x_i^{k+1} &= (1-\omega)x_i^k + \omega(x_i^{k+1})^{\texttt{GS}} \\ &= (1-\omega)x_i^k + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{k+1} - \sum_{j>i} a_{ij}x_j^k\right), \end{aligned}$$

or in matrix notation

$$x^{k+1} = x^k + \left(\frac{1}{\omega}D - L\right)^{-1}\left(b - Ax^k\right).$$

Hence, for SOR the matrix M is

$$M = \frac{1}{\omega}D - L.$$

For $0 < \omega < 1$ the update is called **under-relaxation**, while for $\omega > 1$ the update is called **over-relaxation**. When $\omega = 1$, SOR equals Gauss-Seidel. For certain choices of $\omega(> 1)$, SOR may converge substantially faster than GS.

Detailed Explanation of the SOR Setup Above

Remark: This is adapted from the Saad textbook.

Let $\omega > 0$ be an arbitrary relaxation factor. I claim that the system $Ax = b$ can be equivalently written as

$$(D - \omega L)x = \omega b - [\omega(-U) + (\omega - 1)D]\, x. \tag{8.24}$$

To prove the claim, observe that

$$\begin{aligned} Ax &= b \\ \omega Ax &= \omega b \\ \omega\left[D-U-L\right]x &= \omega b \\ \omega Dx - \omega Ux - \omega Lx &= \omega b \\ Dx - \omega Lx &= \omega b - \omega Dx + \omega Ux + Dx \\ (D-\omega L)x &= \omega b - \left[\omega D - \omega U - D\right]x \\ &= \omega b - \left[\omega(-U) + (\omega-1)D\right]x, \end{aligned}$$

as claimed.

We iterate equation (8.24) to obtain

$$(D-\omega L)x^{k+1} = \omega b - \left[\omega(-U) + (\omega-1)D\right]x^k, \tag{8.25}$$

from which we can compute x^{k+1} given x^k.

To obtain the equation that matches averaging the current x^k with the GS update, observe that $(D-\omega L)$ is lower triangular. Hence we can solve the system defined by (8.25), via forward substitution. This gives

$$x_i^{k+1} = (1-\omega)x_i^k + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{k+1} - \sum_{j>i} a_{ij}x_j^k\right).$$

Our last step is to verify that the provided matrix form of the iteration is correct, so that the definition of the iteration matrix M is also correct. We compute

$$\begin{aligned} (D-\omega L)x^{k+1} &= \omega b - \left[\omega(-U) + (\omega-1)D\right]x^k \\ \left(\frac{1}{\omega}D - L\right)x^{k+1} &= b - \left[(-U) + \left(1-\frac{1}{\omega}\right)D\right]x^k \\ x^{k+1} &= \left(\frac{1}{\omega}D - L\right)^{-1}\left(b - \left[(-U) + \left(1-\frac{1}{\omega}\right)D\right]x^k\right) \\ &= x^k + \left(\frac{1}{\omega}D - L\right)^{-1}\left(b - \left[(-U) + \left(1-\frac{1}{\omega}\right)D + \left(\frac{1}{\omega}D - L\right)\right]x^k\right) \\ &= x^k + \left(\frac{1}{\omega}D - L\right)^{-1}\left(b - \left[(-U) + (-L) + D\right]x^k\right) \\ &= x^k + \left(\frac{1}{\omega}D - L\right)^{-1}\left(b - Ax^k\right), \end{aligned}$$

as claimed.

8.3 Convergence of Splitting Methods

The key questions involving the convergence of splitting methods are:

1. under what conditions does the iteration converge to the correct solution?
2. if it does converge, how quickly does it do so (in terms of number of iterations)?

These convergence questions depend on the **spectral radius** of A, denoted $\rho(A)$. The spectral radius is defined in terms of the eigenvalues of A.

Definition 8.1. *An **eigenvalues** $\lambda \in \mathbb{R}$ and corresponding eigenvector $v \in \mathbb{R}^n$ of $A \in \mathbb{R}^{n\times n}$ satisfy*

$$Av = \lambda v \qquad \text{and } v \neq 0.$$

Definition 8.2. *The **spectral radius** of A is*

$$\rho(A) = \max_i |\lambda_i|,$$

where λ_i are the eigenvalues of A. In other words, $\rho(A)$ is the largest magnitude of an eigenvalue of A.

We can rewrite our usual iteration as

$$x^{k+1} = x^k + M^{-1}(b - Ax^k) = (I - M^{-1}A)x^k + M^{-1}b.$$

We call the matrix $I - M^{-1}A$ the **iteration matrix** for a particular method. The following theorem gives a sufficient condition for convergence.

Theorem 8.1. *Let the true solution x^* satisfy $x^* = (I - M^{-1}A)x^* + M^{-1}b$. If $\|I - M^{-1}A\| < 1$ for some induced matrix norm, then the stationary iterative method converges. That is, for any initial guess x^0,*

$$\lim_{k\to\infty} x^k = x^*.$$

Proof. The iteration is

$$x^{k+1} = (I - M^{-1}A)x^k + M^{-1}b, \tag{8.26}$$

and the true solution satisfies

$$x^* = (I - M^{-1}A)x^* + M^{-1}b. \tag{8.27}$$

Subtracting (8.27) from (8.26) gives

$$x^{k+1} - x^* = (I - M^{-1}A)(x^k - x^*).$$

So for any vector norm $\|\cdot\|$ we have

$$\begin{aligned} \|x^{k+1} - x^*\| &\leq \|I - M^{-1}A\| \cdot \|x^k - x^*\|, \\ \Rightarrow \|e^{k+1}\| &\leq \|I - M^{-1}A\| \cdot \|e^k\|, \end{aligned}$$

where the matrix norm is the one induced by the vector norm. Therefore, if $\|I - M^{-1}A\| < 1$ the magnitude of the error decreases at each iteration. □

Note that an induced matrix norm is defined as $\|A\| := \max_{\|x\|=1} \|Ax\|$. A necessary and sufficient theorem for convergence is given next (we will not prove this).

Theorem 8.2. *The iterative method $x^{k+1} = x^k + M^{-1}(b - Ax^k)$ converges for any x^0 and b if and only if $\rho(I - M^{-1}A) < 1$.*

The speed of convergence depends on the size of $\rho(I - M^{-1}A)$ since the error satisfies

$$\|x^{k+1} - x^*\| \leq \rho(I - M^{-1}A)\|x^k - x^*\|.$$

That is, magnitude of the error scales by $\rho(I - M^{-1}A)$ on each iteration. We will call $\rho(I - M^{-1}A)$ the **convergence factor**. Smaller ρ (closer to zero) implies faster convergence. For even more on convergence see Saad textbook, sections 4.2.1 and 1.8.4.

9 Lecture 09: Iterative Methods - Conjugate Gradient Method

Outline

1. Solution by Steepest Descent
   (a) Towards the Conjugate Gradient Method
2. Another Search Direction Idea
   (a) Gram-Schmidt (A-)orthogonalization
   (b) Conjugate Directions Method
3. Conjugate Gradient Method
   (a) Efficient Conjugate Gradient Method
   (b) Error Behaviour

In Lecture 08 we looked at stationary iterative methods. We will now begin to look at other iterative methods for solving $Ax = b$. The **steepest descent** method and the **conjugate gradient** method are discussed in this lecture.

There is an equivalent minimization interpretation of solving $Ax = b$. We assume $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite (SPD) and consider the quadratic function

$$F(x) = \frac{1}{2}x^T Ax - b^T x, \quad x \in \mathbb{R}^n.$$

We will show that the solution of $Ax = b$ is equivalent to the solution of the minimization problem

$$\min_x F(x).$$

Consider visualizing the case with $n = 2$ as shown in Figure 9.22. Here x is a length-2 vector $x = [x_1, x_2]^T$. The function $F(x)$ is a scalar that gives height in the plot. Plotting F gives a paraboloid with minimum value at $x = A^{-1}b$. Note that A being SPD implies F is convex.

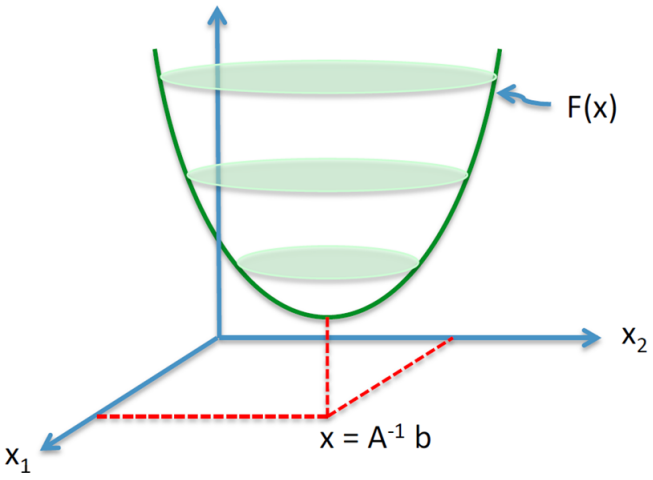


Figure 9.22: Visualization of $F(x)$ when $n = 2$.

Lemma 9.1. *For any A and x, $\nabla\left(x^TAx\right) = (A + A^T)x$.*

Proof. First observe that $x^TAx = \sum_{i=1}^n \left(x_i \sum_{j=1}^n x_j A_{ij}\right)$.

Now let $1 \leq k \leq n$ be arbitrary. Then

$$\begin{aligned} & \frac{\partial}{\partial x_k}\left(x^TAx\right) \\ \underbrace{=}_{\text{product rule}} & \sum_{i=1}^n \left(\frac{\partial x_i}{\partial x_k}\sum_{j=1}^n x_j A_{ij}\right) + \sum_{i=1}^n \left(x_i \sum_{j=1}^n \frac{\partial x_j}{\partial x_k} A_{ij}\right) \\ = & \sum_{j=1}^n x_j A_{kj} + \sum_{i=1}^n x_i A_{ik} \end{aligned}$$

Assembling each of the above for all $1 \leq k \leq n$ gives

$$\begin{aligned} & \nabla\left(x^TAx\right) \\ = & \begin{bmatrix} \sum_{j=1}^n x_j A_{1j} + \sum_{i=1}^n x_i A_{i1} \\ \sum_{j=1}^n x_j A_{2j} + \sum_{i=1}^n x_i A_{i2} \\ \vdots \\ \sum_{j=1}^n x_j A_{nj} + \sum_{i=1}^n x_i A_{in} \end{bmatrix} \\ = & Ax + \left(x^TA\right)^T \\ = & (A + A^T)x. \end{aligned}$$

□

Theorem 9.1. *If A is symmetric, then $\nabla F(x) = Ax - b$.*

Proof. By Lemma 9.1, $\nabla\left(x^TAx\right) = (A + A^T)x$.

Similarly, for an arbitrary $1 \leq k \leq n$, we have

$$\begin{aligned} \frac{\partial}{\partial x_k}\left(b^Tx\right) &= \frac{\partial}{\partial x_k}\left(\begin{bmatrix} b_1 & \cdots & b_n \end{bmatrix}\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}\right) \\ &= \frac{\partial}{\partial x_k}\sum_{j=1}^n b_j x_j \\ &= \sum_{j=1}^n b_j \frac{\partial x_j}{\partial x_k} \\ &= b_k, \text{ so that} \\ \nabla\left(b^Tx\right) &= b. \end{aligned}$$

Putting it all together, we get

$$\begin{aligned}
& \nabla F(x) \\
= \quad & \nabla\left(\frac{1}{2}x^TAx - b^Tx\right) \\
= \quad & \frac{1}{2}\nabla\left(x^TAx\right) - \nabla\left(b^Tx\right) \\
= \quad & \frac{1}{2}(A + A^T)x - b \\
\underbrace{=}_{A^T=A} \quad & \left(\frac{2A}{2}\right)x - b \\
= \quad & Ax - b.
\end{aligned}$$

□

Theorem 9.2. *The solution of the linear system and minimization form are the same.*

Proof. The solution of the minimization satisfies $\nabla F(x) = 0$, i.e.,

$$\frac{\partial F}{\partial x_i} = 0, \forall i.$$

Since, by Theorem 9.1, $\nabla F(x) = Ax - b$, therefore the solution of $Ax = b$ corresponds to a stationary point. However, since $F(x)$ is (strictly) convex any local minimum is the global minimum. □

9.1 Solution by Steepest Descent

By Theorem 9.2, if we can solve the minimization problem, we have solved the linear system. So we will try to find the minimum by "walking downhill". The main idea is, at each step, first choose a **search direction** vector $p \neq 0$. Then find the point along that direction with the lowest value. Therefore, we iterate as $x^{k+1} = x^k + \alpha p$, where $\alpha \in \mathbb{R}$.

We want to find α that gives the minimum value of $F(x^{k+1}) = F(x^k + \alpha p)$. Let

$$\begin{aligned}
f(\alpha) &= F(x^k + \alpha p), \\
&= \frac{1}{2}(x^k + \alpha p)^T A(x^k + \alpha p) - (x^k + \alpha p)^T b, \\
&= \frac{1}{2}(x^k)^T Ax^k + \left(\frac{\alpha}{2}p^T Ax^k + \frac{\alpha}{2}(x^k)^T Ap\right) + \frac{\alpha^2}{2}p^T Ap - (x^k)^T b - \alpha p^T b, \\
&= \underbrace{\left[\frac{1}{2}(x^k)^T Ax^k - (x^k)^T b\right]}_{F(x^k)} + \alpha\left[p^T Ax^k - p^T b\right] + \frac{\alpha^2}{2}\left[p^T Ap\right].
\end{aligned}$$

To optimize we differentiate f with respect to α, set it to 0, and solve for α.

$$f'(\alpha) = -p^T(b - Ax^k) + \alpha p^T Ap = 0,$$
$$\Rightarrow \alpha = \frac{p^T(b - Ax^k)}{p^T Ap} = \frac{p^T r^k}{p^T Ap},$$

where $r^k = b - Ax^k$ is the residual of the kth iterate. This is the optimal α given the update rule $x^{k+1} = x^k + \alpha p$ and search vector $p \neq 0$. Note that since A is SPD, $p^T Ap > 0$. So α gives the minimum point along a given search direction.

Now consider choosing the search directions p. What is the **optimal** search direction? A vector pointing straight from x^k towards the solution x, so $p = x - x^k$.

This p would give the solution in **one step**. Unfortunately we do not know x! Therefore, we pick the search directions in a **locally optimal** manner. That is, we determine what direction reduces F as rapidly as possible **at the current point**.

We saw above that $f'(\alpha) = p^T(Ax^k - b) + \alpha p^T Ap$. This gives the rate of change at distance α along the search vector. So, $f'(0)$ gives rate of change along p at the **current position** (i.e., x^k). The idea is to pick p to make $f'(0)$ as negative as possible. This gives the direction of fastest decrease. We have that $f'(0) = p^T(Ax^k - b) = p^T \nabla F(x^k)$, so $f'(0)$ is minimized for

$$p = -\frac{\nabla F(x^k)}{\|\nabla F(x^k)\|}, \quad \text{(assuming we want unit } p, \|p\| = 1)$$
$$= \frac{b - Ax^k}{\|b - Ax^k\|} = \frac{r^k}{\|r^k\|}.$$

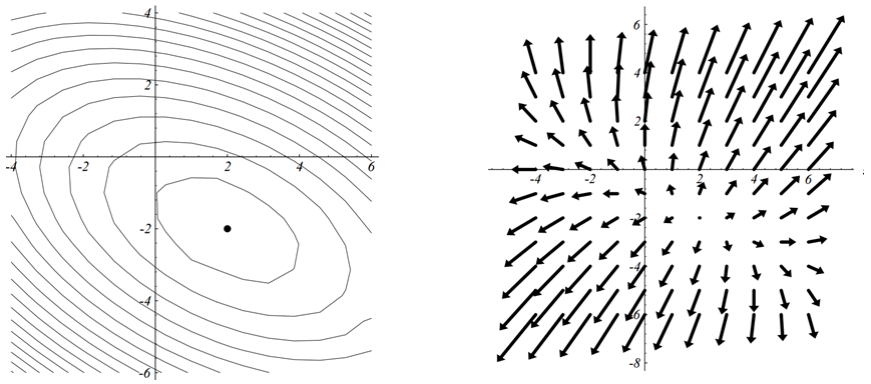
It is actually not necessary to normalize the search direction vector. Therefore we take $p = r^k$, which gives the iteration $x^{k+1} = x^k + \alpha r^k$ and optimal α as

$$\alpha = \frac{p^T r^k}{p^T Ap} = \frac{(r^k)^T r^k}{(r^k)^T Ar^k}.$$

Intuitively, we are finding the negative gradient of F (i.e., residual) and following it "downhill" as shown in Figure 9.23.

The steepest descent algorithm is given in Algorithm 9.5. For efficiency, instead of recomputing the residual from scratch at each step we can derive a simple **update rule**

$$r^{k+1} = b - Ax^{k+1},$$
$$= b - A(x^k + \alpha r^k),$$
$$= b - Ax^k - \alpha Ar^k,$$
$$= r^k - \alpha Ar^k.$$



Contours of F Gradient Vectors

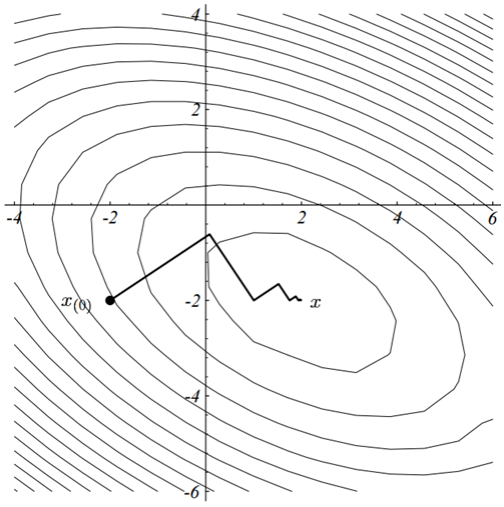
Figure 9.23: Example contours and gradient vectors of a function F.

The term Ar^k is already needed in the algorithm, so we can save one matrix-vector product per iteration by storing its result. Note that you can view steepest descent as a **nonlinear** iterative method with the iteration matrix $M = M^k = \frac{1}{\alpha_k} I$ that **changes** on each iteration.

Algorithm 9.5 : Steepest Descent

Given x_0, compute $r^0 = b - Ax^0$
for $k = 0, 1, 2, \ldots$
$\alpha^k = \frac{(r^k)^T r^k}{(r^k)^T A r^k}$
$x^{k+1} = x^k + \alpha^k r^k$
$r^{k+1} = r^k - \alpha^k A r^k$
end for

The steepest descent algorithm actually behaves quite poorly in terms of convergence. Since we assumed A was SPD steepest descent will indeed converge. However, steepest descent can be "shortsighted" and will often yield slow (zig-zag-like convergence towards the solution) as seen below.



For a SPD matrix A the error vectors $e^k = x^k - x^*$ for steepest descent satisfy

$$\|e^{k+1}\|_A \leq \left(\frac{\lambda_{max} - \lambda_{min}}{\lambda_{max} + \lambda_{min}}\right) \|e^k\|_A,$$

where $\|\cdot\|_A$ indicates the "A-norm" or **energy norm**: $\|x\|_A = \sqrt{x^T A x}$. For a proof of this result see Saad textbook, Section 5.3.1 or [Shewchuk 1994]. Next we will look at the **conjugate gradient** method, which chooses a different sequence of steps that can sometimes perform much better.

9.1.1 Towards the Conjugate Gradient Method

Recall the **steepest descent** method. We considered finding the x that gives the minimum of

$$F(x) = \frac{1}{2}x^T A x - b^T x, \quad x \in \mathbb{R}^n,$$

which also is the solution to $Ax = b$. We developed an iteration $x^{k+1} = x^k + \alpha p^k$ with:

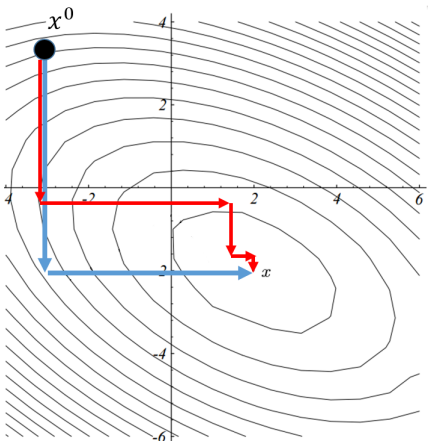
- search direction $p^k = r^k = b - Ax^k$,
- step length $\alpha = \frac{(r^k)^T p^k}{(p^k)^T A p^k} = \frac{(r^k)^T r^k}{(r^k)^T A r^k}$,

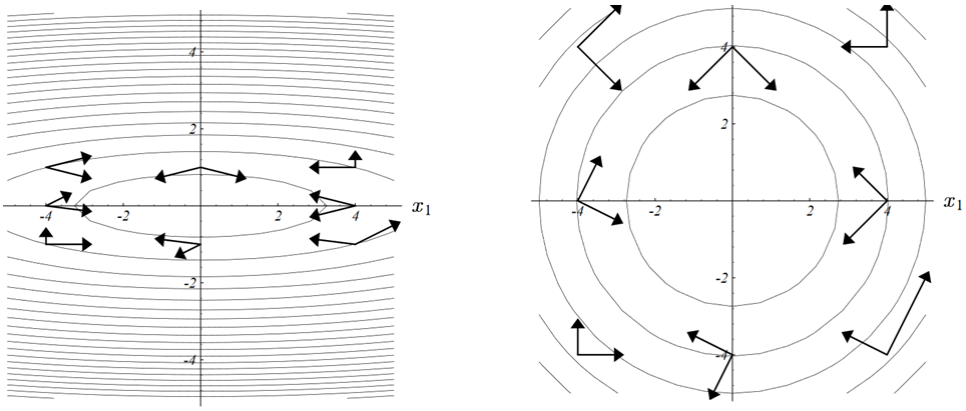
The search direction gives a locally fastest decrease, but not the globally "best" direction. This leads to zig-zag like convergence towards the solution. We now discuss how we can do better, starting with the **conjugate directions** method, then refining it to finally arrive at the **conjugate gradient** method.

9.2 Another Search Direction Idea

Imagine an approach that (somehow) finds the solution in the x_1 axis, then in the x_2 axis, ..., then in the x_n axis. It would complete in n iterations, touching each **orthogonal** axis **once**. Can we achieve something like this?

Choosing step $p^k = \text{axis}_k$ does not actually work. The lowest point (minimum value) along each axis in **not** the solution for that axis. The figure below depicts this idea. The blue vectors are what we would like. But, the red is the (bad) result if we use axes as search directions.





A-orthogonal vector pairs (are not orthogonal!)

Corresponding orthogonal vector pairs after stretching the space according to A

Figure 9.24: Visualization of the concept of A-orthogonality.

Choosing orthogonal directions (axes) and minimizing along them one at a time clearly does not work. Let us try something else, based on **A-orthogonality**. We first need to some definitions.

Definition 9.1. *Suppose A is SPD, then the A-inner product is defined as*

$$(p, q)_A = p^T Aq.$$

Definition 9.2. *The A-norm is given by*

$$\|p\|_A = \sqrt{(p, p)_A}.$$

Definition 9.3. *Two vectors p, q are A-orthogonal (or conjugate) if $(p, q)_A = 0$.*

Figure 9.24 visualizes vectors that are A-orthogonal. In general, A-orthogonal vectors are not orthogonal in the standard Euclidean space. The vectors are instead orthogonal when you transform to the new space by multiplying by A.

Intuitively, A-orthogonality is a kind of orthogonality that respects the properties of A. We build a new search direction method based on A-orthogonality. We choose each search direction to be A-orthogonal to **all** previous search directions. This will avoid searching redundant directions repeatedly. So we now need an algorithm to construct A-orthogonal vectors. We will use Gram-Schmidt A-orthogonalization.

9.2.1 Gram-Schmidt (A-)orthogonalization

Gram-Schmidt A-orthogonalization construct a set of A-orthogonal vectors, incrementally. Suppose the previous search directions $p^0, p^1, p^2, \ldots, p^{k-1}$ are all mutually A-orthogonal. Given a new proposed direction u^k, we convert it into a p^k that is A-orthogonal to all prior p^i. The idea is to subtract out the components of u^k that are **not** A-orthogonal to earlier p^i's, leaving behind a vector that **is**.

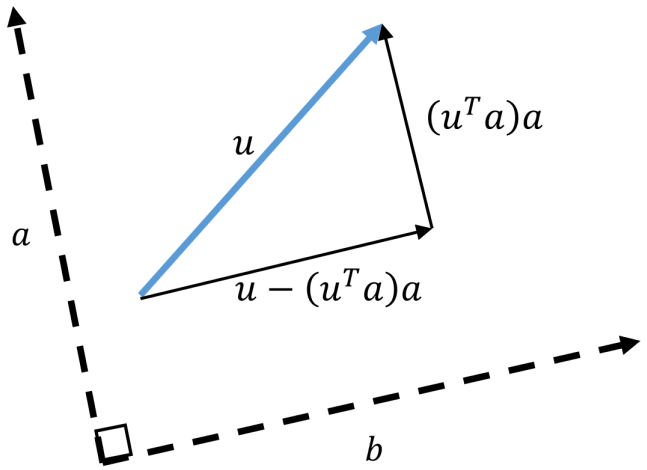


Figure 9.25: Orthogonalizing a vector u with respect to a.

Figure 9.25 gives a visualization of one step of Gram-Schmidt in 2D (for regular orthogonality). Consider some vector u, and a (previous) basis vector a. Then the vector $u - (u^T a)a$ will be orthogonal to a. Let's derive a Gram-Schmidt process to construct A-orthogonal vectors.

We start with a vector u^k and subtract all prior p^i components to form p^k, so

$$p^k = u^k + \sum_{i=0}^{k-1} \beta_i p^i.$$

Now we just need to find the coefficients β_i. For each p^j for $j = 0, \ldots, k-1$ use A-orthogonality against p^k:

$$\begin{aligned} 0 &= (p^k, p^j)_A, \\ &= \left(u^k + \sum_{i=0}^{k-1} \beta_i p^i, p^j \right)_A, \\ &= (u^k, p^j)_A + \sum_{i=0}^{k-1} \beta_i (p^i, p^j)_A. \end{aligned}$$

Earlier p^i's were mutually A-orthogonal, so $(p^i, p^j)_A = 0$ for $i \neq j$. Therefore,

$$\begin{aligned} 0 &= (u^k, p^j)_A + \beta_j (p^j, p^j)_A, \\ \Rightarrow \beta_j &= -\frac{(u^k, p^j)_A}{(p^j, p^j)_A}. \end{aligned} \tag{9.28}$$

Using this strategy we can construct an A-orthogonal set of p^k vectors spanning $\mathbb{R}^n$, when given input u^k's.

9.2.2 Conjugate Directions Method

To summarize, the conjugate directions method starts with a given set of vectors u^k spanning $\mathbb{R}^n$. We then A-orthogonalize them by Gram-Schmidt to get A-orthogonal search directions p^k. The same basic iteration as steepest descent is then performed to compute the solution x to $Ax = b$. The changes to the steepest descent iteration are:

- use new p^k as search directions (instead of residuals r^k),
- use our original step length expression $\alpha = \frac{(r^k)^T p^k}{(p^k)^T A p^k}$ (i.e., not $\frac{(r^k)^T r^k}{(p^k)^T A r^k}$).

The drawbacks of conjugate directions (w/ Gram-Schmidt) are as follows.

- With respect to memory we need to keep **all** prior search vectors p^i.
- In terms of computational cost we need to perform complete Gram-Schmidt at each step, which takes $O(n^3)$ flops.
- There is also the lingering question about how to choose the input (non-A-orthogonal) u^k vectors?
- A fun fact is that if the proposed search vectors u^k at each step (before A-orthogonalization) are the axes, this gives Gaussian Elimination again!

9.3 Conjugate Gradient Method

The conjugate gradient method is a variation on the conjugate directions method that improves in two ways:

1. choose the vectors u^k at each step to be the residual r^k (to be A-orthogonalized),
2. carefully exploit (A-)orthogonality to avoid storing all prior search vectors.

Item 1. immediately turns our earlier Gram-Schmidt process into

$$p^k = r^k + \sum_{i=0}^{k-1} \beta_i p^i = r^k - \sum_{i=0}^{k-1} \frac{(r^k, p^i)_A}{(p^i, p^i)_A} p^i, \tag{9.29}$$

which gives the first version of the conjugate gradient algorithm in Algorithm 9.6.

Algorithm 9.6 : Conjugate Gradient algorithm (version 1.0)

1: $x^0 =$ initial guess
2: $r^0 = b - Ax^0$
3: **for** $k = 0, 1, 2, \ldots, n-1$
4: Compute p^k as described above (Gram-Schmidt)
5: $x^{k+1} = x^k + \alpha_k p^k$
6: $r^{k+1} = r^k - \alpha_k A p^k$
7: **end for**
8: (Special case: use $\beta_{-1} = 0$ on the 0th iteration)

Note that lines 5 and 6 are the same as steepest descent. That is, given the direction

p^k pick optimal α_k, then update solution x and residual r. Recall that $\alpha_k = \frac{(r^k, p^k)}{(p^k, p^k)_A}$ and $r^{k+1} = b - Ax^{k+1}$. This first version is still costly, do not implement this version!

9.3.1 Efficient Conjugate Gradient Method

Let's work towards a more efficient algorithm. We want concise recursive expressions for

- the step lengths α,
- step directions p, and
- Gram-Schmidt coefficients β.

We will need to consider the spaces involved and their relationships, as well as, repeatedly exploit orthogonality and A-orthogonality.

Consider the space spanned by vectors p^i for $i = 0, \ldots, k-1$.

$$\begin{aligned} & \text{span}\{p^0, p^1, \ldots, p^{k-1}\} \\ = \; & \text{span}\{r^0, r^1, \ldots, r^{k-1}\} \quad \text{(by Gram-Schmidt)} \\ = \; & \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-1}r^0\} \quad \text{(See Theorem 9.3).} \end{aligned}$$

A space constructed this way (powers of A times a vector) is called a (k-dimensional) **Krylov subspace**, denoted $\mathcal{K}_k(A, r^0)$.

Theorem 9.3.

$$span\{r^0, r^1, \ldots, r^{k-1}\} = span\{r^0, Ar^0, A^2r^0, \ldots, A^{k-1}r^0\}.$$

Proof.

- The proof is by induction on $k \geq 1$ ($k = 0$ makes no sense).
- Base ($k = 1$)
  - $\text{span}\{r^0\} = \text{span}\{r^0\}$ is trivial.
- Induction ($k > 1$)
  - The induction hypothesis is that $\text{span}\{r^0, r^1, \ldots, r^{k-2}\} = \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-2}r^0\}$.
  - Recall also that from our setup, Gram-Schmidt,

$$\text{span}\{p^0, p^1, \ldots, p^{k-1}\} = \text{span}\{r^0, r^1, \ldots, r^{k-1}\}$$

  - We have this fact, which will be needed throughout the remainder of the argument:

$$r^{k+1} = r^k - \alpha_k A p^k$$

  - It suffices to prove the following two facts:
    1. $\underbrace{r^{k-1} \in \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-1}r^0\}}_{*}$

$$\begin{aligned} r^{k-2} \quad & \in \quad \text{span}\{r^0, r^1, \ldots, r^{k-2}\} \\ & \underbrace{\subseteq}_{\text{I.H.}} \quad \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-2}r^0\} \\ & \subseteq \quad \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-1}r^0\} \end{aligned}$$

* Also
$$p^{k-2} = r^{k-2} + \sum_{i=0}^{k-3} \beta_i p^i.$$
* Note that $\sum_{i=0}^{k-3} \beta_i p^i \in \text{span}\{r^0, r^1, \ldots, r^{k-3}\}$, by Gram-schmidt.
* So $p^{k-2} \in \text{span}\{r^0, r^1, \ldots, r^{k-2}\} \underbrace{\subseteq}_{\text{I.H.}} \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-2}r^0\}$.
* Thus $\alpha_{k-2}Ap^{k-2} \in \text{span}\{Ar^0, A^2r^0, \ldots, A^{k-1}r^0\} \subseteq \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-1}r^0\}$.
* Hence $r^{k-1} \in \text{span}\{r^0, Ar^0, A^2r^0, \ldots, A^{k-1}r^0\}$.

2. $\underline{A^{k-1}r^0 \in \text{span}\{r^0, r^1, \ldots, r^{k-1}\}}$
* Re-arranging $r^{k+1} = r^k - \alpha_k Ap^k$ (assuming each $\alpha_i \neq 0$; if some $\alpha_i = 0$, then $x^{i+1} = x^i$ from this point on) gives
$$Ap^k = \frac{r^k - r^{k+1}}{\alpha_k}.$$
* We have $A^{k-2}r^0 \in \text{span}\{r^0, r^1, \ldots, r^{k-2}\} = \text{span}\{p^0, p^1, \ldots, p^{k-2}\}$.
* Write $A^{k-2}r^0 = \sum_{i=0}^{k-2} a_i p^i$.
* Therefore
$$\begin{aligned} A^{k-1}r^0 &= A\left(A^{k-2}r^0\right) \\ &= \sum_{i=0}^{k-2} a_i Ap^i \\ &= \sum_{i=0}^{k-2} \frac{a_i(r^i - r^{i+1})}{\alpha_i} \\ &\in \text{span}\{r^0, r^1, \ldots, r^{k-1}\}. \end{aligned}$$

□

It can also be shown (See Theorem 9.3.1) that $r^k \perp \text{span}\{p^0, p^1, \ldots, p^{k-1}\} = \text{span}\{r^0, r^1, \ldots, r^{k-1}\}$, i.e., $(r^k, r^j) = 0$ for $j = 0, 1, \ldots, k-1$. That is, the current residual is orthogonal to the prior search directions and residuals. Currently, the **step length** is computed as $\alpha_k = \frac{(r^k, p^k)}{(p^k, p^k)_A}$. Observe however that

$$\begin{aligned} (r^k, p^k) &= \left(r^k, r^k + \sum_{i=0}^{k-1} \beta_i p^i \right), \quad \text{(since } (r^k, p^i) = 0 \text{ for } i < k\text{)} \\ &= (r^k, r^k), \end{aligned} \tag{9.30}$$

which gives a new way of computing α_k as

$$\alpha_k = \frac{(r^k, r^k)}{(p^k, p^k)_A}.$$

To make computing the search direction more efficient we need the identity

$$(r^k, p^i)_A = 0 \text{ for } i = 0, 1, \ldots, k-2. \tag{9.31}$$

Proof. We know that $p^i \in \text{span}\{p^0, \ldots, p^i\} = \text{span}\{r^0, Ar^0, \ldots, A^i r^0\}$. So

$$Ap^i \in \text{span}\{Ar^0, A^2 r^0, \ldots, A^{i+1} r^0\},$$

just by left multiplying all the vectors by A. This space is a subset of the $\text{span}\{r^0, Ar^0, \ldots, A^{i+1} r^0\}$ since it is only missing r^0. Hence, $Ap^i \in \text{span}\{r^0, r^1, \ldots, r^{i+1}\}$ by our earlier relationships.

But, we also have that $r^k \perp Ap^i \in \text{span}\{r^0, r^1, \ldots, r^{i+1}\}$ for $i+1 \leq k-1$. Therefore, $(r^k, p^i)_A = (r^k, Ap^i) = 0$ for $i \leq k-2$ since Ap^i was in the span and r^k was orthogonal to it. □

Now we can construct **search direction** p^k without storing all prior p^i. Starting from (9.29), Gram-Schmidt gives

$$\begin{aligned} p^k &= r^k - \sum_{i=0}^{k-1} \frac{(r^k, p^i)_A}{(p^i, p^i)_A} p^i, \\ &= r^k - \frac{(r^k, p^{k-1})_A}{(p^{k-1}, p^{k-1})_A} p^{k-1}, \end{aligned}$$

by (9.31). Hence, only the current residual and previous step direction are needed to compute p^k. This saves us storage and flops.

To arrive at the standard conjugate gradient method we further simplify β_{k-1}. Equation (9.28) gave $\beta_{k-1} = -\frac{(u^k, p^{k-1})_A}{(p^{k-1}, p^{k-1})_A}$. Later replacing u^k by r^k gave

$$\beta_{k-1} = -\frac{(r^k, p^{k-1})_A}{(p^{k-1}, p^{k-1})_A}. \tag{9.32}$$

In the numerator we have $(r^k, p^{k-1})_A$. By the residual update rule, $r^k = r^{k-1} - \alpha_{k-1} A p^{k-1}$, so applying $(r^k, \cdot)$ to this rule, we get that

$$(r^k, r^k) = \underbrace{(r^k, r^{k-1})}_{=0 \text{ since the } r\text{'s are orthogonal}} - \alpha_{k-1}(r^k, Ap^{k-1}),$$

since the r's are orthogonal. Rearranging, we get that the numerator is

$$(r^k, p^{k-1})_A = (r^k, Ap^{k-1}) = -\frac{1}{\alpha_{k-1}}(r^k, r^k).$$

Now consider the denominator $(p^{k-1}, p^{k-1})_A$. We have that $(r^k, p^{k-1}) = 0$ by orthogonality, and then applying the residual update gives

$$0 = (r^{k-1}, p^{k-1}) - \alpha_{k-1}(Ap^{k-1}, p^{k-1}) = (r^{k-1}, r^{k-1}) - \alpha_{k-1}(p^{k-1}, p^{k-1})_A,$$

because earlier (9.30) we showed $(r^{k-1}, p^{k-1}) = (r^{k-1}, r^{k-1})$. Rearranging, gives the denominator as $(p^{k-1}, p^{k-1})_A = \frac{1}{\alpha_{k-1}}(r^{k-1}, r^{k-1})$. Combining the numerator and denominator we have

$$\begin{aligned}
\beta_{k-1} &= -\frac{(r^k, p^{k-1})_A}{(p^{k-1}, p^{k-1})_A}, \\
&= -\left(-\frac{(r^k, r^k)}{\cancel{\alpha_{k-1}}}\right)\left(\frac{\cancel{\alpha_{k-1}}}{(r^{k-1}, r^{k-1})}\right), \\
&= \frac{(r^k, r^k)}{(r^{k-1}, r^{k-1})}.
\end{aligned}$$

The more efficient version of the conjugate gradient method is given in Algorithm 9.7 based on the simplifications above.

Algorithm 9.7 : Conjugate Gradient (version 2.0)

$x^0 =$ initial guess	▷ Initialize Solution
$r^0 = b - Ax^0$	▷ Initialize Residual
for $k = 0, 1, 2, \ldots, n-1$	▷ Loop (explore n dimensional space)
$\beta_{k-1} = (r^k, r^k)/(r^{k-1}, r^{k-1})$	▷ Find β coeff for p^{k-1} component in r^k
$p^k = r^k + \beta_{k-1}p^{k-1}$	▷ Remove it to get A-orthogonal search direction p^k
$\alpha_k = (r^k, r^k)/(p^k, Ap^k)$	▷ Determine step length α along p^k
$x^{k+1} = x^k + \alpha_k p^k$	▷ Update solution
$r^{k+1} = r^k - \alpha_k A p^k$	▷ Update residual
end for	
(Special case: use $\beta_{-1} = 0$ on the 0th iteration)	

Now conjugate gradient just needs 1 matrix-vector multiply and 2 inner-products per step:

- matrix-vector multiply Ap^k,
- dot products (r^k, r^k) and (p^k, Ap^k).

9.3.2 Error Behaviour

Note that at most n A-orthogonal vectors are needed to span $\mathbb{R}^n$. Therefore conjugate gradient will terminate in (at most) n steps with an exact solution (under exact arithmetic). At each iteration, the current conjugate gradient solution's error has the minimum A-norm within the subspace it has already explored,i.e.,

$$x^k = \underset{x \in \mathcal{K}_k}{\arg\min} \|e^k\|_A^2 = \underset{x \in \mathcal{K}_k}{\arg\min} \|x^k - x^*\|_A^2.$$

This is because **at each iteration, conjugate gradient zeroes out one of the error components**. To see this let $e^i = x^* - x^i$, where x^* is the true solution. We therefore have $r^i = Ae^i$. Now express e^0 as a linear combination of search directions

$$e^0 = \sum_{j=0}^{n-1} \delta_j p^j, \quad \text{for coefficients } \delta_j.$$

Left multiplying by $(p^k)^T A$ (to exploit A-orthogonality) gives

$$\begin{aligned}
(p^k)^T A e^0 &= \sum_{j=0}^{n-1} \delta_j (p^k)^T A p^j, \\
(p^k, e^0)_A &= \sum_{j=0}^{n-1} \delta_j (p^k, p^j)_A, \\
&= \delta_k (p^k, p^k)_A, \qquad \text{(by A-orthogonality)} \\
\Rightarrow \delta_k &= \frac{(p^k, e^0)_A}{(p^k, p^k)_A}.
\end{aligned}$$

Continuing the calculation with the generic $e^k = e^0 - \sum_{i=0}^{k-1} \alpha_i p^i$ (because the iteration is $x^{k+1} = x^k + \alpha_k p^k$) gives

$$\begin{aligned}
\delta_k &= \frac{(p^k, e^0)_A}{(p^k, p^k)_A} \\
&= \frac{(p^k, e^k + \sum_{i=0}^{k-1} \alpha_i p^i)_A}{(p^k, p^k)_A} \\
&= \frac{(p^k, e^k)_A}{(p^k, p^k)_A},
\end{aligned}$$

by the A-orthogonality of the p^i's. But recall that

$$\begin{aligned}
\alpha_k &= \frac{(p^k)^T r^k}{(p^k, p^k)_A} \\
&= \frac{(p^k)^T A e^k}{(p^k, p^k)_A}, \text{ since } r^k = A e^k \\
&= \frac{(p^k, e^k)_A}{(p^k, p^k)_A}.
\end{aligned}$$

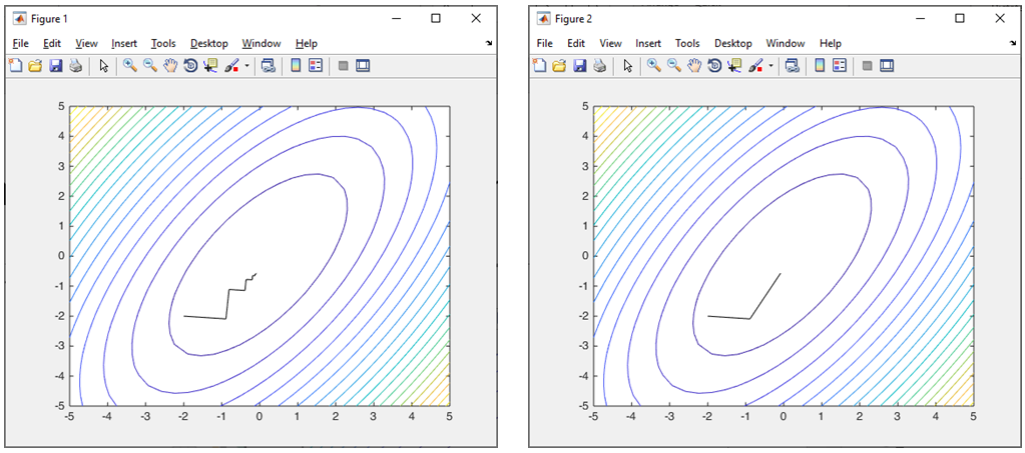
Hence, $\alpha_k = \delta_k$, so conjugate gradient zeroes out one component of the error at each iteration:

$$e^i = e^0 - \sum_{j=0}^{i-1} \alpha_j p^j = \left(\sum_{j=0}^{n-1} \delta_j p^j - \sum_{j=0}^{i-1} \delta_j p^j \right) = \sum_{j=i}^{n-1} \delta_j p^j. \tag{9.33}$$

After n steps, all the components of e^0 will be gone.

Theorem 9.3.1. *With our established notation, $r^j \perp$ span $\{p^0, \ldots, p^{j-1}\}$ = span $\{r^0, \ldots, r^{j-1}\}$.*

Proof.
- Equation (9.33) gives $e^i = \sum_{j=i}^{n-1} \delta_j p^j$.
- Re-write to replace indices: $e^j = \sum_{k=j}^{n-1} \delta_k p^k$.



Steepest Descent Conjugate Gradient

Figure 9.26: Comparison of steepest descent and conjugate gradient methods in $\mathbb{R}^2$.

- Left multiply by $(p^i)^T A$:

$$\begin{aligned} (p^i)^T A e^j &= \sum_{k=j}^{n-1} \delta_k (p^i)^T A p^k \\ (p^i)^T r^j &\underbrace{=}_{r^j = Ae^j} \sum_{k=j}^{n-1} \delta_k (p^i, p^k)_A \end{aligned} \tag{9.34}$$

- Equation (9.34) holds for all i.
- Our desired result is for $i < j$. For such i, by the A-orthogonality of the p^ks, equation becomes $(p^i)^T r^j = 0$.

□

Consider using conjugate gradient on the following example:

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} x = \begin{bmatrix} 2 \\ -8 \end{bmatrix}$$

starting from

$$x^0 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}.$$

Conjugate gradient converges in 2 steps since we are in $\mathbb{R}^2$ as shown in Figure 9.26.

If you are interested in more details, our discussion borrowed heavily from Shewchuk's notes on conjugate gradient.

An Introduction to
the Conjugate Gradient Method
Without the Agonizing Pain

Edition $1\frac{1}{4}$

Jonathan Richard Shewchuk

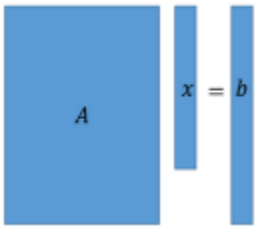
August 4, 1994

10 Lecture 10: Least Squares Problems

Outline

1. Least Squares
2. Method 1: Normal Equations
3. Method 2: QR Factorization
   (a) QR for Least Squares

This lecture discusses solving problems with more equations (rows) than variables. The problem is solved "as well as possible" since the system is over-determined (more equations than necessary). That is, least squares problems solve the equation $Ax = b$, where A is taller that it is wide. We end up with a solution that is over-determined.



10.1 Least Squares

Least squares problems were first posed and formulated by Gauss around 1795 (though published first by Legendre 1805). The method of least squares is often found in applications, e.g., finding a line or polynomial to fit a large set of data/observations.

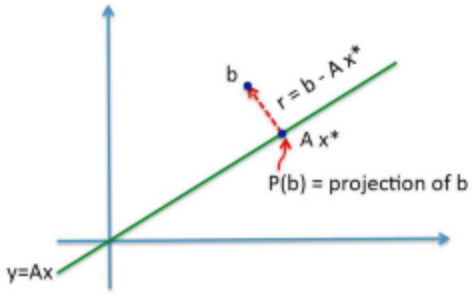
Mathematically, we want to minimize the magnitude of the **residual vector** $r = b - Ax$.

$$\min_{x \in \mathbb{R}^n} \|b - Ax\|_2^2, \text{ for } A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, m \geq n.$$

In general, we can not achieve $r = 0$. Notice this differs from minimizing (square, non-singular) linear systems, for which we use:

$$F(x) = \frac{1}{2} x^T A x - x^T b.$$

A geometric interpretation of least squares problems is as follows. We find the closest point to b on the $y = Ax$ hyperplane. In other words, find the "projection" of b onto the range of A. Notice that residual vector r is orthogonal to y (see figure below).



Theorem 10.1. *Let $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, m \geq n$, and A full rank. A vector $x \in \mathbb{R}^n$ minimizes*

$$\|r\|_2^2 = \|b - Ax\|_2^2$$

if and only if $r \perp range(A)$.

Task: Collin to add the proof of Theorem 10.1 into the Lecture Notes.

Theorem 10.1 implies

$$\begin{aligned} r^T A &= \vec{0} \\ \Leftrightarrow A^T r &= \vec{0} \\ \Leftrightarrow A^T(b - Ax) &= \vec{0} \\ \Leftrightarrow A^T b &= A^T A x. \end{aligned}$$

High Level View of Where We Are Going

- By end of Lecture 10: existence of Q, computed from A.
- By end of Lecture 11: existence of R, computed from A, Q.
- Not proved in Slides / Notes yet: uniqueness of Q, R, given A.
  - A Brief Note From Wikipedia: If A has full rank, then the QR factorization is unique, provided we require the diagonal elements of R to be positive.

The equations $A^T Ax = A^T b$ are known as the **normal equations**. Solving the normal equations (which is a square system) gives the **least squares solution**. This motivates the definition of the pseudo-inverse.

Definition 10.1. *$A^+ = (A^T A)^{-1} A^T$ is called the (Moore-Penrose) **pseudo-inverse** of A.*

The least squares solution satisfies

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

Fact: Any perturbation of x from this solution yields a higher residual norm. To see this

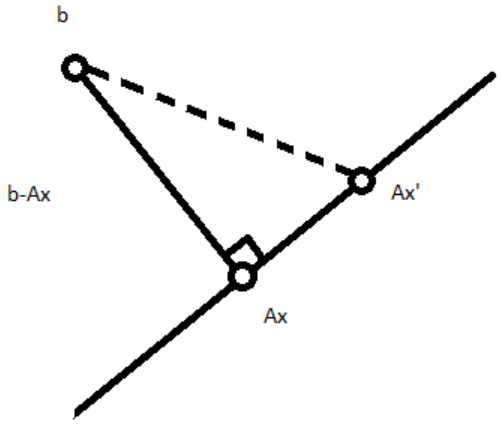
let $x' = x + e$, where x is the least squares solution and e is some perturbation. Then,

$$\begin{aligned}
& \|b - Ax'\|_2^2 \\
= \; & (b - Ax')^T (b - Ax'), \\
= \; & (b - Ax - Ae)^T (b - Ax - Ae), \\
= \; & (b - Ax)^T (b - Ax) - 2(Ae)^T (b - Ax) + (Ae)^T (Ae), \\
= \; & (b - Ax)^T (b - Ax) - 2e^T A^T (b - Ax) + \|Ae\|_2^2, \\
= \; & \|b - Ax\|_2^2 + \|Ae\|_2^2 - 2e^T (A^T b - A^T Ax), \quad (x \text{ is the LS soln})
\end{aligned}$$

where $A^T b - A^T Ax \to 0$.

$\Rightarrow \|b - Ax'\|_2^2 > \|b - Ax\|_2^2$ for any $e \neq 0$.

Thus, any other point x' yields a larger residual, as seen geometrically below.



We will consider the following two solution strategies (for now):

1. Normal equations: Find and solve normal equations $A^T Ax = A^T b$ to find x (e.g., via Cholesky).
   (a) See Theorem 10.1.1 for an explanation of why if A has full rank, then $A^T A$ is SPD, so that it has a Cholesky factor.
2. QR Factorization: Construct a factorization $A = QR$ (with certain properties) and instead solve $Rx = Q^T b$ for x.

Theorem 10.1.1. *If A has full rank, then $A^T A$ is SPD.*

Proof.
- Symmetry is clear, because $(A^T A)^T = A^T (A^T)^T = A^T A$.
- For any column vector, v, we have $v^T A^T Av = (Av)^T (Av) = (Av, Av) \geq 0$, therefore $A^T A$ is positive semi-definite.
- In particular, if A has full rank, then $A^T A$ is positive definite.

□

10.2 Method 1: Normal Equations

In this subsection we will look at the normal equations solution. In the next subsection will discuss QR factorizations.

We solve $A^TAx = A^Tb$ directly by computing the Cholesky factorization $A^TA = GG^T$, with G lower triangular. Then, we compute x by forward/backward solves. The complexity of this approach has flops to form $A^TA \approx mn^2$ and $GG^T \approx \frac{1}{3}n^3$. Therefore, the total flops $\approx mn^2 + \frac{1}{3}n^3$.

Consider the application of polynomial fitting with least squares. We want to find a polynomial of the form:

$$p(t) = a_0 + a_1t + a_2t^2 + \cdots + a_{n-1}t^{n-1},$$

that best fits the set of 2D points given by (t_i, y_i) for $i = 1, \ldots, m$, with $m > n$. Each data point yields one equation. The coefficients $a_0, a_1, \ldots, a_{n-1}$ are the unknowns. The matrix problem is

$$\begin{bmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^{n-1} \\ 1 & t_2 & t_2^2 & \cdots & t_2^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & t_m & t_m^2 & \cdots & t_m^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}.$$

As a concrete example we are given the set of (t_i, y_i) points $\{(0,0), (0,-1), (2,1), (2,0), (4,2), (4,1)\}$. We want to find the best fit line $y = a_0 + a_1t$ using normal equations. We have,

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 2 \\ 1 & 2 \\ 1 & 4 \\ 1 & 4 \end{bmatrix}, \qquad x = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, \qquad b = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \\ 2 \\ 1 \end{bmatrix}.$$

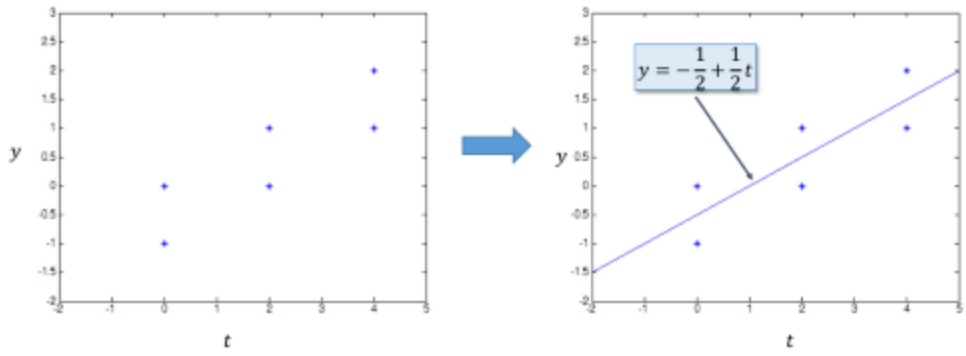
To obtain the solution we construct

$$A^TA = \begin{bmatrix} 6 & 12 \\ 12 & 40 \end{bmatrix}, \quad \text{and } A^Tb = \begin{bmatrix} 3 \\ 14 \end{bmatrix}.$$

Solving $A^TAx = A^Tb$ gives

$$\begin{aligned} a_0 &= -\frac{1}{2}, \\ a_1 &= \frac{1}{2}, \text{ so} \\ p &= -\frac{1}{2} + \frac{1}{2}t. \end{aligned}$$

The figure below shows the points and the line of best fit given by p.



10.3 Method 2: QR Factorization

The previous subsection discussed the first method for solving least squares problems, i.e., via the normal equations. This lecture discusses a second approach using QR factorization. The QR factorization decomposes a matrix A into an orthogonal matrix Q and a triangular matrix R.

Some properties of **orthogonal matrices** are discussed next. They are needed to give a theorem for the existence of a QR factorization at the end of this section.

Definition 10.2. *A square matrix Q is **orthogonal** if $Q^{-1} = Q^T$ (i.e., $Q^TQ = QQ^T = I$).*

Theorem 10.2. *If Q is orthogonal, then $\|Qx\|_2 = \|x\|_2$.*

Proof. $\|Qx\|^2 = (Qx)^T(Qx) = x^TQ^TQx = x^Tx = \|x\|^2$. □

Remark: Permutation matrices, first seen during matrix re-ordering, are examples of orthogonal matrices.

Note that left multiplication by an orthogonal Q corresponds to

$$\begin{cases} \text{rotation if } \det(Q) = 1, \\ \text{reflection if } \det(Q) = -1. \end{cases}$$

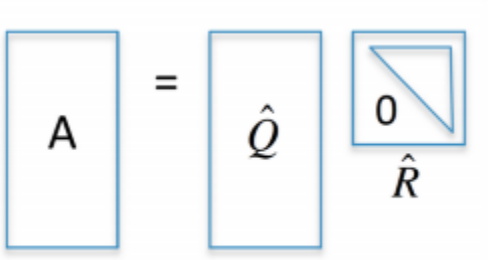
Definition 10.3. *A set of vectors are **orthonormal** if they are mutually orthogonal and each vector has norm $= 1$.*

For example, the columns of an orthogonal matrix are orthonormal. The columns of an $n \times n$ orthogonal matrix, Q, form an orthonormal basis of $\mathbb{R}^n$. Be careful not to confuse the following:

1. orthogonal **vectors** need not be unit length,
2. an orthogonal **matrix** has columns that are **orthonormal**.

The following theorem gives the existence of a QR factorization.

Theorem 10.3. *Suppose $A \in \mathbb{R}^{m\times n}$ has full rank. Then there exists a unique matrix $\hat{Q} \in \mathbb{R}^{m\times n}$ satisfying $\hat{Q}^T\hat{Q} = I$ (i.e., with orthonormal columns) and a unique upper triangular matrix $\hat{R} \in \mathbb{R}^{n\times n}$ with positive diagonals ($r_{i,i} > 0$) such that $A = \hat{Q}\hat{R}$.*



Note that because $\hat{Q}$ is non-square in general here, it is **not** necessarily an orthogonal matrix.

Example: Let

$$\hat{Q} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 \end{bmatrix},$$

so that

$$\hat{Q}^T = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \end{bmatrix}.$$

Then

$$\begin{aligned} \hat{Q}\hat{Q}^T &= \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &\neq I_3. \end{aligned}$$

10.3.1 QR for Least Squares

Consider the least squares problem:

$$\min_x \|Ax - b\|^2.$$

We will try to make $\|Ax - b\|^2$ as small as possible by re-expressing $Ax - b$ in terms of the QR factorization, and adjusting x. Only x can be adjusted because A and b are defined (given) by the problem.

Consider separating $Ax - b$ into orthogonal components using $\hat{Q}\hat{R}$. That is, using $\hat{Q}$ and $\hat{R}$, split $Ax - b$ into two orthogonal components:

$$\begin{aligned} Ax - b &= \hat{Q}\hat{R}x - b, \\ &= \hat{Q}\hat{R}x - (\underbrace{\hat{Q}\hat{Q}^T - \hat{Q}\hat{Q}^T}_{=0} + I)b, \\ &= \underbrace{\hat{Q}(\hat{R}x - \hat{Q}^T b)}_{} - \underbrace{(I - \hat{Q}\hat{Q}^T)b}_{}. \end{aligned}$$

We claim that these two vectors are orthogonal. These vectors are orthogonal if and only if their inner product is zero. We can verify that the inner product is zero as follows:

$$\begin{aligned} & \left[\hat{Q}(\hat{R}x - \hat{Q}^T b)\right]^T \left[(I - \hat{Q}\hat{Q}^T)b\right] \\ = \; & (\hat{R}x - \hat{Q}^T b)^T \hat{Q}^T (I - \hat{Q}\hat{Q}^T)b, \\ = \; & (\hat{R}x - \hat{Q}^T b)^T (\hat{Q}^T - \underbrace{\hat{Q}^T\hat{Q}}_{=I}\hat{Q}^T)b, \quad (\hat{Q}\text{'s columns orthonormal}) \\ = \; & (\hat{R}x - \hat{Q}^T b)^T (\underbrace{\hat{Q}^T - \hat{Q}^T}_{=0})b = 0. \end{aligned}$$

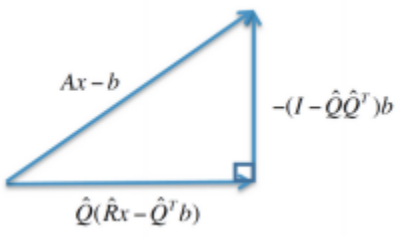
Note that since $\hat{Q}$ is not square, $\hat{Q}\hat{Q}^T \neq I$, but $\hat{Q}^T\hat{Q} = I$.

The goal of the least squares problem is to minimize the size of $r = b - Ax$. We can only modify x to make the vector $r = b - Ax$ as short as possible. By Pythagoras we have

$$\begin{aligned} \|Ax - b\|^2 &= \|\hat{Q}(\hat{R}x - \hat{Q}^T b)\|^2 + \|(I - \hat{Q}\hat{Q}^T)b\|^2, \\ &= \underbrace{\|(\hat{R}x - \hat{Q}^T b)\|^2}_{\text{select } x \text{ to minimize}} + \underbrace{\|(I - \hat{Q}\hat{Q}^T)b\|^2}_{\text{can't adjust}}. \end{aligned}$$

The orthogonal components can be visualized as shown in the figure below. The norm is minimized when the first term is 0. So the least squares solution is

$$\hat{R}x = \hat{Q}^T b \qquad \Rightarrow \qquad x = \hat{R}^{-1}\hat{Q}^T b.$$



We can also relate this solution to pseudoinverse and normal equations as follows.

Pseudoinverse

The pseudoinverse written in terms of QR factors is

$$\begin{aligned} A^+ &= (A^TA)^{-1}A^T \\ &= ((\hat{Q}\hat{R})^T(\hat{Q}\hat{R}))^{-1}(\hat{Q}\hat{R})^T \\ &= (\hat{R}^T \underbrace{\hat{Q}^T\hat{Q}}_{=I} \hat{R})^{-1}(\hat{R}^T\hat{Q}^T) \\ &= (\hat{R}^T\hat{R})^{-1}\hat{R}^T\hat{Q}^T \\ &= \hat{R}^{-1}\underbrace{(\hat{R}^T)^{-1}\hat{R}^T}_{=I}\hat{Q}^T \\ &= \hat{R}^{-1}\hat{Q}^T. \end{aligned}$$

Normal Equations

Then consider the normal equations

$$\begin{aligned} A^TAx &= A^Tb \\ \Leftrightarrow (\hat{R}^T\hat{Q}^T)(\hat{Q}\hat{R})x &= (\hat{R}^T\hat{Q}^T)b, \\ \hat{R}^T\hat{R}x &= (\hat{R}^T\hat{Q}^T)b, \\ \hat{R}x &= \hat{Q}^Tb, \\ x &= \hat{R}^{-1}\hat{Q}^Tb. \end{aligned}$$

Two Sizes of QR Factorization So far we have only seen the **reduced ("economy size")** version of QR factorization. Specifically, $A = \hat{Q}\hat{R}$ where $\hat{Q} \in \mathbb{R}^{m\times n}$ and $\hat{R} \in \mathbb{R}^{n\times n}$. The "full" version of QR adds extra orthonormal columns to make Q square (and thus makes Q a true orthogonal matrix). Extra zero rows in R are also added to match the dimensions (See below).

A full QR factorization is achieved by appending $m-n$ additional orthonormal columns to Q. First define

$$\hat{Q}_{m-n} \equiv \begin{bmatrix} q_{n+1} & q_{n+2} & \cdots & q_m \end{bmatrix}.$$

Then we have

$$\big[A\big]_{m\times n} = \big[\hat{Q}|\hat{Q}_{m-n}\big]_{m\times m}\begin{bmatrix}\hat{R}\\0\end{bmatrix}_{m\times n},$$

which is often useful for theoretical purposes.

Computing the (reduced) QR Factorization To compute the reduced QR factorization we let

$$A = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \dots & a_n \\ | & | & & | \end{bmatrix},$$

where a_i are the columns of A. The columns span the column space of the matrix. So we want to find a set of orthonormal column vectors, $\{q_i\}$ spanning the same space. That is, $\text{span}\{q_1, q_2, \dots, q_j\} = \text{span}\{a_1, a_2, \dots, a_j\}$, for $j = 1, \dots, n$.

For this we can use Gram-Schmidt orthogonalization. We already saw a variant of Gram-Schmidt earlier for constructing A-orthogonal search directions in conjugate gradient. The same general idea is used here:

- use columns of A as proposed vectors to be orthogonalized into Q,
- build each new vector q_j by orthogonalizing a_j with respect to all previous q vectors, $\{q_1, q_2, \ldots, q_{j-1}\}$, and then normalize q_j.

The entries for R can be calculated once we know Q by considering the general form

$$\begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ q_1 & q_2 & \cdots & q_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{bmatrix}.$$

Written out fully, this is

$$\begin{aligned} a_1 &= r_{11}q_1, \\ a_2 &= r_{12}q_1 + r_{22}q_2, \\ a_3 &= r_{13}q_1 + r_{23}q_2 + r_{33}q_3, \\ &\vdots \\ a_n &= r_{1n}q_1 + r_{2n}q_2 + \cdots + r_{nn}q_n. \end{aligned}$$

The next lecture will discuss this approach (using Gram-Schmidt) of computing the QR factorization in more detail.

11 Lecture 11: Gram-Schmidt Orthogonalization

Outline

1. QR factorization via Gram-Schmidt
   (a) Orthonormalization for Q
   (b) Upper Triangular Matrix R
2. Modified Gram-Schmidt
3. Complexity of Gram-Schmidt

We have already seen a variation of Gram-Schmidt orthogonalization in constructing A-orthogonal search directions for the conjugate gradient method (see Lecture 09). The same general idea applies here to construct the QR factorization:

- Use columns of A as proposed vectors to be orthogonalized into Q.
- Build each new vector q_j by orthogonalizing a_j with respect to all previous q vectors, $\{ q_1, q_2, \ldots, q_{j-1} \}$, and then normalizing it.

11.1 QR factorization via Gram-Schmidt

11.1.1 Orthonormalization for Q

For a vector a_j, we can orthogonalize it against all previous vectors q_i for $i = 1, \ldots, j-1$, using

$$v_j = a_j - (q_1^T a_j)q_1 - (q_2^T a_j)q_2 - \cdots - (q_{j-1}^T a_j)q_{j-1}. \tag{11.35}$$

This removes a_j's components in the orthogonal directions constructed so far. We can then directly normalize, giving us

$$q_j = \frac{v_j}{\|v_j\|_2}.$$

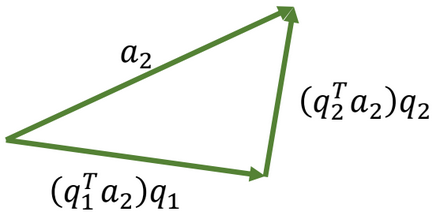
Equation (11.35) was derived previously in Lecture 09 assuming the form $v_j = a_j + \sum_{i=1}^{j-1} \beta_i q_i$. To recap, the coefficients β_j were then derived as follows. Since we want v_j to be orthogonal to all previous q_i's, we get the equation (for some $1 \le k \le j-1$):

$$\begin{aligned} 0 &= q_k^T v_j, \\ &= q_k^T a_j + \sum_{i=1}^{j-1} \beta_i (q_k^T q_i), \\ &= q_k^T a_j + \beta_k (q_k^T q_k). \end{aligned}$$

Since $q_k^T q_k = 1$ we have that $\beta_k = -q_k^T a_j$ giving the equation in (11.35)

$$v_j = a_j - \sum_{i=1}^{j-1} (q_i^T a_j) q_i.$$

Consider the following example in 2D. We are given a_2 and the (previous) unit vector q_1. We want to find q_2 that is orthonormal to q_1.



We apply the following steps:

1. Orthogonalize using $v_2 = a_2 - (q_1^T a_2)q_1$,
2. Normalize to get $q_2 = \frac{v_2}{\|v_2\|_2}$.

So we have that

$$q_2 = \frac{a_2 - (q_1^T a_2)q_1}{\|a_2 - (q_1^T a_2)q_1\|_2}.$$

But what are the entries of R in our QR factorization?

11.1.2 Upper Triangular Matrix R

To get the entries of R consider the general form of the QR factorization

$$\begin{bmatrix} | & | & & | \\ a_1 & a_2 & \dots & a_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ q_1 & q_2 & \dots & q_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} r_{11} & \dots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{bmatrix}.$$

Written out componentwise we have

$$\begin{aligned}
a_1 &= r_{11}q_1 && \Rightarrow \quad q_1 = \tfrac{a_1}{r_{11}}, \\
a_2 &= r_{12}q_1 + r_{22}q_2 && \Rightarrow \quad q_2 = \tfrac{a_2 - r_{12}q_1}{r_{22}}, \\
a_3 &= r_{13}q_1 + r_{23}q_2 + r_{33}q_3 && \Rightarrow \quad q_3 = \tfrac{a_3 - r_{13}q_1 - r_{23}q_2}{r_{33}}, \\
&\vdots && \qquad\quad \vdots \\
a_n &= r_{1n}q_1 + r_{2n}q_2 + \ldots + r_{nn}q_n && \Rightarrow \quad q_n = \tfrac{a_n - \sum_{i=1}^{n-1} r_{in}q_i}{r_{nn}}.
\end{aligned}$$

Now we compare the q_i above with the result from our Gram-Schmidt orthogonalization expressions. For the 2D example above we have

$$\begin{aligned}
\text{Gram-Schmidt: } q_2 &= \frac{a_2 - (q_1^T a_2)q_1}{\|a_2 - (q_1^T a_2)q_1\|_2}, \\
\text{Factorization: } q_2 &= \frac{a_2 - r_{12}q_1}{r_{22}}.
\end{aligned}$$

$$\begin{aligned}
\text{Gram-Schmidt: } q_2 &= \frac{a_2 - (q_1^T a_2)q_1}{\|a_2 - (q_1^T a_2)q_1\|_2}, \\
\text{Factorization: } q_2 &= \frac{a_2 - r_{12}q_1}{r_{22}}.
\end{aligned}$$

So

$$\begin{aligned} r_{12} &= q_1^T a_2, \text{ and} \\ r_{22} &= \|a_2 - (q_1^T a_2) q_1\|_2. \end{aligned}$$

In general (higher dimensions), the entries of R can be written as

$$\begin{aligned} r_{ij} &= (q_i^T a_j), \\ r_{jj} &= \left\| a_j - \sum_{i=1}^{j-1} r_{ij} q_i \right\|_2. \end{aligned}$$

1. The off-diagonal entries r_{ij} correspond to the lengths of components of a_j in previous directions $q_1, \ldots, q_{j-1}$.
2. The diagonal entries r_{jj} correspond to the length of v_j, required to normalize to q_j.

The classic Gram-Schmidt (CGS) algorithm for QR factorization is given in Algorithm 11.1.2. Note that the classic Gram-Schmidt is numerically **unstable**. That is, it is sensitive to round off error (and can even yield non-orthogonal q's).

Algorithm 11.8 : Gram-Schmidt Algorithm (Classic)

```
for j = 1, 2, ..., n
    v_j = a_j                                   ▷ get next column
    for i = 1, 2, ..., j − 1                    ▷ Orthogonalize
        r_ij = q_i^T a_j
        v_j = v_j − r_ij q_i
    end for
    r_jj = ||v_j||_2                            ▷ Normalize
    q_j = v_j / r_jj
end for
```

11.2 Modified Gram-Schmidt

We can alter the classic Gram-Schmidt algorithm in the inner loop for better numerical stability.

Algorithm 11.9 : QR Factorization With Modified Gram-Schmidt Algorithm

```
for j = 1 : n
    v_j = a_j                                   ▷ Get next column
end for
for j = 1 : n
    r_jj = ||v_j||_2
    q_j = (1/r_jj) v_j                          ▷ Normalize
    for k = j + 1 : n                           ▷ Orthogonalize
        r_jk = q_j^T v_k
        v_k = v_k − r_jk q_j
    end for
end for
```

The modified Gram-Schmidt algorithm (see Algorithm 11.2) gives an identical result as the classic Gram-Schmidt algorithm **in exact arithmetic**.

In floating point arithmetic, the modified Algorithm 11.2 is more numerically stable than the classic Algorithm 11.1.2.

Think carefully about these pseudocodes.

1. In classical Gram-Schmidt, we take each vector, one at a time, and make it orthogonal to all previous vectors.
2. In modified Gram-Schmidt, we take each vector, and modify all forthcoming vectors to be orthogonal to it.

Once you argue this way, it is clear that both methods are performing the same operations, and are mathematically equivalent. But, importantly, modified Gram-Schmidt suffers from round-off instability to a significantly lesser degree. This can be explained, in part, from the formulas for Gram-Schmidt, without QR-factorization:

$$\begin{aligned} CGS: v_k &= a_k - \left(q_j^T a_k\right) q_j, \text{ and} \\ MGS: v_k &= v_k - \left(q_j^T v_k\right) q_j. \end{aligned}$$

If an error is made in computing q_2 in CGS, so that $q_1^T q_2 = \delta$ is small, but non-zero, this will not be corrected for in any of the computations that follow:

$$\begin{aligned} v_3 &= a_3 - (q_1^T a_3)q_1 - (q_2^T a_3)q_2, \\ q_2^T v_3 &= q_2^T a_3 - q_2^T(q_1^T a_3)q_1 - q_2^T(q_2^T a_3)q_2, \\ &= q_2^T a_3 - (q_1^T a_3)\underbrace{q_2^T q_1}_{=\delta} - (q_2^T a_3)\underbrace{q_2^T q_2}_{=1}, \\ &= q_2^T a_3 - (q_1^T a_3)\delta - (q_2^T a_3) \\ &= -(q_1^T a_3)\delta. \end{aligned}$$

Similarly,

$$\begin{aligned} v_3 &= a_3 - (q_1^T a_3)q_1 - (q_2^T a_3)q_2, \\ q_1^T v_3 &= q_1^T a_3 - q_1^T(q_1^T a_3)q_1 - q_1^T(q_2^T a_3)q_2, \\ &= q_1^T a_3 - (q_1^T a_3)\underbrace{q_1^T q_1}_{=1} - (q_2^T a_3)\underbrace{q_1^T q_2}_{=\delta}, \\ &= q_1^T a_3 - (q_1^T a_3) - (q_2^T a_3)\delta \\ &= -(q_2^T a_3)\delta. \end{aligned}$$

We see that v_3 is not orthogonal to either of q_1 or q_2.

On the other hand, assume the same for MGS: $q_1^T q_2 = \delta$ is small, but non-zero. Let's examine how the third vector v_3 changes:

Initially, $v_3^{(0)} = a_3$.

- $\underline{j = 1, k = 3}$

$$\begin{aligned} r_{13} &= q_1^T v_3^{(0)} \\ v_3^{(1)} &= v_3^{(0)} - r_{13} q_1 \\ &= v_3^{(0)} - (q_1^T v_3^{(0)})q_1 \end{aligned}$$

- $\underline{j = 2, k = 3}$

$$\begin{aligned} r_{23} &= q_2^T v_3^{(1)} \\ v_3^{(2)} &= v_3^{(1)} - r_{23} q_2 \\ &= v_3^{(1)} - (q_2^T v_3^{(1)})q_2 \end{aligned}$$

Hence we obtain

$$\begin{aligned} q_2^T v_3^{(2)} &= q_2^T v_3^{(1)} - q_2^T(q_2^T v_3^{(1)})q_2 \\ &= q_2^T v_3^{(1)} - (q_2^T v_3^{(1)})\underbrace{q_2^T q_2}_{=1} \\ &= 0 \qquad (11.36) \\ q_1^T v_3^{(2)} &= q_1^T v_3^{(1)} - q_1^T(q_2^T v_3^{(1)})q_2 \\ &= q_1^T v_3^{(1)} - (q_2^T v_3^{(1)})\underbrace{q_1^T q_2}_{=\delta} \\ &= q_1^T v_3^{(1)} - q_2^T v_3^{(1)}\delta \qquad (11.37) \end{aligned}$$

Computing each of the terms on line (11.37) gives

$$\begin{aligned}
q_1^T v_3^{(1)} &= q_1^T[v_3^{(0)} - (q_1^T v_3^{(0)})q_1] \\
&= q_1^T v_3^{(0)} - q_1^T(q_1^T v_3^{(0)})q_1] \\
&= q_1^T v_3^{(0)} - (q_1^T v_3^{(0)}) \underbrace{q_1^T q_1}_{=1} \\
&= q_1^T v_3^{(0)} - q_1^T v_3^{(0)} \\
&= 0, \text{ and} \\
q_2^T v_3^{(1)} &= q_2^T[v_3^{(0)} - (q_1^T v_3^{(0)})q_1] \\
&= q_2^T v_3^{(0)} - q_2^T(q_1^T v_3^{(0)})q_1 \\
&= q_2^T v_3^{(0)} - (q_1^T v_3^{(0)}) \underbrace{q_2^T q_1}_{=\delta} \\
&= q_2^T v_3^{(0)} - (q_1^T v_3^{(0)})\delta
\end{aligned}$$

Putting it all together gives

$$\begin{aligned}
q_1^T v_3^{(2)} &= q_1^T v_3^{(1)} - q_2^T v_3^{(1)}\delta \\
&= 0 - [q_2^T v_3^{(0)} - (q_1^T v_3^{(0)})\delta]\delta \\
&= -q_2^T v_3^{(0)}\delta + (q_1^T v_3^{(0)})\delta^2
\end{aligned} \tag{11.38}$$

- Recall that $v_3 = v_3^{(2)}$ is the final form of the third vector (before normalization).
- Let's check orthogonality, assuming no more errors are made.
- First for q_2:

$$q_2^T v_3^{(2)} = 0, \text{ from equation (11.36).}$$

  So, we perserve orthogonality to q_2.
- Second for q_1:

$$q_1^T v_3^{(2)} = -q_2^T v_3^{(0)}\delta + (q_1^T v_3^{(0)})\delta^2, \text{ from equation (11.38).}$$

Summarized Comparison of CGS versus MGS

Inner Product	CGS	MGS
$q_2^T v_3$	$-(q_1^T a_3)\delta$	0
$q_1^T v_3$	$-(q_2^T a_3)\delta$	$-q_2^T v_3^{(0)}\delta + (q_1^T v_3^{(0)})\delta^2$ $\underbrace{=}_{v_3^{(0)}=a_3}$ $-q_2^T a_3\delta + q_1^T a_3\delta^2$

Remarks:

1. Since δ is very small, therefore δ^2 is much smaller.
2. Because of the opposite signs of the terms involved, it is likely, but not guaranteed, that $|-q_2^T a_3\delta + q_1^T a_3\delta^2| \leq |-(q_2^T a_3)\delta|$.

3. The error in $q_1^T v_3$ is likely no worse than in CGS, but we have eliminated the errors in $q_2^T v_3$, an improvement.

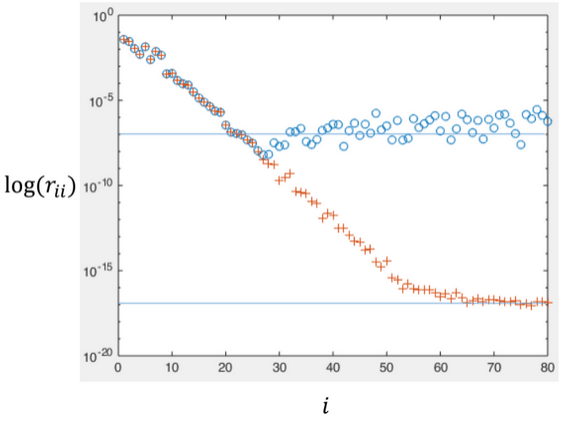
(Reference: https://www.math.uci.edu/~ttrogdon/105A/html/Lecture23.html)

Explanation for the Computation of r_{jk}
Note that the Modified Gram-Schmidt algorithm uses $r_{jk} = q_j^T v_k$, where one might instead expect $r_{jk} = q_j^T a_k$. Here I will prove that these two choices must always yield identical results. Use the notation of the algorithm throughout the explanation.

- Let $1 \leq j \leq n$ be arbitrary.
- Let $j+1 \leq k \leq n$ be arbitrary.
- I claim that $q_j^T v_k = q_j^T a_k$.
- Note that v_k is updated once per (outer) j-loop.
- Since that update takes place AFTER the computation of the $q_j^T v_k$ inner product, at the time of that computation,

$$\begin{aligned} v_k &= a_k - \sum_{\ell=1}^{j-1} r_{\ell k} q_\ell, \text{ so that} \\ q_j^T v_k &= q_j^T \left[a_k - \sum_{\ell=1}^{j-1} r_{\ell k} q_\ell \right] \\ &= q_j^T a_k - \sum_{\ell=1}^{j-1} r_{\ell k} \underbrace{q_j^T q_\ell}_{=0, \text{ since } \ell<j} \\ &= q_j^T a_k, \text{ as claimed.} \end{aligned}$$

The example below computes the QR factorization of a random matrix with hugely varying magnitudes of r_{ii} (diagonal) entries.



The blue points are the result using the **classic** Gram-Schmidt algorithm. It runs out of accuracy at around $\sqrt{E_{\text{machine}}}$. The orange points are the result with the **modified** Gram-Schmidt algorithm. The modified algorithm runs out of accuracy at around E_{machine}. For more information see Lecture 9, Experiment 2 in Trefethen & Bau.

Another exercise is to find the QR factorization of

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

via Gram-Schmidt orthogonalization.

We will solve this problem using Classic Gram Schmidt, then again with modified Gram-Schmidt. Since we are computing exactly, we will obtain the same answer both ways.

Classical Gram-Schmidt

- $\underline{j = 1}$

$$\begin{aligned} v_1 &= a_1 \\ &= \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\ r_{11} &= \|v_1\|_2 \\ &= \left\| \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right\|_2 \\ &= \sqrt{2} \\ q_1 &= \left(\frac{1}{r_{11}} \right) v_1 \\ &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\ &= \frac{\sqrt{2}}{2} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \end{aligned}$$

- $\underline{j = 2}$

$$\begin{aligned} v_2 &= a_2 \\ &= \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \end{aligned}$$

– $\underline{i = 1}$

$$\begin{aligned} r_{12} &= q_1^T a_2 \\ &= \frac{\sqrt{2}}{2}\begin{bmatrix}1 & 0 & 1\end{bmatrix}\begin{bmatrix}2\\1\\0\end{bmatrix} \\ &= \sqrt{2} \\ v_2 &= v_2 - r_{12}q_1 \\ &= \begin{bmatrix}2\\1\\0\end{bmatrix} - \sqrt{2}\left(\frac{\sqrt{2}}{2}\begin{bmatrix}1\\0\\1\end{bmatrix}\right) \\ &= \begin{bmatrix}2\\1\\0\end{bmatrix} - \begin{bmatrix}1\\0\\1\end{bmatrix} \\ &= \begin{bmatrix}1\\1\\-1\end{bmatrix} \end{aligned}$$

$$\begin{aligned} r_{22} &= \|v_2\|_2 \\ &= \left\|\begin{bmatrix}1\\1\\-1\end{bmatrix}\right\|_2 \\ &= \sqrt{3} \\ q_2 &= \left(\frac{1}{r_{22}}\right)v_2 \\ &= \frac{1}{\sqrt{3}}\begin{bmatrix}1\\1\\-1\end{bmatrix} \\ &= \frac{\sqrt{3}}{3}\begin{bmatrix}1\\1\\-1\end{bmatrix} \end{aligned}$$

- $\underline{j = 3}$

$$\begin{aligned} v_3 &= a_3 \\ &= \begin{bmatrix}0\\1\\1\end{bmatrix} \end{aligned}$$

– $\underline{i = 1}$

$$\begin{aligned} r_{13} &= q_1^T a_3 \\ &= \frac{\sqrt{2}}{2} \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \\ &= \frac{\sqrt{2}}{2} \\ v_3 &= v_3 - r_{13} q_1 \\ &= \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} - \frac{\sqrt{2}}{2} \left(\frac{\sqrt{2}}{2} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) \\ &= \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} - \frac{1}{2} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\ &= \frac{1}{2} \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix} \end{aligned}$$

– $\underline{i = 2}$

$$\begin{aligned} r_{23} &= q_2^T a_3 \\ &= \frac{\sqrt{3}}{3} \begin{bmatrix} 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \\ &= 0 \\ v_3 &= v_3 - r_{23} q_2 \\ &= \frac{1}{2} \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix} \end{aligned}$$

$$
\begin{aligned}
r_{33} &= \|v_3\|_2 \\
&= \left\| \frac{1}{2} \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix} \right\|_2 \\
&= \frac{\sqrt{6}}{2} \\
q_3 &= \left(\frac{1}{r_{33}} \right) v_3 \\
&= \frac{2}{\sqrt{6}} \left(\frac{1}{2} \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix} \right) \\
&= \frac{\sqrt{6}}{6} \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}
\end{aligned}
$$

To sum up, the solution to this problem is

$$
\begin{aligned}
Q &= \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{3}}{3} & -\frac{\sqrt{6}}{6} \\ 0 & \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} \end{bmatrix} \\
R &= \begin{bmatrix} \sqrt{2} & \sqrt{2} & \frac{\sqrt{2}}{2} \\ 0 & \sqrt{3} & 0 \\ 0 & 0 & \frac{\sqrt{6}}{2} \end{bmatrix}.
\end{aligned}
$$

Modified Gram-Schmidt We begin by initializing:

$$
\begin{aligned}
v_1 &= a_1 \\
&= \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\
v_2 &= a_2 \\
&= \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \\
v_3 &= a_3 \\
&= \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}
\end{aligned}
$$

Then we compute

- $\underline{j = 1}$

$$\begin{aligned} r_{11} &= \|v_1\|_2 \\ &= \left\| \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right\|_2 \\ &= \sqrt{2} \\ q_1 &= \left(\frac{1}{r_{11}} \right) v_1 \\ &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\ &= \frac{\sqrt{2}}{2} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \end{aligned}$$

  - $\underline{k = 2}$

$$\begin{aligned} r_{12} &= q_1^T v_2 \\ &= \frac{\sqrt{2}}{2} \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \\ &= \sqrt{2} \\ v_2 &= v_2 - r_{12} q_1 \\ &= \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} - \sqrt{2} \left(\frac{\sqrt{2}}{2} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) \\ &= \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \end{aligned}$$

– $\underline{k=3}$

$$\begin{aligned} r_{13} &= q_1^T v_3 \\ &= \frac{\sqrt{2}}{2}\begin{bmatrix}1 & 0 & 1\end{bmatrix}\begin{bmatrix}0\\1\\1\end{bmatrix} \\ &= \frac{\sqrt{2}}{2} \\ v_3 &= v_3 - r_{13}q_1 \\ &= \begin{bmatrix}0\\1\\1\end{bmatrix} - \frac{\sqrt{2}}{2}\left(\frac{\sqrt{2}}{2}\begin{bmatrix}1\\0\\1\end{bmatrix}\right) \\ &= \begin{bmatrix}0\\1\\1\end{bmatrix} - \frac{1}{2}\begin{bmatrix}1\\0\\1\end{bmatrix} \\ &= \frac{1}{2}\begin{bmatrix}-1\\2\\1\end{bmatrix} \end{aligned}$$

- $\underline{j=2}$

$$\begin{aligned} r_{22} &= \|v_2\|_2 \\ &= \left\|\begin{bmatrix}1\\1\\-1\end{bmatrix}\right\|_2 \\ &= \sqrt{3} \\ q_2 &= \left(\frac{1}{r_{22}}\right)v_2 \\ &= \frac{1}{\sqrt{3}}\begin{bmatrix}1\\1\\-1\end{bmatrix} \\ &= \frac{\sqrt{3}}{3}\begin{bmatrix}1\\1\\-1\end{bmatrix} \end{aligned}$$

– $\underline{k=3}$

$$\begin{aligned} r_{23} &= q_2^T v_3 \\ &= \frac{\sqrt{3}}{3}\begin{bmatrix}1 & 1 & -1\end{bmatrix}\left(\frac{1}{2}\begin{bmatrix}-1\\2\\1\end{bmatrix}\right) \\ &= 0 \\ v_3 &= v_3 - r_{23}q_2 \\ &= \frac{1}{2}\begin{bmatrix}-1\\2\\1\end{bmatrix} \end{aligned}$$

- $\underline{j=3}$

$$\begin{aligned} r_{33} &= \|v_3\|_2 \\ &= \left\|\frac{1}{2}\begin{bmatrix}-1\\2\\1\end{bmatrix}\right\|_2 \\ &= \frac{\sqrt{6}}{2} \\ q_3 &= \left(\frac{1}{r_{33}}\right)v_3 \\ &= \frac{2}{\sqrt{6}}\left(\frac{1}{2}\begin{bmatrix}-1\\2\\1\end{bmatrix}\right) \\ &= \frac{\sqrt{6}}{6}\begin{bmatrix}-1\\2\\1\end{bmatrix} \end{aligned}$$

To sum up, the solution to this problem is

$$\begin{aligned} Q &= \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{3}}{3} & -\frac{\sqrt{6}}{6} \\ 0 & \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} \end{bmatrix} \\ R &= \begin{bmatrix} \sqrt{2} & \sqrt{2} & \frac{\sqrt{2}}{2} \\ 0 & \sqrt{3} & 0 \\ 0 & 0 & \frac{\sqrt{6}}{2} \end{bmatrix}. \end{aligned}$$

11.3 Complexity of Gram-Schmidt

The inner i-loop involves

- $r_{ij} = q_i^T a_j$ (CGS) or $q_i^T v_j$ (MGS) $\Rightarrow m$ (scalar) multiplications and $m-1$ additions for the inner products,

- $v_j = v_j - r_{ij}q_i$ (CGS) or $v_j = v_j - r_{jk}q_j$ (MGS) $\Rightarrow m$ multiplications, and m subtractions.

Hence, the flops per inner loop $\approx 4m$. An approximation of the total flops is therefore

$$\begin{aligned}\sum_{j=1}^{n}\sum_{i=1}^{j-1} 4m &= 4m\sum_{j=1}^{n}(j-1),\\ &\approx 4m\sum_{j=1}^{n} = 4m\frac{n(n+1)}{2},\\ &\approx 2mn^2.\end{aligned}$$

If the matrix is square ($m = n$), flops(Gram-Schmidt) $= 2n^3 + O(n^2) \approx 3 \times$ flops(LU). We could use QR factorization to solve linear systems also, but at a cost $3\times$ greater than LU factorization. As was pointed out earlier, if A is square and $A = QR$, then solving $Ax = b$ for x is equivalent to solving $Rx = Q^T b$ for x.

12 Lecture 12: Householder QR factorizations

Outline

1. Householder Triangularization
2. Householder QR Factorization Algorithm
3. Example: Householder Reflector
4. Example: QR Factorization via Householder

Recall that in this course we consider three common algorithms for QR factorization:

1. Gram-Schmidt orthogonalization,
2. Householder reflections,
3. Givens rotations.

Gram-Schmidt orthogonalization was discussed in Lecture 11. This lecture will introduce the idea of Householder reflections for building the QR factorization. A final approach of Givens rotations will be presented in the next lecture.

- Note that the QR factorization we produce here is similar, but not identical, to the one we produced last time:
  1. R is $m \times n$ and Q is square, instead of the other way around, and
  2. Negative entries can occur on R's "diagonal".

12.1 Householder Triangularization

Note that Gram-Schmidt orthogonalization is a "triangular orthogonalization" process. In matrix form, Gram-Schmidt can be written as right-multiplication by triangular matrices that make the columns of A orthonormal (see end of Lecture 8 of Trefethen & Bau)

$$A \underbrace{R_1 R_2 \cdots R_n}_{\hat{R}^{-1}} = \hat{Q}.$$

Householder reflections instead provide an "orthogonal triangularization" process. The matrix A is made to be triangular (R) by applying orthogonal matrices Q_j, i.e.,

$$\underbrace{Q_n \cdots Q_2 Q_1}_{Q^{-1}} A = R.$$

Hence, the premise of Householder reflections (aka triangularization) is to find the orthogonal matrices $Q_j \in \mathbb{R}^{m \times m}$. This method is similar to LU-factorization, as each Q_j will zero the lower entries of column j.

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

We will build orthogonal matrices of the following form

$$Q_j = \begin{bmatrix} I & 0 \\ 0 & F \end{bmatrix} \begin{matrix} \} \ j-1 \text{ rows, already done,} \\ \} \ m-(j-1) \text{ rows, still to be done.} \end{matrix}$$

where F is the **Householder reflector** matrix. F reflects a vector x across a (specific) hyperplane H to produce a vector **along the axis**. See Figure 12.27 for a visualization of applying the Householder reflector (note $e_1 = [1, 0, \ldots, 0]^T$).

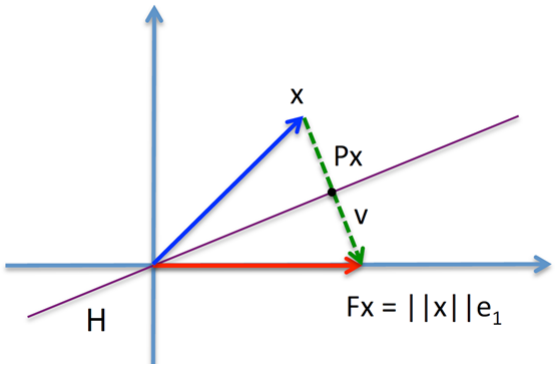


Figure 12.27: Applying the Householder reflector F to the vector x, which reflects x across the hyperplane H.

$$x + v^- = -\|x\|e_1 \Leftrightarrow v^- = -\|x\|e_1 - x.$$

After reflection, the output vector has the same magnitude as x, and is parallel to e_1. It depends on both of x and e_1.

At step j, we start with e_1, and reflect onto the subspace spanned by $\{e_i, \ldots e_j\}$.

We find the Householder reflector matrix F, to perform the reflection, as follows. Suppose

$$x = \begin{bmatrix} \times \\ \times \\ \vdots \\ \times \end{bmatrix},$$

Find F such that

$$Fx = \begin{bmatrix} \|x\| \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \|x\|e_1.$$

The F reflects x across the hyperplane H **orthogonal to** $v = \|x\|e_1 - x$. That is because we want to produce a new vector of the same length as x, aligned with the axis e_1 (so all but the first entry are zeros).

The **orthogonal projection** P of x **onto** the hyperplane H (orthogonal to the vector v) is

$$Px = x - \left(\left(\frac{v}{\|v\|} \right)^T x \right) \frac{v}{\|v\|} = x - v \left(\frac{v^T x}{v^T v} \right).$$

Note that this orthogonal projection P is similar to the steps in Gram-Schmidt orthogonalization. **Idea:** Subtract out the component of x **along** v. To **reflect** x across H (instead of projecting onto H) we must go twice as far in the same direction (see Figure 12.27)

$$Fx = x - 2v \left(\frac{v^T x}{v^T v} \right).$$

Therefore, the **Householder reflector** F is given by

$$F = I - 2 \left(\frac{vv^T}{v^T v} \right)$$

where $v = \|x\| e_1 - x$.

Remark that we could instead reflect to the point along the axis with a **negative** sign. That is, reflect to $-\|x\| e_1$ instead of $\|x\| e_1$, which gives

$$Fx = \begin{bmatrix} -\|x\| \\ 0 \\ \vdots \\ 0 \end{bmatrix} = -\|x\| e_1.$$

Either choice zeros out the desired entries of the active column. We just get a different v as shown in Figure 12.28.

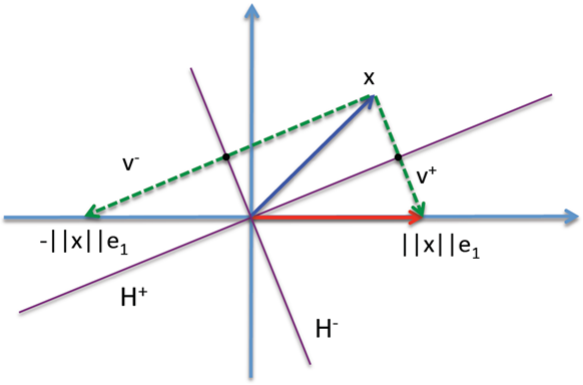


Figure 12.28: The two alternative Householder reflections.

Reflecting to either of $\|x\| e_i$ or $-\|x\| e_i$ will zero the remainder of the desired column.

Which Householder reflector F should we choose? For numerical stability, we want the F that reflects x farther away from itself. Thus,

- if $x_1 > 0$ we choose the negative one, $-\|x\|e_1$,
- if $x_1 < 0$ we choose the positive one, $\|x\|e_1$.

This gives $v = -\text{sign}(x_1)\|x\|e_1 - x$, or more simply (because only direction is important, and either choice gives the same F) $\text{sign}(x_1)\|x\|e_1 + x$. This choice of v avoids subtracting nearby quantities, which can introduce cancellation error. Therefore, choosing the F that reflects x farther away is more numerically stable.

Alternative Derivation

We will show an alternate derivation of Householder triangularization. Consider a reflection operator $F = I - 2\frac{vv^T}{v^Tv}$ for an arbitrary vector v. We want to find v such that $Fx \in \text{span}\{e_1\}$ to zero the lower entries in column 1. Let $Fx \in \text{span}\{e_1\}$, in other words,

$$Fx = x - \frac{2vv^Tx}{v^Tv} = x - \left[\frac{2(v^Tx)}{(v^Tv)}\right] v \in \text{span}\{e_1\}.$$

Observe $v \in \text{span}\{e_1, x\}$ by construction, since $Fx \in \text{span}\{e_1\}$. Write $Fx = c_2e_1$ for some scalar c_2. Hence,

$$\begin{aligned} c_2e_1 &= x - c_1v \\ \Rightarrow v &= \hat{c_1}x + \hat{c_2}e_1, \end{aligned}$$

(for scalars $\hat{c_1}$ and $\hat{c_2}$) which means $v \in \text{span}\{e_1, x\}$.

Now let $\hat{v} = x + \alpha e_1$ for some scalar α. (Note the length of v does not matter; only its direction matters.) We will write v for $\hat{v}$ from now on. Then,

$$\begin{aligned} v^Tx &= (x + \alpha e_1)^Tx \\ &= x^Tx + \alpha e_1^Tx \\ &= x^Tx + \alpha \underbrace{x_{(1)}}_{\text{scalar}} \end{aligned}$$

and

$$\begin{aligned} v^Tv &= (x + \alpha e_1)^T(x + \alpha e_1) \\ &= x^Tx + 2\alpha x_{(1)} + \alpha^2. \end{aligned}$$

Plugging into Fx to determine α, we have

$$\begin{aligned}
Fx &= x - 2\left(\frac{v^T x}{v^T v}\right)(x + \alpha e_1), \\
&= \left(1 - \frac{2v^T x}{v^T v}\right) x - \left(2\alpha \frac{v^T x}{v^T v}\right) e_1, \\
&= \left(1 - \frac{2(x^T x + \alpha x_{(1)})}{x^T x + 2\alpha x_{(1)} + \alpha^2}\right) x - \left(2\alpha \frac{v^T x}{v^T v}\right) e_1, \\
&= \left(\frac{x^T x + \cancel{2\alpha x_{(1)}} + \alpha^2 - 2x^T x - \cancel{2\alpha x_{(1)}}}{x^T x + 2\alpha x_{(1)} + \alpha^2}\right) x - \left(2\alpha \frac{v^T x}{v^T v}\right) e_1, \\
&= \underbrace{\left(\frac{\alpha^2 - x^T x}{x^T x + 2\alpha x_{(1)} + \alpha^2}\right) x}_{\text{must be } 0} - \left(2\alpha \frac{v^T x}{v^T v}\right) e_1.
\end{aligned}$$

Since $Fx \in \text{span}\{e_1\}$ the first term must be zero, so $\alpha^2 - x^T x = 0 \Rightarrow \alpha = \pm\|x\|$. Hence,

$$v = x \pm \|x\| e_1 \quad \text{and} \quad Fx = \mp \|x\| e_1,$$

as we saw last time.

12.2 Householder QR Factorization Algorithm

Algorithm 12.10 gives the QR factorization of A via Householder triangularization.

Algorithm 12.10 : Householder QR factorization algorithm

```
for k = 1, 2, ..., n
    x = A(k : m, k)                          ▷ Get current column
    v_k = sign(x_1)‖x‖e_1 + x                ▷ Form the reflection vector
    v_k = v_k / ‖v_k‖                        ▷ Normalize
    for j = k, k + 1, ..., n                 ▷ Apply F to active lower-right block
        A(k : m, j) = A(k : m, j) − 2v_k(v_k^T A(k : m, j))
    end for
end for
```

The notation used follows Matlab, i.e., $A(k:m, j) = j^{\text{th}}$ column of A from row k to row m. The algorithm converts A into R (upper "triangular") using Householder reflections F. Note that one could **further** vectorize the inner loop (more efficient in Matlab) to the matrix operation

$$A(k:m, k:n) = A(k:m, k:n) - 2v_k(v_k^T A(k:m, k:n)).$$

Algorithm 12.10 does not construct Q, only the vectors v_k. Why is this not a problem in practice? We often do not need Q but just the products $Q^T b$ or Qx (e.g., for least squares

we solve $Rx = Q^T b$). Since

$$\begin{aligned} Q^T &= Q_n Q_{n-1} \dots Q_2 Q_1, \\ Q &= Q_1^T Q_2^T \dots Q_{n-1}^T Q_n^T, \end{aligned}$$

we can efficiently (in $O(mn)$ flops) compute $Q^T b$ or Qx using just the v_k's to apply the appropriate reflections (see Algorithms 12.11 and 12.12). **Note the parentheses**, we compute $v(v^T b)$ rather than $(vv^T)b$ to avoid forming the matrix vv^T.

Algorithm 12.11 : Implicit $Q^T b$

for $k = 1, 2, \dots, n$
  $b(k:m) = b(k:m) - 2v_k \left(v_k^T b(k:m)\right)$
end for

Algorithm 12.12 : Implicit Qx

for $k = n, n-1, \dots, 1$
  $x(k:m) = x(k:m) - 2v_k \left(v_k^T x(k:m)\right)$
end for

Explicitly building the matrix Q may sometimes be necessary. So how **could** we use these implicit products to recover Q itself? We compute the product $QI = Q$ by applying Q to the columns of the identity matrix I $(e_1, e_2, \dots)$, i.e., $q_1 = Qe_1, q_2 = Qe_2, \dots$. For the reduced QR factorization

$$A = \hat{Q}\hat{R} = \left[\ \right]_{m \times n} \left[\ \right]_{n \times n}.$$

So $\hat{Q}$ is given by just the first n columns of I

$$\hat{Q} = \begin{bmatrix} | & | & & | \\ Qe_1 & Qe_2 & \cdots & Qe_n \\ | & | & & | \end{bmatrix}.$$

Complexity of Householder-Based QR Work is dominated by inner loop

$$A(k:m, j) = A(k:m, j) - 2v_k \left(v_k^T A(k:m, j)\right).$$

Tallying the cost gives:

- $v_k^T A(k:m, j) \approx 2(m-k+1)$ flops (dot product),
- $2v_k \left(v_k^T A(k:m, j)\right) \approx (m-k+1)$ flops (scalar multiply),
- $A(k:m, j) - 2v_k \left(v_k^T A(k:m, j)\right) \approx (m-k+1)$ flops (subtraction).

So it approximately costs $4(m-k+1)$ flops per inner step, which is done $(n-k+1)$ times (j-loop). This totals to $4(m-k+1)(n-k+1)$ flops per outer iteration. The outer k-loop runs from 1 to n, which means the total flops can be approximated by

$$\sum_{k=1}^{n} 4(m-k+1)(n-k+1) \approx 2mn^2 - \frac{2}{3}m^3.$$

Note that this does **not** include forming Q.

For $m = n$ (square), flops(Householder) $\approx \frac{4}{3}n^3 = 2 \times$ flops(LU). Recall from Lecture 11 that Gram-Schmidt orthogonalization cost $\approx 3 \times$ flops(LU). So Householder triangularization is faster than Gram-Schmidt orthogonalization.

12.3 Example: Householder Reflector

Given $x = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$ find the Householder reflector F and the product Fx.

Answer: We see that $\|x\| = \sqrt{1^2 + 2^2 + 2^2} = 3$. Therefore

$$v = \pm\|x\|e_1 + x = \pm 3 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix} \text{ or } \begin{bmatrix} -2 \\ 2 \\ 2 \end{bmatrix}.$$

We choose $v = \text{sign}(x_1)\,\|x\|\,e_1 + x = 3 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix}$, for its numerical stability.

$$\begin{aligned}
\frac{vv^T}{v^Tv} &= \left(\frac{1}{\begin{bmatrix} 4 & 2 & 2 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix}} \right) \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix} \begin{bmatrix} 4 & 2 & 2 \end{bmatrix} \\
&= \left(\frac{1}{24} \right) \begin{bmatrix} 16 & 8 & 8 \\ 8 & 4 & 4 \\ 8 & 4 & 4 \end{bmatrix} \\
&= \left(\frac{1}{6} \right) \begin{bmatrix} 4 & 2 & 2 \\ 2 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \\
F &= I - 2 \left(\frac{vv^T}{v^Tv} \right) \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \frac{1}{3} \begin{bmatrix} 4 & 2 & 2 \\ 2 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \\
&= \frac{1}{3} \begin{bmatrix} -1 & -2 & -2 \\ -2 & 2 & -1 \\ -2 & -1 & 2 \end{bmatrix}, \text{ so that} \\
Fx &= \frac{1}{3} \begin{bmatrix} -1 & -2 & -2 \\ -2 & 2 & -1 \\ -2 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} \\
&= \begin{bmatrix} -3 \\ 0 \\ 0 \end{bmatrix}.
\end{aligned}$$

Since we are computing exactly, we are not forced to consider numerical stability. We re-

compute everything with the other possible choice, namely $v = \begin{bmatrix} -2 \\ 2 \\ 2 \end{bmatrix}$.

$$\begin{aligned}
\frac{vv^T}{v^Tv} &= \left(\frac{1}{\begin{bmatrix} -2 & 2 & 2 \end{bmatrix} \begin{bmatrix} -2 \\ 2 \\ 2 \end{bmatrix}} \right) \begin{bmatrix} -2 \\ 2 \\ 2 \end{bmatrix} \begin{bmatrix} -2 & 2 & 2 \end{bmatrix} \\
&= \left(\frac{1}{12} \right) \begin{bmatrix} 4 & -4 & -4 \\ -4 & 4 & 4 \\ -4 & 4 & 4 \end{bmatrix} \\
&= \left(\frac{1}{3} \right) \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix} \\
F &= I - 2 \left(\frac{vv^T}{v^Tv} \right) \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \frac{2}{3} \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix} \\
&= \frac{1}{3} \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & -2 \\ 2 & -2 & 1 \end{bmatrix}, \text{ so that} \\
Fx &= \frac{1}{3} \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & -2 \\ 2 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} \\
&= \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}.
\end{aligned}$$

12.4 Example: QR Factorization via Householder

Perform QR factorization using Householder reflections on the matrix

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

Step 1: $x = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}, v_- = \begin{bmatrix} 4 \\ 2 \\ 2 \end{bmatrix}, v_+ = \begin{bmatrix} 2 \\ -2 \\ -2 \end{bmatrix}$

(sign of v is irrelevant, since we only use vv^T and v^Tv.) Ordinarily we would use v_-, but let's use v_+ for variety.

Then $F_1 = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & \frac{-2}{3} \\ \frac{2}{3} & \frac{-2}{3} & \frac{1}{3} \end{bmatrix} = Q_1$ and (by multiplying) $Q_1 A = \begin{bmatrix} 3 & 2 \\ 0 & -3 \\ 0 & -4 \end{bmatrix}$

Step 2: $x = \begin{bmatrix} -3 \\ -4 \end{bmatrix}$ and so

$$\begin{aligned} v &= x + sign(x_1)\|x\|e_1 \\ &= \begin{bmatrix} -3 \\ -4 \end{bmatrix} + (-1) \cdot 5 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -8 \\ -4 \end{bmatrix} \end{aligned}$$

which lets us calculate

$$\begin{aligned} F_2 &= I - \frac{2vv^T}{v^T v} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \frac{2}{80} \begin{bmatrix} 64 & 32 \\ 32 & 16 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \frac{1}{5} \begin{bmatrix} 8 & 4 \\ 4 & 2 \end{bmatrix} \\ &= \begin{bmatrix} \frac{-3}{5} & \frac{-4}{5} \\ \frac{-4}{5} & \frac{3}{5} \end{bmatrix} \end{aligned}$$

Therefore $Q_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & & \\ 0 & & F_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{-3}{5} & \frac{-4}{5} \\ 0 & \frac{-4}{5} & \frac{3}{5} \end{bmatrix}$

So $Q_2(Q_1 A) = R = \begin{bmatrix} 3 & 2 \\ 0 & 5 \\ 0 & 0 \end{bmatrix}$ (by multiplying)

Therefore

$$\begin{aligned} A &= Q_1^{-1} Q_2^{-1} R \\ &= Q_1^T Q_2^T R \qquad \text{By orthogonality} \\ &= Q_1 Q_2 R \qquad \text{by symmetry} \\ &= QR \end{aligned}$$

Orthogonality does not imply symmetry, but those Q's were constructed to be symmetric, $I - \frac{2vv^T}{v^T v}$.

$$\begin{aligned} Q = Q_1 Q_2 &= \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & \frac{-2}{3} \\ \frac{2}{3} & \frac{-2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{-3}{5} & \frac{-4}{5} \\ 0 & \frac{-4}{5} & \frac{3}{5} \end{bmatrix} \\ &= \frac{1}{15} \begin{bmatrix} 5 & -14 & -2 \\ 10 & 5 & -10 \\ 10 & 2 & 11 \end{bmatrix} \end{aligned}$$

13 Lecture 13: Givens Rotations

Outline

1. Givens Rotations
2. Hessenberg via Givens
3. Least Squares: Normal Equations vs QR

In this lecture we discuss the last of the three common algorithms for QR factorization:

1. Gram-Schmidt orthogonalization,
2. Householder reflections,
3. Givens rotations.

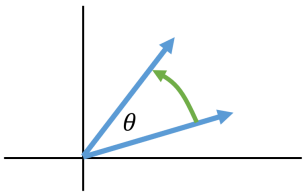
13.1 Givens Rotations

Rotation Matrices in 2D

First consider rotating a vector in **two dimensions**. This can be described as multiplication with a 2×2 matrix

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

That is, $a = Rb$ rotates the vector b **counterclockwise** by θ as shown in the figure below.

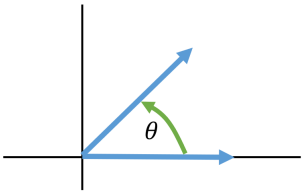


The columns of $R(\theta)$ are orthonormal: using trigonometric identities we have $\cos^2\theta + \sin^2\theta = 1$ and $\cos\theta\sin\theta - \cos\theta\sin\theta = 0$. Hence it is easy to see that $R(\theta)$ is an orthogonal matrix. The transpose of the matrix $R(\theta)$ gives a **clockwise** rotation (i.e., the inverse operation).

As an example, if we want to rotate a vector by $\theta = \frac{\pi}{4}$ (45 degrees) the rotation matrix is

$$R = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

If $b = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ then, $a = Rb = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$ as shown in the figure below.



A **Givens rotation** zeros individual elements "selectively" by an orthogonal matrix operation that performs rotation in the **(i,k)-plane only**.

Givens rotation matrices have the form

$$G(i,k,\theta)^T = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & & -s & & \\ & & & 1 & & & \\ & & & & \ddots & & \\ & & & & & 1 & \\ & & s & & c & & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \text{row}(i) \\ \\ \\ \\ \\ \\ \text{row}(k) \\ \\ \end{matrix}$$

$$\begin{matrix} & & \text{col}(i) & & \text{col}(k) & & \end{matrix}$$

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

- The matrix mostly resembles the identity except for two rows/columns describing a rotation.
- $G(i,k,\theta)$ is always an orthogonal matrix.
- We choose i,k so that left multiplication by $G(i,k,\theta)$ uses the entry on row i, to zero out the entry on row k, in the column in which we are working.

Explanation of the Following Setup

- Let x, y be vectors in $\mathbb{R}^m$.
- Both x and y will be columns of the matrix we are about to process:
  - x is the input column, and
  - y is the output column.
- It is implicit that we have already chosen a column on which to work. This provides us with our input, x, and shows what y we are pursuing.

Consider $y = G(i,k,\theta)^T x$, then

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

To force $y_k = 0$ we must let

$$\begin{aligned} c &= \frac{x_i}{\sqrt{x_i^2 + x_k^2}}, \text{ and} \\ s &= -\frac{x_k}{\sqrt{x_i^2 + x_k^2}}. \end{aligned}$$

Exercise: confirm $y_k = 0$ with the above values of c and s by substituting into the definition of y_j. Note that the value of θ itself is not needed when computing $y = G(i,k,\theta)^T x$. Also, note that computing the product $G(i,k,\theta)^T A$ affects only row(i) and row(k).

Solution to Exercise: Recall that y_k (i.e. y_j, where $j = k$) is defined by

$$y_k = sx_i + cx_k.$$

So we compute

$$\begin{aligned} y_k &= sx_i + cx_k \\ &= \left(-\frac{x_k}{\sqrt{x_i^2 + x_k^2}}\right) x_i + \left(\frac{x_i}{\sqrt{x_i^2 + x_k^2}}\right) x_k \\ &= -\frac{x_i x_k}{\sqrt{x_i^2 + x_k^2}} + \frac{x_i x_k}{\sqrt{x_i^2 + x_k^2}} \\ &= \frac{x_i x_k - x_i x_k}{\sqrt{x_i^2 + x_k^2}} \\ &= 0. \end{aligned}$$

Givens QR Factorization Process

To perform a QR factorization we zero entries one at a time, working upwards along columns. For example, the process performed on a 4×3 matrix is

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{G(3,4)^T} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{G(2,3)^T} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{G(1,2)^T}$$

$$\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{G(3,4)^T} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{G(2,3)^T} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{G(3,4)^T} R$$

Remarks on Matrix Dimensions:

1. A is $m \times n$, for some $m \geq n$.
2. The G_is are all $m \times m$.

- We are, step-by-step, turning A into R.

- The output will be a reduced QR factorization: R is $m \times n$; Q is $m \times m$, orthogonal.
- We can recover Q later, from the G's.
- Each rotation is computed based on the current matrix, not based on the original matrix.

Explanation:

1. x is a column of our coefficient matrix.
2. y is the same column of the coefficient matrix, after we have applied a Givens rotation to zero out the k^{th} entry.
3. We construct $G(i,k)^T$, to zero out the k^{th} entry of x. This is why we set $y_k = 0$ to determine what c and s have to be.
4. Think of $G(i,k)^T$ as the matrix which carries out the needed rotation in the (i,k)-plane to zero out the k^{th} entry of x.
5. As pointed out above, $G(i,k)^T$ affects only rows i and k.
6. Also as pointed out above, to perform a QR factorization we zero entries one at a time, **working upwards along columns**.
7. This should now explain the indices in the diagrams:
   (a) The first line works upwards through column 1:
      i. Perform the Givens rotation on rows 3 and 4 that zeroes out the (4,1) entry of the matrix $(G(3,4)^T)$.
      ii. Perform the Givens rotation on rows 2 and 3 that zeroes out the (3,1) entry of the matrix $(G(2,3)^T)$.
      iii. Perform the Givens rotation on rows 1 and 2 that zeroes out the (2,1) entry of the matrix $(G(1,2)^T)$.
   (b) The second line then does the analogous steps to create the needed zeroes in columns 2 and 3.
   (c) Note that $G(3,4)^T$ on line 2 is **not** the same as $G(3,4)^T$ on line 1: they are operating on different columns. The column number is implicit (because we always know which column we are processing). The (i,k) indices refer to the **row entries in the current column**.

To obtain Q we let G_ℓ^T denote the ℓ^{th} Givens rotation. We can assemble Q from

$$\begin{aligned} G_\ell^T G_{\ell-1}^T \cdots G_2^T G_1^T A &= R, \\ \Rightarrow A &= G_1 G_2 \cdots G_{\ell-1} G_\ell R, \quad (G_i \text{ orthogonal}) \\ \Rightarrow Q &= G_1 G_2 \cdots G_{\ell-1} G_\ell, \quad (\text{because } A = QR). \end{aligned}$$

Quick Reminder About Dimensions:

1. A is $m \times n$, for some $m \geq n$.
2. The G_is are all $m \times m$.

Remark: We do not require R's "diagonal" entries to be positive. So this QR factorization might not agree with the unique QR factorization described in Lecture 10.

Example: Let $A = \begin{bmatrix} 1 & -1 \\ 0 & 2 \\ 1 & 1 \end{bmatrix}$. We will compute a QR factorization of A, via Givens rotations.

Column #1

- There is no need to zero out the $(2,1)$ entry.
- Compute $G(1,3)^T$ to use the $(1,1)$ entry to zero out the $(3,1)$ entry.

$$\begin{aligned} c &= \frac{x_1}{\sqrt{x_1^2 + x_3^2}} \\ &= \frac{1}{\sqrt{1^2 + 1^2}} \\ &= \frac{1}{\sqrt{2}} \\ &= \frac{\sqrt{2}}{2}, \text{ and} \end{aligned}$$

$$\begin{aligned} s &= -\frac{x_3}{\sqrt{x_1^2 + x_3^2}} \\ &= -\frac{1}{\sqrt{1^2 + 1^2}} \\ &= -\frac{1}{\sqrt{2}} \\ &= -\frac{\sqrt{2}}{2}. \end{aligned}$$

Hence we get

$$G(1,3)^T = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix}.$$

We check that left mulitplying A by $G(1,3)^T$ has the desired effect.

$$\begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 2 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 \\ 0 & 2 \\ 0 & \sqrt{2} \end{bmatrix}.$$

This completes Column #1.

Column #2

- Compute $G(2,3)^T$ to use the $(2,2)$ entry to zero out the $(3,2)$ entry.

$$\begin{aligned} c &= \frac{x_2}{\sqrt{x_2^2 + x_3^2}} \\ &= \frac{2}{\sqrt{2^2 + \sqrt{2}^2}} \\ &= \frac{2}{\sqrt{6}} \\ &= \frac{2\sqrt{6}}{6} \\ &= \frac{\sqrt{6}}{3}, \text{ and} \end{aligned}$$

$$\begin{aligned} s &= -\frac{x_3}{\sqrt{x_2^2 + x_3^2}} \\ &= -\frac{\sqrt{2}}{\sqrt{2^2 + \sqrt{2}^2}} \\ &= -\frac{\sqrt{2}}{\sqrt{6}} \\ &= -\frac{1}{\sqrt{3}} \\ &= -\frac{\sqrt{3}}{3}. \end{aligned}$$

Hence we get

$$G(2,3)^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{6}}{3} & \frac{\sqrt{3}}{3} \\ 0 & -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} \end{bmatrix}.$$

We check that left mulitplying A by $G(2,3)^T$ has the desired effect.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{6}}{3} & \frac{\sqrt{3}}{3} \\ 0 & -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} \end{bmatrix} \begin{bmatrix} \sqrt{2} & 0 \\ 0 & 2 \\ 0 & \sqrt{2} \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 \\ 0 & \sqrt{6} \\ 0 & 0 \end{bmatrix}.$$

This completes Column #2.

This gives immediately that $R = \begin{bmatrix} \sqrt{2} & 0 \\ 0 & \sqrt{6} \\ 0 & 0 \end{bmatrix}$.

We compute Q as the product of the G's in the reverse order:

$$\begin{aligned} Q &= \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{6}}{3} & -\frac{\sqrt{3}}{3} \\ 0 & \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{6}}{6} & -\frac{\sqrt{3}}{3} \\ 0 & \frac{\sqrt{6}}{3} & -\frac{\sqrt{3}}{3} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{6} & \frac{\sqrt{3}}{3} \end{bmatrix}. \end{aligned}$$

One can verify that

$$QR = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{6}}{6} & -\frac{\sqrt{3}}{3} \\ 0 & \frac{\sqrt{6}}{3} & -\frac{\sqrt{3}}{3} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{6} & \frac{\sqrt{3}}{3} \end{bmatrix} \begin{bmatrix} \sqrt{2} & 0 \\ 0 & \sqrt{6} \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 2 \\ 1 & 1 \end{bmatrix} = A.$$

In terms of complexity, flops(Givens QR) $\approx 3mn^2 - n^3 = 1.5 \times$ flops(Householder QR). So why bother with this the Givens QR if it is slower then Householder QR? The reason is because it is more **flexible** than Householder QR. Givens QR factorization can be useful when only a few elements need to be eliminated.

13.2 Hessenberg via Givens

For example, consider an **upper Hessenberg** matrix, which has nonzeros only above the first subdiagonal. Performing QR factorization on an upper Hessenberg matrix involves the following

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{G(1,2)^T} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{G(2,3)^T} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ & 0 & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{G(3,4)^T}$$

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ & 0 & \times & \times & \times \\ & & 0 & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{G(4,5)^T} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ & 0 & \times & \times & \times \\ & & 0 & \times & \times \\ & & & 0 & \times \end{bmatrix}$$

That is, we only need to zero out the first subdiagonal entries at a cost of $\approx 3n^2$ flops. This is less than if one used Householder QR factorization, which operates columnwise on all the entries below the main diagonal.

13.3 Least Squares: Normal Equations vs QR

We have now seen three ways to compute the QR factorization (Gram-Schmidt, Householder, Givens). Recall that, as explained in Lecture 12, the shape of the output of the Householder

QR factorization is different from the shapes of the other outputs. However, the steps to solving the least square problem are the same after the $A = QR$ is computed. Therefore, the two main approaches to solving least squares problems are using:

- the normal equations or
- the QR factorization.

QR factorization can also be used to (exactly) solve the system $Ax = b$, when A is square. In this case, we solve $Rx = Q^T b$.

The drawback of using normal equations ($A^T Ax = A^T b$) when solving for the least squares problems is that is it poorly conditioned! Recall that the accuracy of a (square) linear system solution is dictated by the condition number $\kappa(A)$. However, with the normal equations the accuracy depends on $\kappa(A^T A)$ instead of $\kappa(A)$, which is often much worse. For example,

$$A = \begin{bmatrix} 1 + 10^{-8} & -1 \\ -1 & 1 \end{bmatrix} \Rightarrow \kappa(A) = 4 \times 10^8,$$
$$A^T A = \begin{bmatrix} 2 + 10^{-8} + 10^{-16} & -2 - 10^{-8} \\ -2 - 10^{-8} & 2 \end{bmatrix} \Rightarrow \kappa(A) \approx 16 \times 10^{16}.$$

The QR factorization approach to least squares involves solving the system $Rx = Q^T b$. Therefore, the solution's accuracy depends on $\kappa(R)$ because

$$\kappa_2(A) = \kappa_2(QR) = \kappa_2(R),$$

since $\|Q\|_2 = 1 :$ Q has orthonormal columns.

Remark: For this to make sense, R must be square.

The main point is, we prefer the QR approach, despite its extra cost, because of the potential to encounter ill-conditioned problems (for which the normal equations roughly square the condition number). However, the normal equations approach can be used if it is known that A is well-conditioned.

14 Lecture 14: Eigenvalues / Eigenvectors

Outline

1. Eigenvalue Problem Definitions
2. Traditional Eigenvalue Problem Review
3. Solving Eigenvalue Problems (Naïve Approach)
4. Eigenvalue/Eigenvector Review Example
5. Rayleigh quotient
6. Power Iteration

So far we have discussed solving linear systems and least-squares problems of the form $Ax = b$. We now consider eigenvalue problems, which have the form

$$Ax = \lambda x.$$

In this lecture, we begin with some definitions and theory about eigenvalue problems. Much of the beginning of this lecture should be review from previous courses.

14.1 Eigenvalue Problem Definitions

Definition 14.1. *Let $A \in \mathbb{R}^{n\times n}$. A non-zero vector $x \in \mathbb{R}^n$ is a (right) **eigenvector** with corresponding **eigenvalue** $\lambda \in \mathbb{R}$ if*

$$Ax = \lambda x.$$

Note that if x is an eigenvector then so is ax, for $a \neq 0$. That is, eigenvectors are unique only up to a multiplicative constant.

Definition 14.2. *The set of A's eigenvalues is called its **spectrum**, denoted $\Lambda(A)$.*

Definition 14.3. *The **eigendecomposition** of a diagonalizable matrix, A, is*

$$A = X\Lambda X^{-1}$$

where

$$X = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_n \\ | & | & & | \end{bmatrix}, \qquad \Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix},$$

and $Ax_i = \lambda_i x_i$ for $i = 1, 2, \ldots, n$.

The columns of X in the eigendecomposition are eigenvectors of A. The diagonal matrix Λ has entries that are the eigenvalues corresponding to each eigenvector. Equivalently, the eigendecomposition can be written as $AX = X\Lambda$

$$\begin{bmatrix} & \\ A & \\ & \end{bmatrix} \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix}.$$

This form corresponds to the form of the eigenvalue problem $Ax = \lambda x$, but stacks all eigenvalue/eigenvector pairs (x_i, λ_i) in one matrix equation (for $i = 1, \ldots, n$).

Fact: Real symmetric matrices are diagonalizable by orthogonal matrices.

Proof that $AX = X\Lambda$: We reverse the usual subscript order in the X matrix: the first index is the column; the second is the row.

$$\begin{aligned}
X\Lambda &= \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix} \\
&= \begin{bmatrix} \lambda_1 x_{11} & \lambda_2 x_{21} & \cdots & \lambda_n x_{n1} \\ \lambda_1 x_{12} & \lambda_2 x_{22} & \cdots & \lambda_n x_{n2} \\ \vdots & & & \vdots \\ \lambda_1 x_{1n} & \lambda_2 x_{2n} & \cdots & \lambda_n x_{nn} \end{bmatrix} \\
&= \begin{bmatrix} \lambda_1 x_1 & | & \cdots & | & \lambda_n x_n \end{bmatrix} \\
&= \begin{bmatrix} Ax_1 & | & \cdots & | & Ax_n \end{bmatrix} \\
&= A \begin{bmatrix} x_1 & | & \cdots & | & x_n \end{bmatrix} \\
&= AX,
\end{aligned}$$

as claimed.

14.2 Traditional Eigenvalue Problem Review

In introductory linear algebra courses you would normally compute (by hand) eigenvalues and eigenvectors using the characteristic polynomial.

Definition 14.4. *The **characteristic polynomial** of A, denoted $p_A(z)$, is the degree n (monic) polynomial given by*

$$p_A(z) = \det(zI - A).$$

Theorem 14.1. *λ is an eigenvalue of A iff $p_A(\lambda) = 0$.*

Proof.

$$\begin{aligned}
\lambda &\text{ is } \text{an eigenvalue} \\
\Leftrightarrow (\lambda I - A)x &= 0 \text{ (for some } x \neq 0), \\
\Leftrightarrow \lambda I - A &\text{ is } \text{ singular}, \\
\Leftrightarrow \det(\lambda I - A) &= 0.
\end{aligned}$$

□

The **fundamental theorem of algebra** tells us that the degree n polynomial $p_A(z)$ has n (possibly complex) roots. So A has n (possibly complex) eigenvalues, given by the roots. Therefore, we can write

$$p_A(z) = (z - \lambda_1)(z - \lambda_2)\dots(z - \lambda_n).$$

Given an eigenvalue λ, the corresponding eigenvector(s) are given by solving $(\lambda I - A)x = 0$ for x (i.e., the nullspace of $\lambda I - A$.) We will see later why the choice of A will yield real eigenvalues. Conversely, for every monic polynomial of degree n,

$$p(z) = z^n + a_{n-1}z^{n-1} + \dots + a_1 z + a_0,$$

there always exists a matrix whose eigenvalues are roots of $p(z)$. This matrix is called the **companion matrix**

$$C = \begin{bmatrix} 0 & & & & -a_0 \\ 1 & 0 & & & -a_1 \\ & 1 & 0 & & -a_2 \\ & & \ddots & \ddots & \vdots \\ & & & 1 & -a_{n-1} \end{bmatrix}.$$

The following definitions are concerned with the **multiplicity** of eigenvalues.

Definition 14.5. *The **algebraic multiplicity** of λ is the number of times it appears as a root of $p_A(z)$.*

Definition 14.6. *The **geometric multiplicity** of λ is the dimension of the nullspace of the matrix $\lambda I - A$.*

Remarks:

1. Why geometric multiplicity cannot exceed algebraic multiplicity: The geometric multiplicity is the number of linearly independent vectors, and each vector is the solution to one algebraic eigenvector equation, so there must be at least as much algebraic multiplicity.
2. If algebraic multiplicity exceeds the geometric multiplicity then λ is a **defective eigenvalue** (See Lecture Notes examples).
3. The matrix A is then called a **defective matrix**.
4. This is important because only **non**-defective matrices have eigenvalue decompositions.

Multiplicity Example Consider the following example with

$$A = \begin{bmatrix} 2 & & \\ & 2 & \\ & & 2 \end{bmatrix} \text{ and } B = \begin{bmatrix} 2 & 1 & \\ & 2 & 1 \\ & & 2 \end{bmatrix}.$$

The characteristic polynomial is $p_A(z) = (z - 2)^3$ for both matrices. Thus, all eigenvalues are $\lambda = 2$ giving an algebraic multiplicity of 3. Now since

$$(2I - A) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

any 3 linearly independent vectors span the null space. Therefore, the geometric multiplicity is 3 and hence A is non-defective. For B we have that

$$(2I - B) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

Since only $x = [1, 0, 0]^T$ is in the null space, B is **defective** (geometric multiplicity = 1).

Another Defective Eigenvalue Example:

- Let $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$.
- It is clear that A has one eigenvalue, namely 0, with algebraic multiplicity 2.
- The eigenspace for eigenvalue $\lambda = 0$ is the null space of A itself (since $0I - A = A$).
- This nullspace is clearly $span\left\{\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right\}$:

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

  so that A kills $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, and sends $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ to $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$.
- This shows that the geometric multiplicity of eigenvalue 0 is 1.

14.3 Solving Eigenvalue Problems (Naïve Approach)

Sadly, no **closed form** solutions exist for degree 5 or higher polynomials as the roots cannot be found exactly using a finite number of rational operations. Therefore we must use approximations to find their eigenvalues. This suggests the application of iterative methods.

1. Form the characteristic polynomial, $p_A(z)$.
2. Use the numerical root-finding method to extract the approximate roots/eigenvalues. (e.g bisection, Newton, etc.)

The problem with this approach, is that root-finding tends to be ill-conditioned, i.e. a small change or error in the input drastically changes the roots. More effective strategies based on finding eigendecompositions exist. We will explore some of these methods in the following lectures.

Some results about bounding eigenvalues based on the Gershgorin circle theorem will be useful later.

Theorem 14.2. *(Gershgorin circle theorem) Let A be any square matrix. The eigenvalues λ of A are located in the union of the n disks (on the complex plane) given by*

$$|\lambda - a_{ii}| \leq \sum_{j \neq i} |a_{ij}|.$$

Disks are denoted by $D(a_{ii}, R_i)$, where $R_i = \sum_{j \neq i} |a_{ij}|$.

Proof. Consider (λ, x) such that $Ax = \lambda x$ and $x \neq 0$. Now scale x such that $\|x\|_\infty = 1 = x_i$ for some i. Then

$$\lambda x_i = (Ax)_i = \sum_{j=1}^{n} a_{ij}x_j = a_{ii}x_i + \sum_{j \neq i} a_{ij}x_j.$$

Rearranging and taking the absolute value of both sides gives

$$|(\lambda - a_{ii})x_i| = \left| \sum_{j \neq i} a_{ij}x_j \right|.$$

Applying the triangle inequality, we have

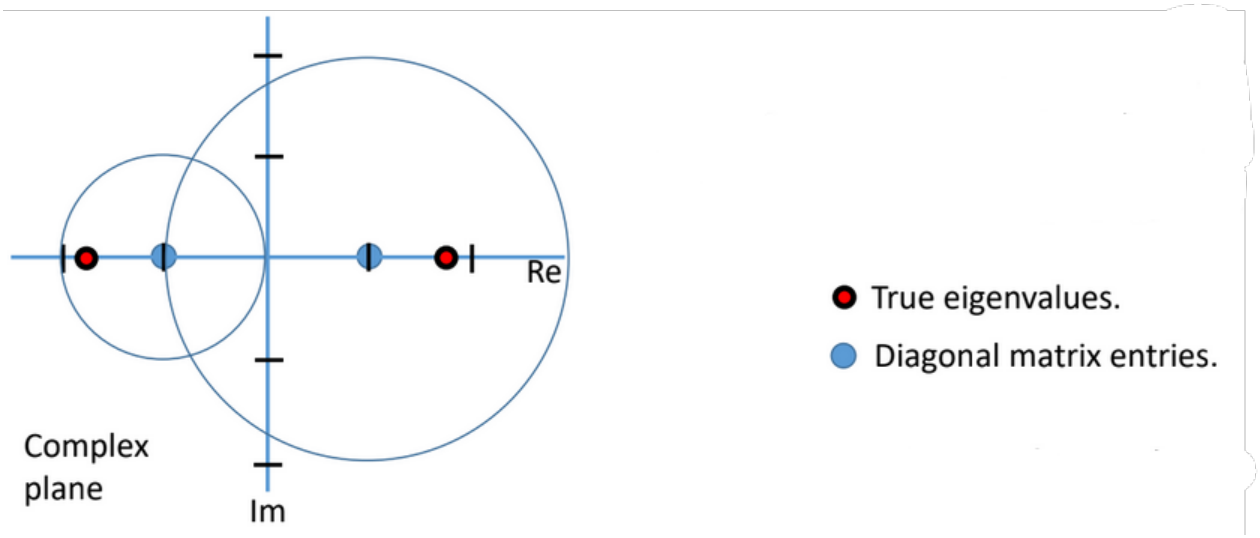
$$\begin{aligned} |(\lambda - a_{ii})| \underbrace{|x_i|}_{=1} &\leq \sum_{j \neq i} |a_{ij}x_j|, \\ |\lambda - a_{ii}| &\leq \sum_{j \neq i} |a_{ij}|, \quad \text{since } |x_j| \leq 1. \end{aligned}$$

□

The Gershgorin circle theorem essentially says the following. If off-diagonal entries in a row are small, then the corresponding eigenvalue must be close to the diagonal entry. For example, with

$$A = \begin{bmatrix} 1 & 2 \\ 1 & -1 \end{bmatrix}, \text{ which has } \Lambda(A) = \left\{ \sqrt{3}, -\sqrt{3} \right\}.$$

The figure below shows the Gershgorin disks $D(a_{ii}, \sum_{j \neq i} |a_{ij}|)$ for this example.



For this example we have Disk 1 centered at (1,0) with radius 2 or D(1,2). There is also disk 2 centered at (-1,0) with radius 1 or D(-1,1). Note that complex eigenvalues will give circles centred off the real axis.

As an exercise, find the Gershgorin disks for

$$A = \begin{bmatrix} 5 & 2 & 1 \\ 2 & 4 & -1 \\ 1 & -1 & 2 \end{bmatrix}.$$

Can we determine any of the eigenvalues exactly for A? What if A was a diagonal matrix, could the eigenvalues be determined exactly?

14.4 Eigenvalue/Eigenvector Review Example

We present an example that reviews computing the eigenvalues and eigenvectors using the characteristic polynomial. In this example we find the eigenvalues and eigenvectors for the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 2 & -1 & 2 \\ 4 & -4 & 5 \end{bmatrix}.$$

We solve $\det(\lambda I - A) = 0$ for λ's. The matrix $\lambda I - A$ is

$$\lambda I - A = \begin{bmatrix} \lambda - 1 & 0 & 0 \\ -2 & \lambda + 1 & -2 \\ 4 & -4 & \lambda - 5 \end{bmatrix}.$$

So the determinant is

$$\begin{aligned} p_A(\lambda) &= \det(\lambda I - A) \\ &= (\lambda - 1)((\lambda + 1)(\lambda - 5) - (-2)(4)), \\ &= (\lambda - 1)(\lambda^2 - 4\lambda + 3), \\ &= (\lambda - 1)(\lambda - 1)(\lambda - 3). \end{aligned}$$

Therefore, $\Lambda(A) = \{1, 3\}$ where $\lambda = 1$ has algebraic multiplicity of 2.

Now we find the eigenvectors, first we start with when $\lambda = 3$

$$\begin{aligned} (3I - A)x &= 0 \\ \Rightarrow \begin{bmatrix} 2 & 0 & 0 \\ -2 & 4 & -2 \\ -4 & 4 & -2 \end{bmatrix} x &= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \end{aligned}$$

Hence, we have

$$\begin{aligned} x_1 &= 0, \\ \Rightarrow \begin{bmatrix} 4 & -2 \\ 4 & -2 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \\ \Rightarrow 4x_2 - 2x_3 &= 0, \\ \Rightarrow x_2 &= \frac{1}{2} x_3. \end{aligned}$$

Thus, we can take

$$x_1 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix},$$

for the eigenvector corresponding to $\lambda = 3$. Now we do the same for $\lambda = 1$

$$\begin{aligned} (I - A)x &= 0, \\ \Rightarrow \begin{bmatrix} 0 & 0 & 0 \\ -2 & 2 & -2 \\ -4 & 4 & -4 \end{bmatrix} x &= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \\ \Rightarrow -x_1 + x_2 - x_3 &= 0. \end{aligned}$$

So two linearly independent eigenvectors that span the nullspace are

$$x_2 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \quad \text{and} \quad x_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

Assembling the eigendecomposition we get

$$\begin{aligned} AX &= \begin{bmatrix} 1 & 0 & 0 \\ 2 & -1 & 2 \\ 4 & -4 & 5 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 2 & -1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 1 \\ 3 & 0 & 1 \\ 6 & -1 & 0 \end{bmatrix}, \end{aligned}$$

and

$$\begin{aligned} X\Lambda &= \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} 3 & & \\ & 1 & \\ & & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 1 \\ 3 & 0 & 1 \\ 6 & -1 & 0 \end{bmatrix}. \end{aligned}$$

We will generally consider matrices $A \in \mathbb{R}^{n \times n}$ that are symmetric ($A^T = A$). Such matrices have useful properties such as real eigenvalues, and a complete set of orthogonal eigenvectors.

$$\{\lambda_1, \lambda_2, \ldots, \lambda_n\}, \{q_1, q_2, \ldots, q_n\}, \text{with } \|q_i\| = 1.$$

Therefore,

$$A = Q\Lambda Q^T,$$

where Q is orthogonal.

14.5 Rayleigh quotient

There are two quantities that must be solved for in eigenvalue problems: the **eigenvalues** and the **eigenvectors**. Consider first computing eigenvalues, when given an approximation to an eigenvector. An important quantity in this case is the **Rayleigh quotient** defined next.

Definition 14.7. *The **Rayleigh quotient** of a nonzero vector x with respect to A is*

$$r(x) = \frac{x^T A x}{x^T x}.$$

Note that if x is an eigenvector of A then $r(x) = \lambda$ since

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

Otherwise, $r(x)$ gives a scalar α that behaves "most like" an eigenvalue for a given vector x.

We can justify the definition of the Rayliegh quotient in another way. Consider the following single-variable, $n \times 1$ least squares problem for an unknown $\alpha \in \mathbb{R}$:

$$\min_{\alpha} \left\| \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \alpha - Ax \right\|_2^2,$$

for given A and x. Constructing the normal equations for this problem, we have

$$\begin{aligned} (x^T x)\alpha &= x^T(Ax), \\ \Rightarrow \quad \alpha &= \frac{x^T A x}{x^T x} \\ &= r(x). \end{aligned}$$

Consider this Rayleigh quotient example for the following matrix

$$A = \begin{bmatrix} 3 & 2 & 5 \\ 2 & 7 & 5 \\ 0 & 2 & 8 \end{bmatrix},$$

which has an eigenvector near $v \approx [0.7, -0.7, 0.3]^T$. The Rayleigh quotient gives

$$\begin{aligned}
\alpha &= \frac{v^T A v}{v^T v} \\
&= \left(\frac{1}{\begin{bmatrix} \frac{7}{10} & -\frac{7}{10} & \frac{3}{10} \end{bmatrix} \begin{bmatrix} \frac{7}{10} \\ -\frac{7}{10} \\ \frac{3}{10} \end{bmatrix}} \right) \begin{bmatrix} \frac{7}{10} & -\frac{7}{10} & \frac{3}{10} \end{bmatrix} \begin{bmatrix} 3 & 2 & 5 \\ 2 & 7 & 5 \\ 0 & 2 & 8 \end{bmatrix} \begin{bmatrix} \frac{7}{10} \\ -\frac{7}{10} \\ \frac{3}{10} \end{bmatrix} \\
&= \frac{324}{107} \\
&\approx 3.028,
\end{aligned}$$

which is close to the true value of $\lambda = 3$.

Computation of Eigenvalues and Eigenvectors:

$$\begin{aligned}
|\lambda I - A| &= \begin{vmatrix} \lambda - 3 & -2 & -5 \\ -2 & \lambda - 7 & -5 \\ 0 & -2 & \lambda - 8 \end{vmatrix} \\
&= (\lambda - 3) \begin{vmatrix} \lambda - 7 & -5 \\ -2 & \lambda - 8 \end{vmatrix} + 2 \begin{vmatrix} -2 & -5 \\ -2 & \lambda - 8 \end{vmatrix} \\
&= (\lambda - 3)\left[(\lambda - 7)(\lambda - 8) - 10\right] + 2\left[-2(\lambda - 8) - 10\right] \\
&= (\lambda - 3)\left[\lambda^2 - 15\lambda + 56 - 10\right] + 2\left[-2\lambda + 16 - 10\right] \\
&= (\lambda - 3)\left[\lambda^2 - 15\lambda + 46\right] + 2\left[-2\lambda + 6\right] \\
&= (\lambda - 3)\left[\lambda^2 - 15\lambda + 46\right] + (\lambda - 3)\left[-4\right] \\
&= (\lambda - 3) \underbrace{\left[\lambda^2 - 15\lambda + 42\right]}_{\text{eigenvalues are not real}}
\end{aligned}$$

To compute the eigenvectors for eigenvalue $\lambda = 3$, we solve the homogeneous system defined

by

$$\begin{aligned}
&= A - 3I \\
&= \begin{bmatrix} 0 & 2 & 5 \\ 2 & 4 & 5 \\ 0 & 2 & 5 \end{bmatrix} \\
&\sim \begin{bmatrix} 2 & 4 & 5 \\ 0 & 2 & 5 \\ 0 & 2 & 5 \end{bmatrix} \begin{matrix} R_1 \leftarrow R_2 \\ R_2 \leftarrow R_1 \end{matrix} \\
&\sim \begin{bmatrix} 2 & 4 & 5 \\ 0 & 2 & 5 \\ 0 & 0 & 0 \end{bmatrix} R_3 \leftarrow R_3 - R_2 \\
&\sim \begin{bmatrix} 2 & 2 & 0 \\ 0 & 2 & 5 \\ 0 & 0 & 0 \end{bmatrix} R_1 \leftarrow R_1 - 2R_2 \\
&\sim \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & \frac{5}{2} \\ 0 & 0 & 0 \end{bmatrix} \begin{matrix} R_1 \leftarrow \frac{1}{2}R_1 \\ R_2 \leftarrow \frac{1}{2}R_2 \end{matrix}
\end{aligned}$$

which has the general solution

$$\left\{ t \begin{bmatrix} 5 \\ -5 \\ 2 \end{bmatrix} : t \in \mathbb{R} \right\},$$

in other words, $\begin{bmatrix} 5 \\ -5 \\ 2 \end{bmatrix}$ is an eigenvector for eigenvalue $\lambda = 3$.

From this example we see that if we had a reasonable guess at an eigenvector, the Rayleigh quotient would be a useful approximation of the eigenvalue. In fact, the following theorem (see Trefethen & Bau, Lecture 27) states that the approximation converges quadratically.

Theorem 14.3. *Let q_j be an eigenvector, and $x \approx q_j$. Then*

$$r(x) - r(q_j) = O(\|x - q_j\|^2) \text{ as } x \to q_j.$$

That is, as $x \to q_j$, the error in the estimate of the eigenvalue λ decreases quadratically.

To summarize, eigenvalues are approximated using the Rayliegh quotient, given an approximation of an eigenvector. We will now look at how to get an approximation of the eigenvectors. We will again use **iterative** approaches.

14.6 Power Iteration

The idea of the power iteration is simple. Start with an initial vector $v^{(0)}$, then repeatedly multiply by A and normalize:

$$\begin{aligned} v^{(1)} &= \frac{Av^{(0)}}{\|Av^{(0)}\|}, \\ v^{(2)} &= \frac{Av^{(1)}}{\|Av^{(1)}\|}, \\ &\vdots \\ v^{(k)} &= \frac{Av^{(k-1)}}{\|Av^{(k-1)}\|}. \end{aligned}$$

In the limit, the approximation $v^{(k)}$ approaches q_1, where q_1 is the eigenvector associated with the **largest** magnitude eigenvalue, i.e.,

$$\lim_{k\to\infty} v^{(k)} = q_1.$$

Remarks:

1. Recall that A must be symmetric, so that there exists an orthogonal basis of eigenvectors of A.
2. If we are lucky, and start with $v^{(1)}$ an eigenvector for eigenvalue λ, depending on $sign(\lambda)$, we will get back the same vector in 1 or possibly 2 steps.

Let's prove that Power Iteration converges.

Proof. Let $v^{(0)}$ be an initial guess at the eigenvector q_1. Also let $\{q_i\}$ denote the set of orthonormal eigenvectors. Then we can write

$$v^{(0)} = c_1q_1 + c_2q_2 + \cdots + c_nq_n,$$

for some coefficients c_i, because the eigenvectors span the space. Now since $Aq_i = \lambda_i q_i$ we can write

$$Av^{(0)} = c_1\lambda_1q_1 + c_2\lambda_2q_2 + \cdots + c_n\lambda_nq_n.$$

Further multiplication by A gives

$$\begin{aligned} A^kv^{(0)} &= c_1\lambda_1^kq_1 + c_2\lambda_2^kq_2 + \cdots + c_n\lambda_n^kq_n, \\ &= \lambda_1^k\left(c_1q_1 + c_2\left(\frac{\lambda_2}{\lambda_1}\right)^k q_2 + \cdots + c_n\left(\frac{\lambda_n}{\lambda_1}\right)^k q_n\right). \end{aligned}$$

Now if we have $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$ and $c_1 = q_1^Tv^{(0)} \neq 0$, then we observe the following:

$$\left(\frac{\lambda_i}{\lambda_1}\right)^k \to 0 \text{ as } k \to \infty, \text{ for } i > 1.$$

Therefore $A^k v^{(0)} \approx c_1 \lambda_1^k q_1$, for large k. Since the eigenvectors are orthonormal, the scale factor doesn't matter. We can normalize to find q_1 as

$$q_1 \to \frac{A^k v^{(0)}}{\|A^k v^{(0)}\|} \quad \text{as } k \to \infty.$$

This is the eigenvector for the the largest magnitude eigenvalue. The next theorem summarizes this result. □

Theorem 14.4 (Power iteration convergence)**.** *Suppose* $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$*, and* $q_1^T v^{(0)} \neq 0$*, where* q_1 *is the eigenvector for* λ_1*. Then*

$$\left\|v^{(k)} - (\pm q_1)\right\| = O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right), \text{ and } |\lambda^{(k)} - \lambda_1| = O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^{2k}\right),$$

as $k \to \infty$.

This is **linear** convergence for the eigenvector with convergence factor is $\left|\frac{\lambda_2}{\lambda_1}\right|$. Convergence is therefore slow if $|\lambda_2| \approx |\lambda_1|$, i.e., if the first two eigenvalues are close in magnitude. There will be no convergence at all if $|\lambda_2| = |\lambda_1|$. Algorithm 14.13 gives pseudocode for the power iteration.

Algorithm 14.13 Power Iteration Algorithm

$v^{(0)} =$ initial guess, s.t. $\left\|v^{(0)}\right\| = 1$
for $k = 1, 2, \ldots$
  $w = Av^{(k-1)}$ ▷ Apply A
  $v^{(k)} = \frac{w}{\|w\|}$ ▷ Normalize
  $\lambda^{(k)} = \left(v^{(k)}\right)^T Av^{(k)}$ ▷ Rayleigh Quotient
end for

Remark: We have not said at all yet how to know which k we might need to make our approximation close enough.

15 Lecture 15: Eigenvectors / Eigenvalues - Iterative Methods

Outline

1. Inverse iteration
   (a) Shifting Eigenvalues
2. Rayleigh Quotient iteration
3. Computational Complexity
4. QR Iteration

In this lecture we look at three iterative methods for finding eigenvalues of a matrix A:

- power iteration,
- inverse iteration,
- Rayleigh Quotient iteration,
- QR Iteration.

15.1 Inverse Iteration

With the power iteration we can only recover q_1. What about other eigenvectors? We can find another eigenvector using the same idea of repeatedly multiplying a starting vector by a matrix. The inverse iteration can recover the eigenvector q_n associated with the **smallest** magnitude eigenvalue.

We are assuming throughout this lecture that A is invertible. This is OK, because our assumption from last time, that A is SPD, carries on throughout this lecture.

The inverse iteration instead multiplies the starting vector $v^{(0)}$ by A^{-1}. Note that if

$$\begin{aligned} Ax &= \lambda x, \text{ then} \\ x &= \lambda A^{-1}x, \text{ and so} \\ \frac{1}{\lambda}x &= A^{-1}x, \text{ assuming } \lambda \neq 0. \end{aligned}$$

Therefore, the eigenvalues of A^{-1} are the reciprocals of the eigenvalues of A:

$$\text{If } \Lambda(A) = \{\lambda_i\}, \text{ then } \Lambda(A^{-1}) = \left\{\frac{1}{\lambda_i}\right\}.$$

We can therefore use this to find q_n instead of q_1. The proof follows the same steps as the one for the power iteration.

Proof. We have that

$$\begin{aligned} A^{-1}v^{(0)} &= \frac{c_1 q_1}{\lambda_1} + \frac{c_2 q_2}{\lambda_2} + \ldots + \frac{c_n q_n}{\lambda_n} \\ A^{-k}v^{(0)} &= \frac{c_1 q_1}{(\lambda_1)^k} + \frac{c_2 q_2}{(\lambda_2)^k} + \ldots + \frac{c_n q_n}{(\lambda_n)^k} \\ &= \frac{1}{(\lambda_n)^k}\left(c_1 q_1 \left(\frac{\lambda_n}{\lambda_1}\right)^k + c_2 q_2 \left(\frac{\lambda_n}{\lambda_2}\right)^k + \ldots + c_n q_n\right) \end{aligned}$$

For large k,

$$A^{-k}v^{(0)} \approx c_n \left(\frac{1}{\lambda_n}\right)^k q_n.$$

Since the eigenvectors are orthonormal, the scale factor doesn't matter. We can normalize to find q_n as

$$q_n \to \frac{A^{-k}v^{(0)}}{\|A^{-k}v^{(0)}\|} \quad \text{as } k \to \infty.$$

This is the eigenvector for the the smallest magnitude eigenvalue. □

Note we don't actually form A^{-1}; we instead solve a linear system. The inverse iteration pseudocode is given in Algorithm 15.14.

Algorithm 15.14 (Basic) Inverse Iteration Algorithm

$v^{(0)} =$ initial guess, s.t. $\left\|v^{(0)}\right\| = 1$
for $k = 1, 2, \ldots$
  $w = A^{-1}v^{(k-1)}$ ▷ Actually, solve a linear system: ▷ $Aw = v^{(k-1)}$
  $v^{(k)} = \frac{w}{\|w\|}$
  $\lambda^{(k)} = \left(v^{(k)}\right)^T A v^{(k)}$ ▷ Rayleigh Quotient
end for

15.1.1 Shifting Eigenvalues

The inverse iteration still only allows us compute one eigenvector, q_n. "Shifting" the eigenvalues will let us find more. The idea is to use the fact that the smallest possible magnitude eigenvalue is zero! We then try to modify A so the "target" eigenvector has the smallest magnitude eigenvalue near zero. Therefore, the inverse iteration will find this target eigenvector.

Consider $B = A - \mu I$, with $\mu \neq 0$ not an eigenvalue. If A's eigenvalues/vectors are known, what are B's? Since

$$\begin{aligned} Ax &= \lambda x, \text{ we have} \\ Ax - \mu x &= \lambda x - \mu x, \text{ and so} \\ \underbrace{(A - \mu I)}_{B} x &= (\lambda - \mu)x. \end{aligned}$$

Therefore, for the matrix B we have that the

1. eigenvectors are the same,
2. eigenvalues are shifted: $\lambda_j - \mu$ for $\lambda_j \in \Lambda(A)$.

If we (somehow) expect λ_j close to μ, then $\lambda_j - \mu$ is the smallest magnitude eigenvalue of B. We can then apply the inverse iteration to find its eigenvector q_j. This is an advantage of the inverse iteration over the power iteration. With the inverse iteration we can select the specific eigenvector to recover, **if** we can choose μ close to the corresponding λ_j.

The inverse iteration has linear convergence behaviour, as stated in the next theorem.

Theorem 15.1 (Inverse iteration, with shifting, convergence)**.** *Suppose λ_J is the closest eigenvalue to μ and λ_L is the next closest, or $|\mu - \lambda_J| < |\mu - \lambda_L| \leq |\mu - \lambda_j|$, for $j \neq J$, and $q_J^T v^{(0)} \neq 0$.*

Then

$$\left\| v^{(k)} - (\pm q_J) \right\| = O\left(\left| \frac{\mu - \lambda_J}{\mu - \lambda_L} \right|^k \right), \text{ and } |\lambda^{(k)} - \lambda_J| = O\left(\left| \frac{\mu - \lambda_J}{\mu - \lambda_L} \right|^{2k} \right),$$

as $k \to \infty$.

So convergence depends on ratios of **shifted** eigenvalues rather than original eigenvalues.

If we unluckily choose $v^{(0)}$ orthogonal to q_J, so that $q_J^T v^{(0)} = 0$, we will not get convergence; instead we will get to max-iterations, without any answer. In this case, we can make a different guess for $v^{(0)}$, and try again.

Algorithm 15.15 gives pseudocode for the shifted inverse iteration.

Algorithm 15.15 (Shifted) Inverse Iteration Algorithm

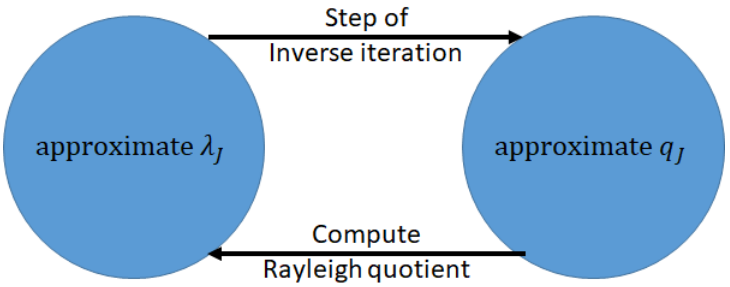
For given μ:
$v^{(0)} =$ initial guess, s.t. $\left\| v^{(0)} \right\| = 1$
for $k = 1, 2, \ldots$
  Solve $(A - \mu I)w = v^{(k-1)}$
  $v^{(k)} = \frac{w}{\|w\|}$
  $\lambda^{(k)} = \left(v^{(k)}\right)^T A v^{(k)}$ ▷ Rayleigh Quotient
end for

15.2 Rayleigh Quotient Iteration

The shifted inverse iteration needs an estimate of the eigenvalue λ_j to use for μ. Observe the following:

1. **Rayleigh quotient**, $r(v)$, estimates an eigenvalue given its approximate eigenvector v (with quadratic convergence).
2. **Inverse iteration** estimates an eigenvector given the approximate eigenvalue μ (with linear convergence, by Theorem [Inverse Iteration Convergence]).

The Rayleigh quotient iteration combines the two! It is the inverse iteration that updates μ with the latest guess for λ_j at each step (see figure below).



Pseudocode for the Rayleigh quotient iteration is given in Algorithm 15.16.

Algorithm 15.16 Rayleigh Quotient Iteration Algorithm

$v^{(0)} =$ initial guess, s.t. $\left\|v^{(0)}\right\| = 1$
$\lambda^{(0)} = \left(v^{(0)}\right)^T A v^{(0)} \left(= r(v^{(0)})\right)$ ▷ Rayleigh Quotient
for $k = 1, 2, \ldots$
  Solve $(A - \lambda^{(k-1)} I)w = v^{(k-1)}$ ▷ Using current λ estimate
  ▷ instead of fixed initial μ
  $v^{(k)} = \frac{w}{\|w\|}$
  $\lambda^{(k)} = \left(v^{(k)}\right)^T A v^{(k)}$ ▷ Rayleigh Quotient
end for

The convergence when combining the Rayliegh quotient and the inverse iteration (i.e., Rayliegh quotient iteration) is cubic.

Theorem 15.2 (Rayleigh quotient iteration convergence)**.** *RQI converges cubically for "almost all" starting vectors $v^{(0)}$. That is,*

$$\left\|v^{(k+1)} - (\pm q_J)\right\| = O\left(\left\|v^{(k)} - (\pm q_J)\right\|^3\right),$$

and

$$|\lambda^{(k+1)} - \lambda_J| = O\left(|\lambda^{(k)} - \lambda_J|^3\right).$$

In practice, each iteration roughly triples the number of digits of accuracy.

Remarks:

1. Note that "almost all" includes at least $q_J^T v^{(0)} \neq 0$.

For example, consider this matrix:

$$A = \begin{bmatrix} 21 & 7 & -1 \\ 5 & 7 & 7 \\ 4 & -4 & 20 \end{bmatrix}, \quad v^{(0)} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

The estimated eigenvalue with the Rayleigh quotient iteration is ($\lambda^{(0)}$ as initial guess)

$$\begin{array}{lllllll} \lambda^{(0)} & = & 22 & v^{(0)} & = & \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T, \\ \lambda^{(1)} & = & 24.0802 & v^{(1)} & = & \begin{bmatrix} 0.8655 & 0.3619 & 0.3462 \end{bmatrix}^T, \\ \lambda^{(2)} & = & 24.0013 & v^{(2)} & = & \begin{bmatrix} -0.8169 & -0.4079 & -0.4077 \end{bmatrix}^T, \\ \lambda^{(3)} & = & 24.00000017 & v^{(3)} & = & \begin{bmatrix} 0.8164 & 0.4082 & 0.4082 \end{bmatrix}^T. \end{array}$$

Remarks:

1. It is weird, but correct (verify it for yourself), that we get the negative of the previous and future eigenvectors, as $v^{(2)}$.

15.3 Computational Complexity

The following gives the operation counts for each of the iterative methods discussed so far.

- **Power iteration**: each step involved $Av^{(k-1)} \to O(n^2)$ flops.
- **Inverse iteration**: each step requires solving $(A - \mu I)w = v^{(k-1)}$. This **would** be $O(n^3)$ per step, however, we can pre-factor into L and U at the start (Recall, at cost: $\frac{2}{3}n^3 + O(n^2)$). Then we just do forward/backward solves for each iteration. Hence, we have $O(n^2)$ flops per step.
- **Rayleigh quotient iteration**: Matrix $A - \lambda^{(k-1)}I$ changes at each step so we can not pre-factor. Therefore, for RQI we have $O(n^3)$ flops per iteration.
- For all three of the methods, if A is tridiagonal the operation counts reduce to $O(n)$ flops.

15.4 QR Iteration

In the previous sections we looked at the power, inverse, and Rayleigh quotient iterations for finding a **single** eigenvector/eigenvalue. Our next goal is to find **more than one** eigenvector/eigenvalue pair at a time.

First some definitions are needed.

Definition 15.1. *Matrices A and B are **similar** if $B = X^{-1}AX$ for some non-singular X.*

Definition 15.2. *If $X \in \mathbb{R}^{n\times n}$ is nonsingular, then $A \to X^{-1}AX$ is called a **similarity transformation** of A.*

Theorem 15.3. *If matrices A and B are similar, then they have the same characteristic polynomial, and hence the same eigenvalues.*

Proof.

$$\begin{aligned} p_B(z) &= \det(zI - X^{-1}AX), \\ &= \det(X^{-1}(zI - A)X), \\ &= \det(X^{-1})\det(zI - A)\det(X), \\ &= \det(zI - A), \text{ since } \det(X^{-1}) = \frac{1}{\det(X)}, \\ &= p_A(z). \end{aligned}$$

□

The idea is to apply a sequence of similarity transformations to A that converge to a **diagonal** matrix, which has the eigenvalues on its diagonal. Recall that we are only considering real symmetric matrices. To achieve this we will rely on the QR factorization again.

Fun Fact: In 2000, the QR Iteration was named one of the top ten most important algorithms for science and engineering developed in the 20th century.

Given $A^{(k-1)}$, we factor it:

$$\begin{aligned} A^{(k-1)} &= Q^{(k)}R^{(k)}, \text{ so that} \\ \left(Q^{(k)}\right)^T A^{(k-1)} &= R^{(k)}. \end{aligned}$$

Defining the next matrix, $A^{(k)}$, as $R^{(k)}Q^{(k)}$, then yields a similarity transformation:

$$\begin{aligned} A^{(k)} &:= R^{(k)}Q^{(k)} \\ &= \left(Q^{(k)}\right)^T A^{(k-1)}Q^{(k)}. \end{aligned}$$

Therefore, $A^{(k-1)}$ and $A^{(k)}$ are similar. The QR Iteration simply repeats this process as shown in Algorithm 15.17.

Algorithm 15.17 Basic QR Iteration

$A^{(0)} = A$
for $k = 1, 2, \ldots$
  $Q^{(k)}R^{(k)} = A^{(k-1)}$ ▷ Compute QR factors of $A^{(k-1)}$
  $A^{(k)} = R^{(k)}Q^{(k)}$ ▷ "Recombine" in reverse order
end for

That's it! We just compute the QR factorization of $A^{(k-1)} = QR$ and reverse the order to construct $A^{(k)} = RQ$. Eventually $A^{(k)}$ becomes diagonal, with the eigenvalues of A on the diagonal. As $A^{(k)}$ converges to eigenvalues on the diagonal (we will justify why this happens in the next lecture), the product of the $Q^{(k)}$'s gives the set of eigenvectors. That is, denoting

$$\underline{Q}^{(k)} = Q^{(1)}Q^{(2)}\ldots Q^{(k)},$$

we have the relation

$$A^{(k)} = \left(\underline{Q}^{(k)}\right)^T A\underline{Q}^{(k)}.$$

Consider the following QR Iteration example with

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 4 \end{bmatrix} = A^{(0)}.$$

One can verify (say using Matlab) that A is SPD.

We can use this Matlab code to compute the first three QR iterations:

```
A0 = [2 1 1 ; 1 3 1 ; 1 1 4];
[Q1,R1] = qr(A0);
A1 = R1 * Q1;
[Q2,R2] = qr(A1);
A2 = R2 * Q2;
[Q3,R3] = qr(A2);
A3 = R3 * Q3;
```

Our results are then

$$\begin{aligned} A^{(1)} &= \begin{bmatrix} 4.1667 & 1.0954 & -1.2671 \\ 1.0954 & 2.0000 & 0.0000 \\ -1.2671 & 0.0000 & 2.8333 \end{bmatrix}, \\ A^{(2)} &= \begin{bmatrix} 5.0909 & 0.1574 & 0.6232 \\ 0.1574 & 1.8618 & -0.5470 \\ 0.6232 & -0.5470 & 2.0473 \end{bmatrix}, \\ A^{(3)} &= \begin{bmatrix} 5.1987 & -0.0759 & -0.2073 \\ -0.0759 & 2.1818 & 0.4966 \\ -0.2073 & 0.4966 & 1.6195 \end{bmatrix}. \end{aligned}$$

The true solution for this matrix is $\Lambda(A) = \{5.2143, 2.4608, 1.3249\}$. At each iteration above the off-diagonals get closer to zero. Moreover, the diagonal entires are converging to the true eigenvalues.

Remarks:

1. QR Iteration is mathematically sound, but not good computationally.
2. Do not implement this algorithm!
3. In the next lecture we will discuss an equivalent algorithm that is computationally better.

16 Lecture 16: Eigenvectors / Eigenvalues - Practical QR

Outline

1. Simultaneous (aka Block Power) Iteration
2. Simultaneous Iteration vs. QR Iteration
   (a) Convergence of QR Iteration
   (b) Eigenvalue Problems Recap
3. Reduction to Upper Hessenberg
4. Aside: The QR Iteration's Inventors

16.1 Simultaneous (aka Block Power) Iteration

Motivating Question: How can a simple algorithm possibly work to give us all eigenvector/eigenvalue pairs? We first consider the simpler-to-analyze **simultaneous iteration**. Then, we argue that the simultaneous iteration is **equivalent** to the QR Iteration.

Simultaneous iteration (aka **block** power iteration) is when we apply power iteration to **several** vectors at once, while maintaining linear independence among them. Start with a set of p orthonormal vectors, $v_1^{(0)}, v_2^{(0)}, \ldots, v_p^{(0)}$. The matrix multiplication $A^k v_1^{(0)}$ converges to q_1 as $k \to \infty$, where $|\lambda_1|$ is the largest (as seen in Lecture 14). However, span $\left\{A^k v_1^{(0)}, \ldots, A^k v_p^{(0)}\right\}$ also converges to span $\{q_1, q_2, \ldots, q_p\}$, where $\lambda_1, \ldots, \lambda_p$ are the p largest magnitude eigenvalues.

In matrix form, denote $V^{(0)} = \begin{bmatrix} v_1^{(0)} & v_2^{(0)} & \cdots & v_p^{(0)} \end{bmatrix}$ and $V^{(k)} = A^k V^{(0)}$. With the multiplication of $A^k V^{(0)}$ all the vectors are ultimately converging to (multiples of) q_1. This provides a **very** ill-conditioned basis for the space of eigenvectors. The solution is to **orthonormalize** the vectors at each step using QR factorization.

Remarks:

1. Our algorithm would fall apart if we ever had $A^k v_\ell^{(0)} = A^k v_m^{(0)}$, where $\ell \neq m$. Why can this not happen? This would mean that A maps different input vectors to the same output vector. In other words, A is not of full rank.

Algorithm 16.18 gives pseudocode for the simultaneous iteration.

Algorithm 16.18 Simultaneous Iteration Algorithm

Pick initial $\hat{Q}^{(0)} \in \mathbb{R}^{n \times p}$ with orthonormal columns
for $k = 1, 2, \ldots$
  $Z^{(k)} = A\hat{Q}^{(k-1)}$ ▷ (Block) power iteration step
  $Z^{(k)} = \hat{Q}^{(k)}\hat{R}^{(k)}$ ▷ (Reduced) QR factorization
end for

The column spaces of $\hat{Q}^{(k)}$ and $Z^{(k)}$ are the same, and they both are equal to the column

space of $A^k\hat{Q}^{(0)}$ (could prove by induction on k).

Similar to power iteration, the simultaneous iteration relies on two assumptions:

1. The leading $p+1$ eigenvalues are distinct in absolute value, i.e.

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_p| > |\lambda_{p+1}| \geq |\lambda_{p+2}| \geq \cdots \geq |\lambda_n|, \text{ and}$$

2. all of the leading principal submatrices of $(\hat{Q}^{(0)})^T V^{(0)}$ are non-singular (i.e. none of the p-many initial guess vectors comprising $V^{(0)}$ is orthogonal to the subspace generated by the first p-many eigenvectors).

With the above assumptions we have the following theorem that states that the simultaneous iteration converges linearly.

Theorem 16.1 (Simultaneous iteration convergence)**.** *Suppose the simultaneous iteration is applied and the preceding two assumptions are satisfied. Then as $k \to \infty$,*

$$\left\| q_j^{(k)} - (\pm q_j) \right\| = O(c^k), \quad j = 1, 2, \cdots, p,$$

where $c = \max_{1 \leq k \leq p} \left| \frac{\lambda_{k+1}}{\lambda_k} \right| < 1$.

Remarks:

1. Why we need $\pm$ in front of the true eigenvector, q_j, here: We might converge to the negative of the true eigenvector, which is fine. In this case, without $\pm$, the convergence would not work as stated.
2. The ratio $\left| \frac{\lambda_{k+1}}{\lambda_k} \right|$ is present for the same reasons as in Power Iteration.

16.2 Simultaneous Iteration vs. QR Iteration

In this section we show that the QR Iteration is identical to the simultaneous iteration when $\hat{Q}^{(0)} = I$ and $p = n$. Since the matrices are square, we will drop the hats on $\hat{Q}$ and $\hat{R}$. We will use the following notation:

- $Q^{(k)}$ for Q's from QR Iteration,
- $\underline{Q}^{(k)}$ for Q's from simultaneous iteration, i.e. $\underline{Q}^{(k)} = Q^{(1)}Q^{(2)} \cdots Q^{(k)}$, similar to the Householder notation.

Consider the QR Iteration and simultaneous iteration algorithms shown side by side below. We add steps (C) and (D) just for the upcoming proof of Theorem 16.2.

Algorithm 16.19 Simultaneous Iteration

$\underline{Q}^{(0)} = I$
for $k = 1, 2, \ldots$
  $Z^{(k)} = A\underline{Q}^{(k-1)}$ ▷ (A)
  $Z^{(k)} = \underline{Q}^{(k)} R^{(k)}$ ▷ (B)
  $A^{(k)} = \left(\underline{Q}^{(k)}\right)^T A\underline{Q}^{(k)}$ ▷ (C)
  $\underline{R}^{(k)} = R^{(k)} R^{(k-1)} \cdots R^{(1)}$ ▷ (D)
end for

Algorithm 16.20 QR Iteration

$A^{(0)} = A$
for $k = 1, 2, \ldots$
  $A^{(k-1)} = Q^{(k)} R^{(k)}$ ▷ (A)
  $A^{(k)} = R^{(k)} Q^{(k)}$ ▷ (B)
  $\underline{Q}^{(k)} = Q^{(1)} Q^{(2)} \cdots Q^{(k)}$ ▷ (C)
  $\underline{R}^{(k)} = R^{(k)} R^{(k-1)} \cdots R^{(1)}$ ▷ (D)
end for

Q & A

1. In the Simultaneous Iteration algorithm, should line (A) be corrected to
$$Z^{(k)} = A^{(k-1)}\underline{Q}^{(k-1)}?$$
A: No. $\underline{Q}^{(k)}$ changes at ever iteration. The original A is used to compute $Z^{(k)}$ from $\underline{Q}^{(k-1)}$. The given notation is correct.
2. Then why do we need $A^{(k)}$ here at all?
A: We don't need it for the computation itself. We will need it later, for convergence purposes.

Theorem 16.2. *The QR Iteration and simultaneous iteration algorithms generate identical sequences of matrices,* $\underline{R}^{(k)}, \underline{Q}^{(k)}, A^{(k)}$ *satisfying*

$$\begin{aligned} A^k &= \underline{Q}^{(k)}\underline{R}^{(k)}, \text{ (QR factorization of } \mathbf{k^{th}} \text{power of } A) \\ A^{(k)} &= \left(\underline{Q}^{(k)}\right)^T A\underline{Q}^{(k)}. \text{ (Similarity transform of } A) \end{aligned}$$

Remark: As a consequence, convergence for simultaneous iteration will work the same as it does for QR iteration.

Proof. We will show

(1) $A^k = \underline{Q}^{(k)}\underline{R}^{(k)}$, and
(2) $A^{(k)} = (\underline{Q}^{(k)})^T A Q^{(k)}$,

separately for both algorithms by induction on k. Note the distinction that A^k is a matrix exponential ($A^k = \underbrace{AA\cdots A}_{k \text{ times}}$), whereas, $A^{(k)}$ is a matrix on the kth iteration.

The base case $k = 0$ is trivial.

(1) $A^0 = \underline{Q}^{(0)} = \underline{R}^{(0)} = I$,
(2) $A^{(0)} = A$.

Now, assuming the equations hold for $k-1$, we will now show they hold for k.

Simultaneous Iteration:

(1)

$$\begin{aligned}
& A^k \\
= \; & AA^{k-1}, \\
= \; & A\underline{Q}^{(k-1)}\underline{R}^{(k-1)}, \text{ by inductive hyp. (1)}, A^{k-1} = \underline{Q}^{(k-1)}\underline{R}^{(k-1)} \\
= \; & \underline{Q}^{(k)}R^{(k)}\underline{R}^{(k-1)}, \text{ by alg. (A) and (B)}, A\underline{Q}^{(k-1)} = Z^{(k)} = \underline{Q}^{(k)}R^{(k)} \\
= \; & \underline{Q}^{(k)}\underline{R}^{(k)}. \text{ by def. } R^{(k)} \text{ as in alg. (D)}
\end{aligned}$$

(2) holds directly by (C).

QR Iteration:

(1)

$$\begin{aligned}
& A^k \\
= \; & AA^{k-1}, \\
= \; & A\underline{Q}^{(k-1)}\underline{R}^{(k-1)}, \text{ by inductive hyp. (1)} \\
= \; & \underline{Q}^{(k-1)}A^{(k-1)}\underline{R}^{(k-1)}, \text{ by inductive hyp. (2), i.e., } A\underline{Q}^{(k-1)} = \underline{Q}^{(k-1)}A^{(k-1)} \\
= \; & \underline{Q}^{(k-1)}\left(Q^{(k)}R^{(k)}\right)\underline{R}^{(k-1)}, \text{ by alg. (A)} \\
= \; & \underline{Q}^{(k)}\underline{R}^{(k)}. \text{ by def. } \underline{Q}, \underline{R}^{(k)} \text{ in (C), (D)}
\end{aligned}$$

(2)

$$\begin{aligned}
& A^{(k)} \\
= \; & R^{(k)}Q^{(k)}, \text{ by alg. (B)} \\
= \; & \left(Q^{(k)}\right)^T A^{(k-1)}Q^{(k)}, \text{ by alg. (A), i.e., } A^{(k-1)} = Q^{(k)}R^{(k)} \\
= \; & \left(Q^{(k)}\right)^T \left(\underline{Q}^{(k-1)}\right)^T A\underline{Q}^{(k-1)}Q^{(k)}, \text{ by inductive hyp. (2)} \\
= \; & \left(Q^{(k)}\right)^T A\left(Q^{(k)}\right). \text{ by def. } \underline{Q}^{(k)} \text{ in (C)}
\end{aligned}$$

Therefore the two algorithms yield the same matrix sequences for $\underline{R}^{(k)}, \underline{Q}^{(k)}$ and $A^{(k)}$. □

16.2.1 Convergence of QR Iteration

Observe that $A^k = \underline{Q}^{(k)}\underline{R}^{(k)}$ implies the QR Iteration is computing QR factors of A^k, i.e., an orthonormal basis of A^k. Also, $A^{(k)} = \left(\underline{Q}^{(k)}\right)^T A\underline{Q}^{(k)}$ implies that diagonal entries of $A^{(k)}$ are the **Rayleigh quotients** for column vectors in $\underline{Q}^{(k)}$. Recall the Rayleigh quotient is $r(x) = \frac{x^TAx}{x^Tx}$. As the columns of $\underline{Q}^{(k)}$ approach eigenvectors, these Rayleigh quotients approach the corresponding eigenvalues.

What about off-diagonal entries of $A^{(k)}$? That is, with $i \neq j$

$$A_{ij}^{(k)} = \left(\underline{q_i}^{(k)}\right)^T A \underline{q_j}^{(k)},$$

where $\underline{q_i}^{(k)}, \underline{q_j}^{(k)}$ are columns of $\underline{Q}^{(k)}$. As $\underline{q_i}^{(k)}, \underline{q_j}^{(k)}$ converge to (orthonormal) eigenvectors, q_i, q_j, then

$$A_{ij}^{(k)} \approx q_i^T A q_j = q_i(\lambda q_j) \approx 0, \text{ for } i \neq j.$$

Hence, $A^{(k)}$ converges to a diagonal matrix.

Theorem 16.3 (QR Iteration convergence)**.** *Assume $|\lambda_1| > |\lambda_2| > \cdots |\lambda_n|$ and the corresponding eigenvector matrix Q has nonsingular leading principal submatrices. Then, as $k \to \infty$, $A^{(k)}$ converges linearly to diag$(\lambda_1, \lambda_2, \ldots, \lambda_n)$ with constant*

$$C = \max_k \left| \frac{\lambda_{k+1}}{\lambda_k} \right|.$$

The matrix $\underline{Q}^{(k)}$ also converges linearly to Q with the same constant C.

For more details about the QR Iteration see Lecture 28 of Trefethen & Bau.

Remarks:

1. Why we need $\pm$ in front of the true eigenvector here: We might converge to the negative of the true eigenvector, which is fine. In this case, without $\pm$, the convergence would not work as stated.
2. The ratio $\left|\frac{\lambda_{k+1}}{\lambda_k}\right|$ is present for the same reasons as in Power Iteration.

16.2.2 Eigenvalue Problems Recap

Here we give a recap of what we have discussed so far for eigenvalue problems.

- We used the Rayleigh quotient to recover an estimated eigenvalue, given an estimated eigenvector.
- We saw the power iteration, inverse iteration, shifted inverse iteration, and Rayleigh quotient iteration, as methods to recover **individual** eigenvectors.
- We introduced two schemes to find **multiple** eigenvectors/eigenvalues at once:
  - QR Iteration,
  - Simultaneous (block power) iteration.

Dense QR factorization at **every single step** of the QR Iteration algorithm is costly. This takes approximately $\frac{4}{3}n^3$ flops. In the next section we look at a way to make the QR Iteration more practical (efficient).

The idea is to first pre-process A (with another similarity transform) to increase sparsity! If A is non-symmetric, one can reduce to upper Hessenberg form. Upper Hessenberg matrices only require $\to O(n^2)$ flops for QR factorization. If A is symmetric we can reduce to a tridiagonal matrix, which requires only $\to O(n)$ flops for QR factorization. Exercise: derive efficient QR factorizations of UH and tridiagonal matrices.

16.3 Reduction to Upper Hessenberg

In the general case A can be non-symmetric. One might ask why would we reduce to just upper Hessenberg form and not triangular? Wouldn't having a triangular matrix be even cheaper for QR factorization? Let's consider this by attempting to reduce to triangular instead.

16.3.1 First attempt:

Try to reduce A to **triangular** via usual Householder. Apply Householder Q_1 to A.

$$A = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{Q_1^T \times} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} = Q_1^T A$$

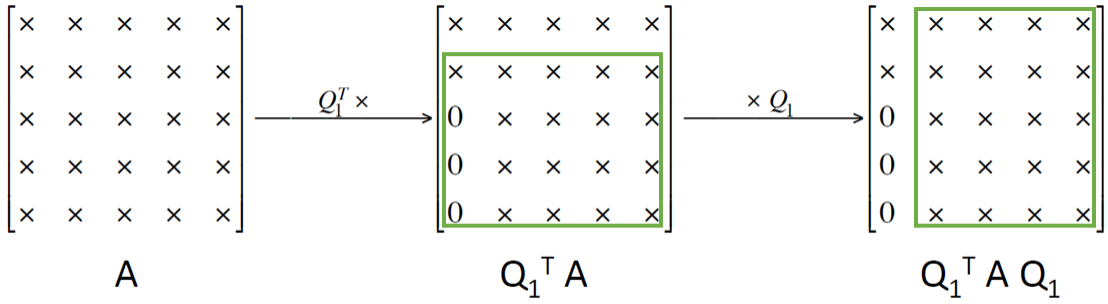
But to maintain similarity, also need to multiply by Q_1 on the **right**

$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{\times Q_1} \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}$$

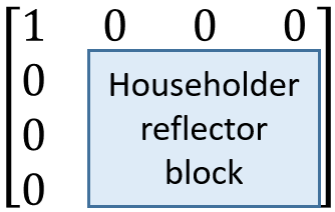
So with Householder reflections our newly created zeros are just destroyed again!

16.3.2 Second Attempt:

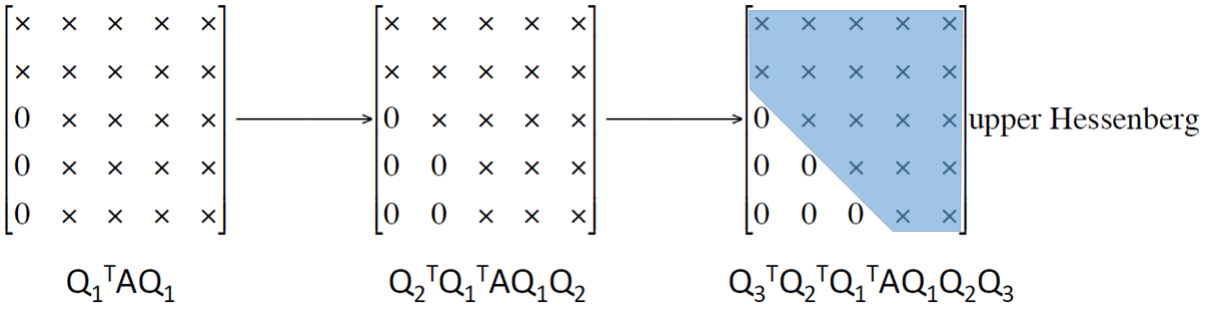
We will be a little less ambitious and choose a different Q_1^T that leaves the **whole row untouched**. Then, when we multiply by Q_1 on the right, it won't destroy our progress (i.e., it will leave the 1^{st} column alone).



To achieve this, the first Q matrix will have a form like:



This leaves the desired first row/column alone after computing $Q_1^T A Q_1$, preserving the new zeros. We apply the same idea to subsequent columns (similar to Householder QR).



This gives $Q = Q_1 Q_2 \cdots Q_{n-2}$ and $Q^T A Q =$ an upper Hessenberg matrix. Algorithm 16.21 gives the pseudocode for the reduction to Hessenberg form.

Algorithm 16.21 Reduction to Hessenberg

for $k = 1, 2, \ldots, n-2$
  $x = A(k+1:n, k)$
  $v_k = \text{sign}(x_1)\|x\|e_1 + x$ ▷ Householder reflection
  $v_k = \frac{v_k}{\|v_k\|}$ ▷ Normalize
  for $j = k, k+1, \ldots, n$ ▷ Left multiply, $Q_k^T \times$
    $A(k+1:n, j) = A(k+1:n, j) - 2v_k \left(v_k^T A(k+1:n, j)\right)$
  end for
  for $j = 1, 2, \ldots, n$ ▷ Right multiply, $\times Q_k$
    $A(i, k+1:n) = A(i, k+1:n) - 2\left(A(i, k+1:n)v_k\right)v_k^T$
  end for
end for

The cost is flops(Reduction to Hessenberg) $\approx \frac{10}{3}n^3$. However, this reduction to Hessenberg is done only **once**, before QR Iteration.

16.3.3 Symmetric Matrices: Two-Phase Process

For the symmetric case, when $A = A^T$, then

$$(Q^T A Q)^T = Q^T A Q,$$

is also symmetric. A matrix that is both symmetric **and** upper Hessenberg is necessarily tridiagonal. Reduction will produce zeros above the diagonal as well. Sparsity and symmetry

together reduce the cost of reduction from $\frac{10}{3}n^3$ to $\frac{4}{3}n^3$. The idea then to make the QR Iteration more practical is a two-phased process:

1. Reduce A to tridiagonal via Householder operations (direct).
2. Perform QR Iteration until convergence (iterative).

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}} \xrightarrow[\text{Reduction}]{\text{Phase 1}} \underset{T = Q^T A Q}{\begin{bmatrix} \times & \times & & \\ \times & \times & \times & \\ & \times & \times & \times \\ & & \times & \times \end{bmatrix}} \xrightarrow[\text{QR Iteration}]{\text{Phase 2}} \underset{D}{\begin{bmatrix} \times & & & \\ & \times & & \\ & & \times & \\ & & & \times \end{bmatrix}}$$

QR Iteration can be additionally improved by:

- Applying shifting to achieve cubic convergence rates (similar to Rayleigh Quotient Iteration).
- Breaking $A^{(k)}$ into sub-matrices once an eigenvalue is found ("deflation").

16.4 Aside: The QR Iteration's Inventors

John Francis published the (implicit, shifted) QR algorithm in 1961. It was named one of the ten "most important" algorithms of the 20th century. John Francis left the field of numerical analysis that same year. He was tracked down in 2007, and had no idea the huge influence of his work! **Concurrently**, the algorithm was invented by Vera Kublanovskaya, who continued to work in numerical analysis until passing away in 2012.

John Francis

Vera Kublanovskaya

17 Lecture 17: Eigenvectors / Eigenvalues - Image Segmentation

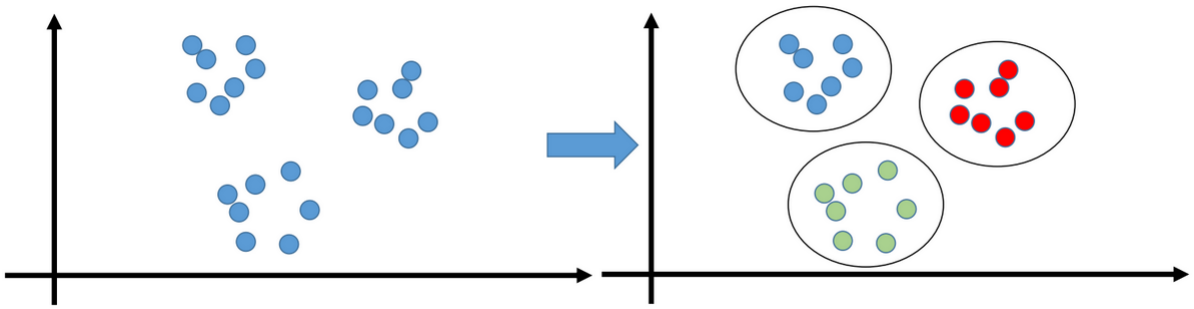
Outline

1. Definitions
2. Graph Laplacians
   (a) Unnormalized Graph Laplacian
   (b) Normalized Graph Laplacian
3. Clustering using Graph Laplacians
   (a) Relaxation of RatioCut via Graph Laplacian
   (b) Relaxation of Ncut via Graph Laplacian
4. K-means Clustering
5. Spectral Clustering: Cuts and K-means Together
   (a) Choosing Weights W
6. Other Applications
   (a) Geometric Mesh Processing
   (b) Motion Analysis

In this lecture we will take a look at the application of eigenvalue problems in **image segmentation**. First we will given some definitions and discuss the graph Laplacian. Then we will make use of the graph Laplacian in spectral clustering.

Motivation: Divide and conquer, say for image de-noising.

Spectral clustering is a family of techniques that use the eigendecomposition of a matrix to identify clusters/groups of "similar" or related elements in a dataset. See for example the figure below.



Segmentation tasks (i.e., identifying distinct parts of an image or shape) can rely on clustering.

- **Image segmentation** tries to group similar and nearby pixels.
- **Shape segmentation** tries to identify distinct parts of an object.

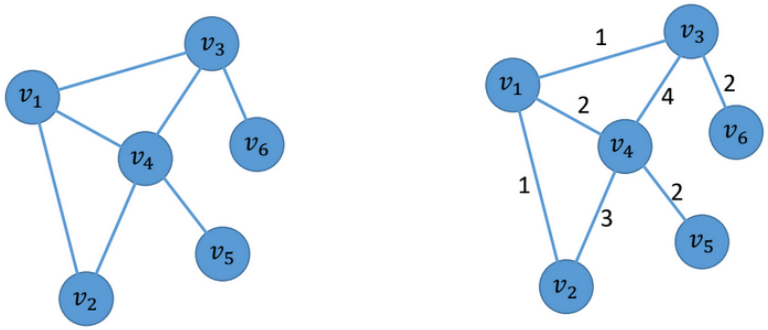


Figure 17.29: Example graph $G(V, E)$ (left) and the same graph with weighted edges (right).

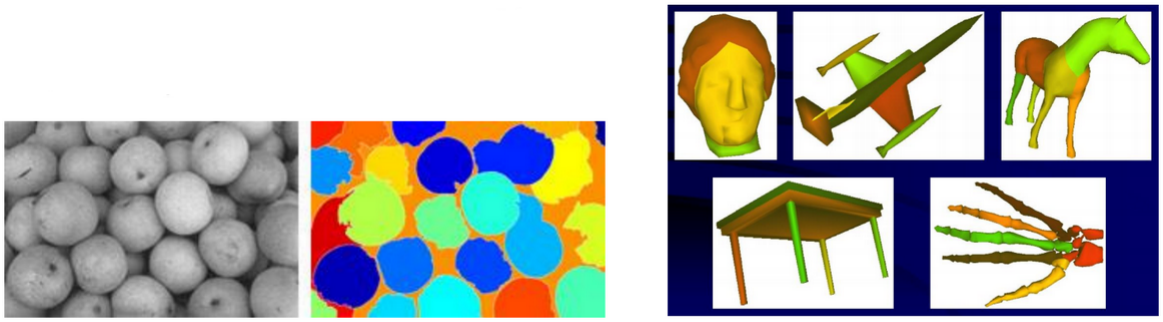


Image Segmentation

Shape Segmentation

17.1 Definitions

Consider an undirected graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is a set of vertices and $E = \{e_{ij}\}$ is a set of edges. That is, the edge between vertices v_i and v_j is denoted e_{ij}.

Definition 17.1. *The graph G is a **weighted** graph if each edge e_{ij} has an associated weight $w_{ij} \geq 0$. We denote by $W = w_{ij}$ the **weighted adjacency matrix** of the graph.*

Figure 17.29 gives an example of an undirected graph and a weighted version of the graph. What is the weight matrix W for this graph in Figure 17.29 (right)? The matrix W has zeros in entries (i, j) that do not have edges joining v_i to v_j. There are nonzeros equal to the weights for any (i, j) that does have an edge, thus

$$W = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 4 & 0 & 2 \\ 2 & 3 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}.$$

- This W is **different** from our earlier adjacency matrix, where diagonal entries could be non-zero.

The definition of the vertex degree must now be altered to include the weights of edges.

Definition 17.2. *The **degree** of a vertex v_i is given by*

$$d_i = \sum_{j=1}^{n} w_{ij},$$

*where $D = diag(d_i)$ is the **degree matrix**.*

This definition of degree is different from our earlier definition of degree, from matrix reordering.

For the example in Figure 17.29 what is the degree of v_4? It is just the sum of the fourth column of W, so $\deg(v_4) = d_4 = 11$. You can do this for all the vertices and construct the degree matrix as

$$D = \begin{bmatrix} 4 & & & & & \\ & 4 & & & & \\ & & 7 & & & \\ & & & 11 & & \\ & & & & 2 & \\ & & & & & 2 \end{bmatrix}.$$

The indicator vector is useful when working with a subset of the graph.

Definition 17.3. *Given a subset $A \subset V$, we define the **indicator vector** $\mathbf{1}_A = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}^T$ such that*

$$x_i = \begin{cases} 1 \text{ if } v_i \in A, \\ 0 \text{ if } v_i \notin A. \end{cases}$$

Definition 17.4. *Given two subsets A, B, we define $W(A, B)$ to be the total weight of all the edges starting in A and ending in B, i.e.,*

$$W(A, B) = \sum_{i \in A, j \in B} w_{ij}.$$

For example, consider the two subsets of vertices, $A = \{v_1, v_2, v_6\}$ and $B = \{v_3, v_4, v_5\}$, in Figure 17.29. What are the indicator vectors $\mathbf{1}_A$ and $\mathbf{1}_B$? They are given by

$$\mathbf{1}_A = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{1}_B = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

What is $W(A, B) = \sum_{i \in A, j \in B} w_{ij}$ for this example? We have

$$W(A, B) = w_{13} + w_{14} + w_{15} + w_{23} + w_{24} + w_{25} + w_{63} + w_{64} + w_{65} = 8.$$

We will consider two ways to measure the **size** of a subset $A \subset V$.

Definition 17.5. *This size of a subset is can be defined in terms of the number of vertices*

$$|A| = \textit{number of vertices in } A,$$

or the degrees of the vertices

$$\textit{vol}(A) = \sum_{i \in A} d_i = \textit{sum of (weighted) degrees of vertices in } A.$$

For example, consider again the graph in Figure 17.29 and the two subsets of vertices $A = \{v_1, v_2, v_6\}$ and $B = \{v_3, v_4, v_5\}$. We have that $|A| = 3$ and $|B| = 3$. Furthermore, we have that $\text{vol}(A) = 10$ and $\text{vol}(B) = 20$.

17.2 Graph Laplacians

The graph Laplacian is a generalization of our finite difference discrete Laplacian operator to arbitrary graphs. We will consider two variants: the **unnormalized** graph Laplacian

$$L = D - W,$$

and the **normalized** graph Laplacian

$$\hat{L} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}.$$

For example, with the graph in Figure 17.29 we have

$$\begin{aligned} L &= D - W \\ &= \begin{bmatrix} 4 & & & & & \\ & 4 & & & & \\ & & 7 & & & \\ & & & 11 & & \\ & & & & 2 & \\ & & & & & 2 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 4 & 0 & 2 \\ 2 & 3 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 4 & -1 & -1 & -2 & 0 & 0 \\ -1 & 4 & 0 & -3 & 0 & 0 \\ -1 & 0 & 7 & -4 & 0 & -2 \\ -2 & -3 & -4 & 11 & -2 & 0 \\ 0 & 0 & 0 & -2 & 2 & 0 \\ 0 & 0 & -2 & 0 & 0 & 2 \end{bmatrix}. \end{aligned}$$

Note that the general matrix pattern is similar to the finite difference discrete Laplacian. The diagonal entries are all positive, while off-diagonals are negative. Moreover, the sum of every row in L is zero.

17.2.1 Unnormalized Graph Laplacian

The following theorem gives some properties of the unnormalized graph Laplacian.

Theorem 17.1. *The unnormalized graph Laplacian L satisfies:*

1. *For any vector x,*
$$x^T L x = \frac{1}{2}\sum_{i,j=1}^{n} w_{ij}(x_i - x_j)^2,$$
2. *L is symmetric and positive **semi**-definite,*
3. *The smallest eigenvalue of L is 0, with corresponding eigenvector being the constant one vector $\mathbf{1} = [1, 1, \ldots, 1]^T$,*
4. *L has n non-negative eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \lambda_3 \leq \cdots \leq \lambda_n$.*

Proof. 1.

$$\begin{aligned}
x^T L x &= x^T D x - x^T W x, \\
&= \sum_{i=1}^{n} d_i x_i^2 - \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j w_{ij}, \\
&= \frac{1}{2}\left(2\sum_{i=1}^{n} d_i x_i^2 - 2\sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j w_{ij}\right), \\
&= \frac{1}{2}\left(\sum_{i=1}^{n} d_i x_i^2 - 2\sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j w_{ij} + \sum_{j=1}^{n} d_j x_j^2\right), \\
&= \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(x_i - x_j)^2.
\end{aligned}$$

2. The property 1. implies $x^T L x \geq 0$, i.e., positive semi-definite, since $w_{ij} \geq 0$. It is symmetric because D and W are symmetric.
3. Since row sums of L are zero we have $Lx = 0$ when $x = \begin{bmatrix}1, 1, \ldots, 1\end{bmatrix}^T$. Therefore, the smallest magnitude eigenvalue is zero with eigenvector $x = \begin{bmatrix}1, 1, \ldots, 1\end{bmatrix}^T$.
4. Since L is positive semi-definite its eigenvalues are non-negative. Positive semi-definite implies that $x^T A x \geq 0$. So if $Ax = \lambda x$, then $x^T A x = x^T(\lambda x) = \lambda \|x\|^2$.

$$\begin{aligned}
\Rightarrow \lambda\|x\|^2 &\geq 0, \\
\Rightarrow \lambda &\geq 0.
\end{aligned}$$

□

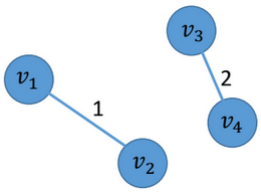
17.2.2 Normalized Graph Laplacian

The next theorem gives properties of the normalized graph Laplacian.

Theorem 17.2. *The normalized graph Laplacian $\hat{L}$ satisfies:*

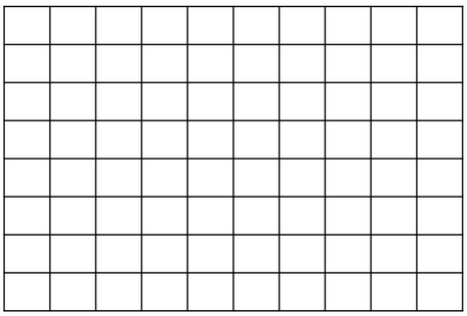
1. *For any vector* x,
$$x^T \hat{L} x = \frac{1}{2} \sum_{i,j=1}^{n} w_{ij} \left(\frac{x_i}{\sqrt{d_i}} - \frac{x_j}{\sqrt{d_j}} \right)^2,$$
2. $\hat{L}$ *is symmetric and positive* ***semi****-definite,*
3. *The smallest eigenvalue of* $\hat{L}$ *is 0 and the corresponding eigenvector is* $D^{\frac{1}{2}}\mathbf{1}$,
4. $\hat{L}$ *has* n *non-negative eigenvalues* $0 = \lambda_1 \leq \lambda_2 \leq \lambda_3 \leq \cdots \leq \lambda_n$.

The **multiplicity** k of the eigenvalue 0 for both L and $\hat{L}$ equals the **number of connected components** $A_1, \ldots, A_k$ in the graph. For example, the L and Λ for the following graph are



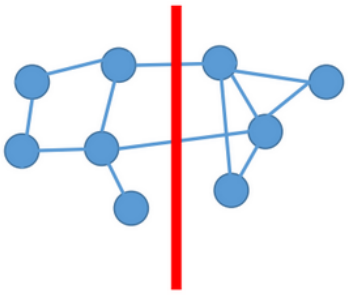
$$L = \begin{bmatrix} 1 & -1 & & \\ -1 & 1 & & \\ & & 2 & -2 \\ & & -2 & 2 \end{bmatrix}, \quad \text{and} \quad \Lambda = \{0, 0, 2, 4\}.$$

A final fact compares the graph Laplacian with the finite difference Laplacian. Suppose the graph is a 2D grid (e.g., representing an image) and we use weights $w_{ij} = 1$. Then, the unnormalized graph Laplacian L is (a scalar multiple of) the usual 2D finite difference Laplacian.



We will now explore how graph Laplacians are used for clustering data. Consider the problem of finding minimally weighted **cuts** that divide the graph into parts. This generally leads to NP-hard problems, so we "relax" the problem, yielding our spectral clustering algorithms.

The problem statement is as follows. Given a graph G with the weight matrix W, find a partition of G such that the edges between the partitions have very low weight. See the figure below for an example of what we want with $k = 2$ subsets.



e. g. , $k = 2$ subsets

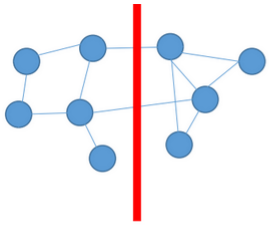
17.3 Clustering using Graph Laplacians

One approach is called **MinCut**, which finds a partition $A_1, \ldots, A_k$ that minimizes

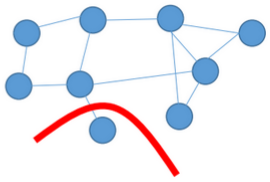
$$\text{cut}(A_1, \ldots, A_k) = \frac{1}{2} \sum_{i=1}^{k} W(A_i, \overline{A_i}).$$

The notation $\overline{A_i}$ denotes the complement of A_i (i.e., vertices not in A_i).

The basic MinCut is fairly easy to solve, but it does not give **useful** results. The minimal solution often separates out individual vertices, rather than finding large subsets of nodes with low weight between them. For example, we would prefer the graph cut on the left below, but MinCut will normally compute the right cut.



Preferred cut



Computed cut

Better cuts in the graph would encourage the size of partitions to be larger, or more "balanced". Therefore, we should divide weights by the **size** of the subset ($|A_i|$ or $\text{vol}(A_i)$). This motivates the following definitions.

Definition 17.6. *The RatioCut and Ncut (aka normalized cut) minimize the following, respectively:*

$$\text{RatioCut}(A_1, \ldots, A_k) = \frac{1}{2} \sum_{i=1}^{k} \frac{W(A_i, \overline{A_i})}{|A_i|} = \sum_{i=1}^{k} \frac{\text{cut}(A_i, \overline{A_i})}{|A_i|},$$

and

$$\text{Ncut}(A_1, \ldots, A_k) = \frac{1}{2} \sum_{i=1}^{k} \frac{W(A_i, \overline{A_i})}{vol(A_i)} = \sum_{i=1}^{k} \frac{\text{cut}(A_i, \overline{A_i})}{vol(A_i)}.$$

Recall that $|A_i|$ is the number of vertices in the set and vol(A_i) is the sum of degrees of vertices in the set.

17.3.1 Relaxation of RatioCut via Graph Laplacian

Sadly, minimizing for the RatioCut and Ncut is NP-hard. However, we can relax the minimization problem once we rewrite it in terms of the graph Laplacian. First, we consider the case of partitioning into 2 subsets, A and $\overline{A}$, subject to

$$\min_A \text{RatioCut}(A, \overline{A}).$$

We can rewrite this in terms of the graph Laplacian as follows.

Given a subset $A \subset V$, define $x = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}^T$, where

$$x_i = \begin{cases} +\sqrt{\frac{|\overline{A}|}{|A|}} \text{ if } v_i \in A, \\ -\sqrt{\frac{|A|}{|\overline{A}|}} \text{ if } v_i \in \overline{A}. \end{cases}$$

We can show the following three results to rewrite the minimization problem:

1. $x^T L x = |V| \cdot \text{RatioCut}(A, \overline{A})$,
2. $\sum_{i=1}^{n} x_i = 0$, i.e., $x^T \mathbf{1} = 0$,
3. $\|x\|^2 = n$.

Proof. 1. We plug the x_i expressions into the following

$$\begin{aligned} x^T L x &= \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} (x_i - x_j)^2, \\ &= \frac{1}{2} \sum_{i \in A} \sum_{j \in \overline{A}} w_{ij} \left(\sqrt{\frac{|\overline{A}|}{|A|}} + \sqrt{\frac{|A|}{|\overline{A}|}} \right)^2 + \frac{1}{2} \sum_{i \in \overline{A}} \sum_{j \in A} w_{ij} \left(-\sqrt{\frac{|\overline{A}|}{|A|}} - \sqrt{\frac{|A|}{|\overline{A}|}} \right)^2, \\ &= \frac{1}{2} \sum_{i=1}^{k} W(A_i, \overline{A}_i) \left(\frac{|\overline{A}|}{|A|} + \frac{|A|}{|\overline{A}|} + 2\cancelto{1}{\sqrt{\frac{|\overline{A}|}{|A|} \frac{|A|}{|\overline{A}|}}} \right), \\ &= \frac{1}{2} \sum_{i=1}^{k} W(A_i, \overline{A}_i) \left(\frac{|A| + |\overline{A}|}{|A|} + \frac{|A| + |\overline{A}|}{|\overline{A}|} \right), \\ &= |V| \cdot \text{RatioCut}(A, \overline{A}), \quad \text{using } |A| + |\overline{A}| = |V|. \end{aligned}$$

2.

$$\sum_{i=1}^{n} x_i = \sum_{i \in A} \sqrt{\frac{|\overline{A}|}{|A|}} - \sum_{i \in \overline{A}} \sqrt{\frac{|A|}{|\overline{A}|}} = |A| \sqrt{\frac{|\overline{A}|}{|A|}} - |\overline{A}| \sqrt{\frac{|A|}{|\overline{A}|}} = \sqrt{|A||\overline{A}|} - \sqrt{|A||\overline{A}|} = 0,$$

3.

$$\|x\|^2 = \sum_{i=1}^{n} x_i^2 = |A|\frac{|\overline{A}|}{|A|} + |\overline{A}|\frac{|A|}{|\overline{A}|} = |A| + |\overline{A}| = n.$$

□

So the minimization problem becomes

$$\min_{A \subset V} x^T L x,$$
$$\text{subject to } x \perp \mathbf{1} \text{ and } \|x\| = \sqrt{n}.$$

But this minimization problem is still discrete and NP-hard! The vertices are strictly only in A or $\overline{A}$. However, we can **relax** the problem by allowing x to consist of arbitrary **real** numbers

$$\min_{x \in \mathbb{R}^n} x^T L x,$$
$$\text{subject to } x \perp \mathbf{1} \text{ and } \|x\| = \sqrt{n}.$$

The solution to this relaxed minimization problem turns out to be the eigenvector, x, of L corresponding to the 2^{nd} smallest eigenvalue (aka "Fielder vector").

We can recover the separation into 2 clusters by thresholding x

$$v_i \in A \text{ if } x_i \geq 0,$$

$$v_i \in \overline{A} \text{ if } x_i < 0,$$

where x_i are the components of the Fielder vector.

17.3.2 Relaxation of Ncut via Graph Laplacian

Applying the same idea for Ncut, but with a different measure of set size, we start from

$$\min_{A} \text{Ncut}(A, \overline{A}).$$

We can rewrite this using the normalized graph Laplacian. Given a subset $A \subset V$, define $x = \{x_1, \ldots, x_n\}$ where

$$x_i = \begin{cases} +\sqrt{\frac{\text{vol}(\overline{A})}{\text{vol}(A)}} & \text{if } v_i \in A, \\ -\sqrt{\frac{\text{vol}(A)}{\text{vol}(\overline{A})}} & \text{if } v_i \in \overline{A}. \end{cases}$$

We can then show that

1. $x^T L x = \text{vol}(V) \cdot \text{Ncut}(A, \overline{A})$,
2. $\sum_{i=1}^{n} d_i x_i = 0$, i.e., $(Dx)^T \mathbf{1} = 0$,

3. $x^T Dx = \text{vol}(V)$.

So the minimization problem becomes

$$\min_{A \subset V} x^T Lx, \text{ subject to } Dx \perp \mathbf{1} \text{ and } x^T Dx = \text{vol}(V).$$

Again, this is still discrete and NP-hard to minimize. However, when we relax the minimization problem we instead solve

$$\min_{x \in \mathbb{R}^n} x^T Lx, \text{ subject to } Dx \perp \mathbf{1} \text{ and } x^T Dx = \text{vol}(V).$$

Defining $y = D^{\frac{1}{2}} x$, the relaxed problem becomes

$$\min_{y \in \mathbb{R}^n} y^T \hat{L} y, \text{ subject to } y \perp D^{\frac{1}{2}} \mathbf{1} \text{ and } \|y\|^2 = \text{vol}(V).$$

The solution again becomes the Fielder vector, but for $\hat{L}$ instead. So we threshold y_i at zero in order to determine the two clusters.

Notation:

- $|V|$ is un-normed.
- $\text{vol}(V)$ is normed, using the weights.

17.4 K-means Clustering

The above works for 2 clusters, but what about $k > 2$ clusters? For more clusters we can not simply threshold to zero, since we have more than 2 groups. Instead, we will make use of **k-means clustering** on data drawn from several eigenvectors. First, let us consider the basic k-means algorithm.

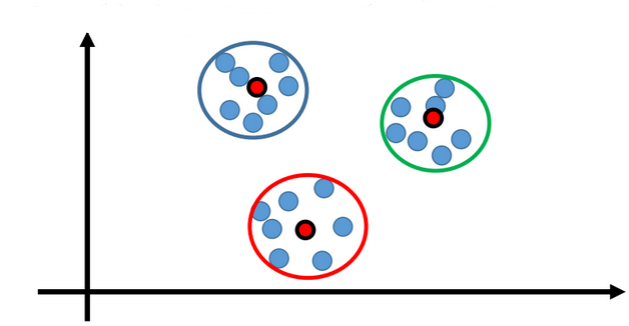
Given a set of n data points/vectors $\{ p_j \}$, find the partition of the points $A_1, \ldots, A_k$ such that each point is assigned to the set whose mean μ_i is closest to it. K-means aims to solve the problem

$$\min_{A_i} \sum_{i=1}^{k} \sum_{p \in A_i} \|p - \mu_i\|^2.$$

There are two factors at play in this minimization problem:

- assignment of points to sets,
- distance of each point to the mean of its set.

Consider the example in the figure below. Given blue (2D) data points try to find $k = 3$ (how to choose k: later) means **and** an assignment of particles to the corresponding 3 clusters.

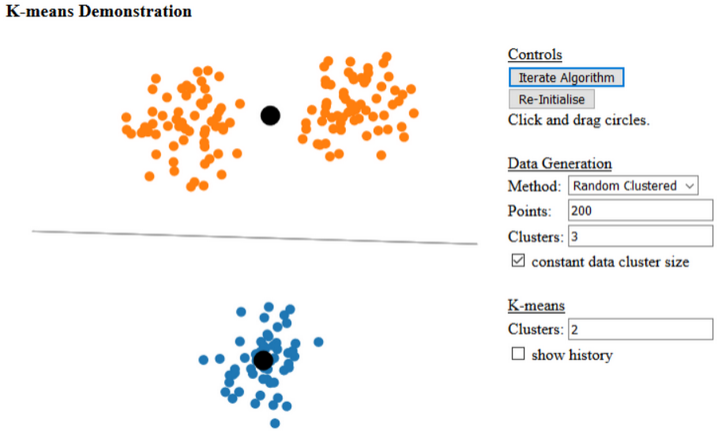


We would expect something like the red points as the means of the 3 clusters. The data points would belong to the cluster whose mean is closest to them (as shown with blue, green, and red encompassing circles).

The **k-means algorithm** performs the following steps:

1. Start with some initial guesses for the k means $\{\mu_i\}$,
2. Assign each point p to the cluster A_i if p is closer to μ_i than any of the other k means,
3. Re-compute new means $\{\mu_i\}$ for all partitions $\{A_i\}$,
4. Repeat.

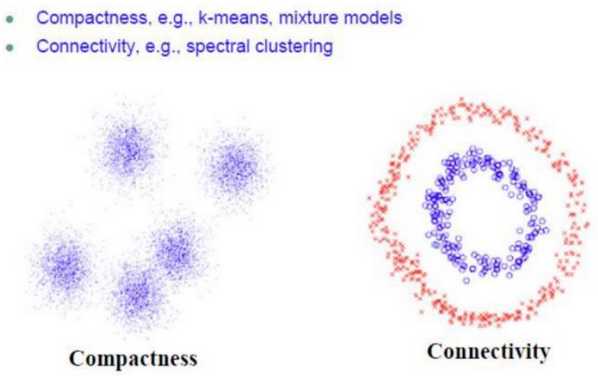
There is a nice interactive demo of k-means available online. Please visit http://alekseynp.com/viz/k-means.html to try it out!



17.5 Spectral Clustering: Cuts and K-means Together

If k-means can do clustering why do we not just apply it to our problem? **Spectral clustering** allows for:

- More general weights/measures of similarity (i.e., not just Euclidean distance),
- Non-convex clusters.



We can apply k-means for the $k = 2$ case instead of thresholding at zero, to assign points into the two clusters. Specifically, considering the entries of the eigenvector $\{x_i\}$ as n data points in $\mathbb{R}$, then apply k-means with $k = 2$. The advantage of this is that it can be extended to $k > 2$ clusters.

Intuitively, spectral clustering consists of the following (also depicted in Figure 17.30):

1. Interpret our input data as a graph and choose weights W to indicate our notions of similarity,
2. Use the eigenvectors of the graph Laplacian to convert vertices into data points in $\mathbb{R}^k$,
3. Apply k-means to cluster the points in Euclidean space ($\mathbb{R}^k$).

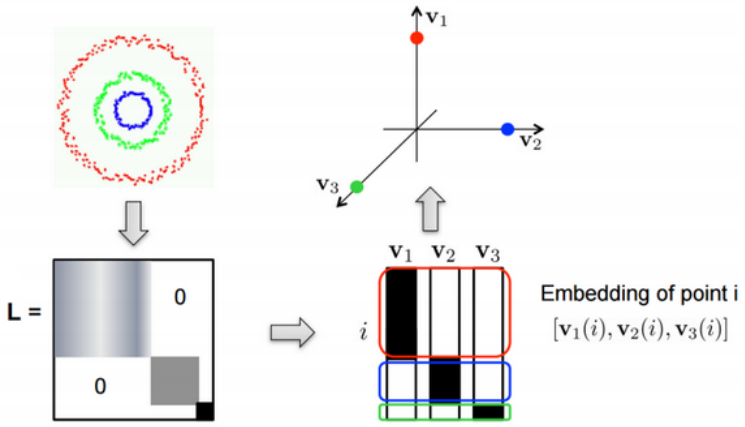


Figure 17.30: Steps of spectral clustering.

In more detail, the **unnormalized spectral clustering** algorithm is:

1. Construct the unnormalized graph Laplacian L,
2. Compute the first k eigenvectors $q_1, \ldots, q_k$ of L (corresponding to **smallest** magnitude eigenvalues),
3. Consider $Q_k = [q_1, \ldots, q_k]$. Let $p_i \in \mathbb{R}^k$ be the vector given by row i of Q_k (i.e., $p_i = Q_k(i, :)$ in Matlab notation),
4. For the resulting n points $\{p_i\}$ in $\mathbb{R}^k$, apply k-means to cluster them into k groups $\{A_1, \ldots, A_k\}$.

The **normalized** version of the **spectral clustering algorithm** requires two changes.

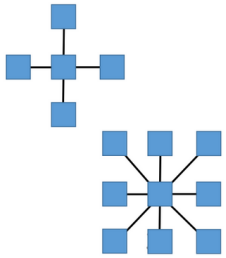
1. First, we use $\hat{L}$ instead of L.
2. Second, instead of p_i, we use normalized rows for points, $\overline{p}_i = \frac{p_i}{\|p_i\|}$.

17.5.1 Choosing Weights W

The choice of the weight matrix W is meant to measure similarity between vertices of the graph. This means W is **problem dependent**. Usually, we want non-zero weights only between a small set of local graph neighbours, (e.g., within graph distance 1 or 2). The more neighbours we include creates more non-zero entries in W. If we include fewer neighbours we create a sparser graph Laplacian (having a lower cost of eigendecomposition).

For an image segmentation task we view pixels as graph vertices. We connect adjacent or nearby pixels with graph edges which form our graph. For example,

- including the 4 adjacent pixels gives W with non-zero structure similar to usual finite difference Laplacian,
- including the 4 diagonal neighbours also would give 8 neighbours, so W has at most 8 non-zeros per row.



We will set w_{ij} to measure similarity between pixels i and j using two factors:

1. Euclidean distance between pixels i and j,
2. intensity difference between pixels i and j.

For $i \neq j$ we will use

$$w_{ij} = \left(e^{-\frac{\|x_i - x_j\|^2}{\sigma_{dist}^2}} \right) \left(e^{-\frac{\|I_i - I_j\|^2}{\sigma_{int}^2}} \right),$$

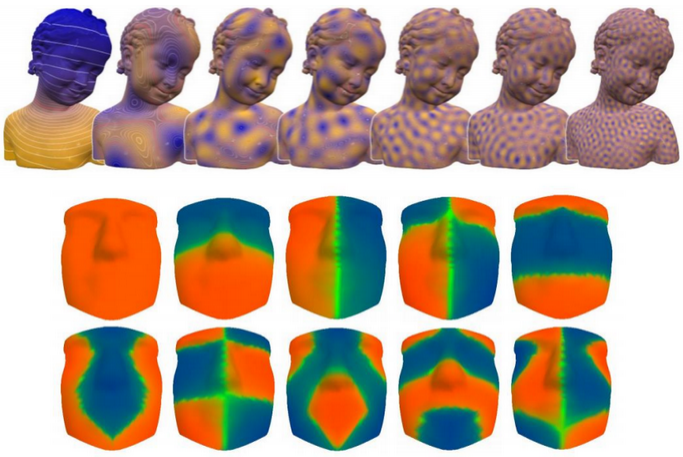
where pixel i is at position x_i with intensity I_i and likewise for pixel j. We define positions as $x_i = (r, c)$ if pixel i is at row r, column c. The parameters σ_{dist}^2 and σ_{int}^2 can be varied to adjust the relative importance of the terms.

17.6 Other Applications

Spectral approaches find many other applications in the field of graphics processing.

17.6.1 Geometric Mesh Processing

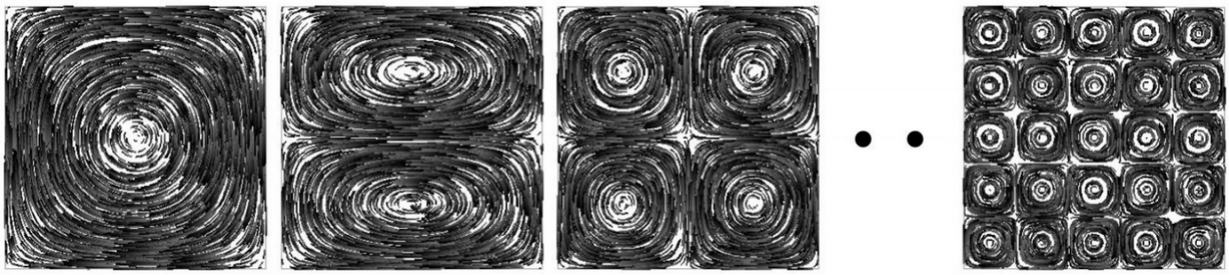
The following are visualizations of several eigenvectors associated to Laplacians defined on 3D triangle meshes.



source: https://www.cs.sfu.ca/~haoz/pubs/zhang_eg07star_spectral.pdf.

17.6.2 Motion Analysis

Extracting dominant motion "modes" in solids or fluids for analysis and efficient simulation.



⟶ increasing eigenvalue magnitude

The above images show the basis of divergent-free fields that are eigenfunctions of the vector Laplacian. Basis fields have correspondence with spatial scales of vorticity. Their coefficients form a discrete spectrum. Another example of motion analysis involves vibrating membranes, see https://en.wikipedia.org/wiki/Vibrations_of_a_circular_membrane.

To learn more, check out "A Tutorial on Spectral Clustering", by Ulrike von Luxburg, http://www.tml.cs.uni-tuebingen.de/team/luxburg/publications/Luxburg07_tutorial.pdf.

18 Lecture 18: Introduction to Singular Value Decompositions

Outline

1. Geometric Motivation: $AV = U\Sigma$
   (a) Matrix Form
   (b) Comparison with Eigendecomposition
2. Properties of the SVD
3. Computing the SVD - 1st Attempt
   (a) Example

This lecture introduces the final decomposition called the **singular value decomposition**. Lecture 4 of Trefethen & Bau provides more detail, see https://people.maths.ox.ac.uk/trefethen/text.html.

18.1 Geometric Motivation: $AV = U\Sigma$

The **image** of the unit hypersphere S in $\mathbb{R}^n$ under any $m \times n$ matrix transformation A is a **hyperellipse** in $\mathbb{R}^m$. Figure 18.31 shows the geometric interpretation of this transformation. Both the hypersphere and hyperellipse are in $\mathbb{R}^2$ in this example. However, the dimensions can be any n and m, not necessarily $n = m$.

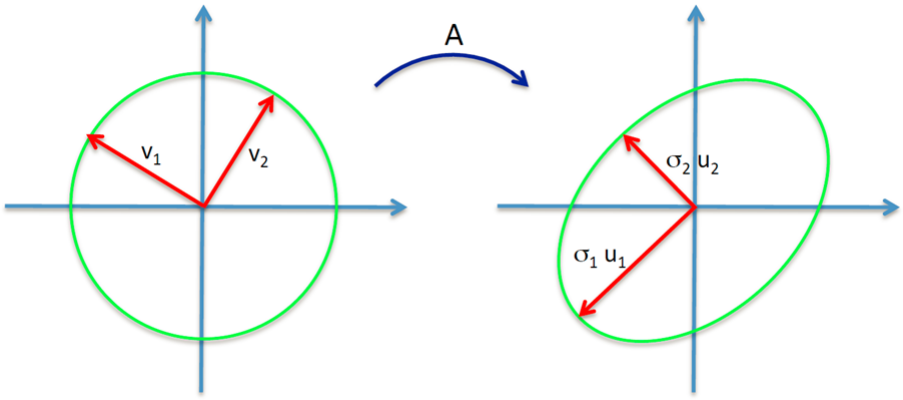


Figure 18.31: Transformation of unit hypersphere S (left) by matrix A into hyperellipse AS (right).

The factors by which the hypersphere is scaled in each of the principal semi-axes of the hyperellipse are called the **singular values** of A. The n singular values are denoted σ_i. By convention we will order them such that

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0.$$

Notice that all the singular values are non-negative.

The n **left singular vectors**, u_i, of A are the unit vectors in the directions of the principal semi-axes of the ellipse. The n **right singular vectors**, v_i, are the unit vectors in S such that

$$Av_j = \sigma_j u_j.$$

In other words, v_i's are the **pre-image** of u_i's under the transformation A.

18.1.1 Matrix Form

We can write the above equation

$$Av_j = \sigma_j u_j, \qquad \text{for } j = 1, 2, \ldots, n,$$

in matrix form to define the **reduced SVD**. Pictorially, we have

$$\underbrace{\begin{bmatrix} \\ A \\ \\ \end{bmatrix}}_{A,\ m\times n} \underbrace{\begin{bmatrix} v_1 & v_2 & \cdots & v_n \\ & & & \end{bmatrix}}_{V,\ n\times n}$$

$$= \underbrace{\begin{bmatrix} & & & \\ u_1 & u_2 & \cdots & u_n \\ & & & \end{bmatrix}}_{\hat{U},\ m\times n} \underbrace{\begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{bmatrix}}_{\hat{\Sigma},\ n\times n}$$

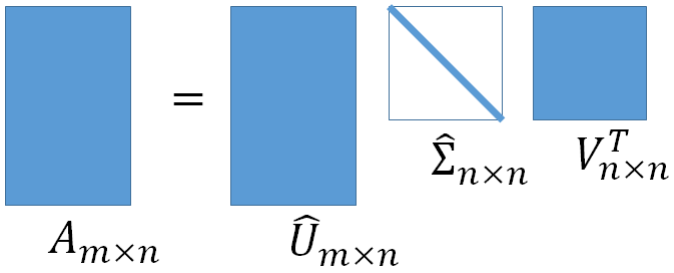
The matrix $\hat{\Sigma}$ is a diagonal matrix, with the singular values of A on its diagonal. The matrices $\hat{U}$ and V have orthonormal columns. Note that the hat notation indicates **reduced** or **economy-sized** SVD

$$AV = \hat{U}\hat{\Sigma}.$$

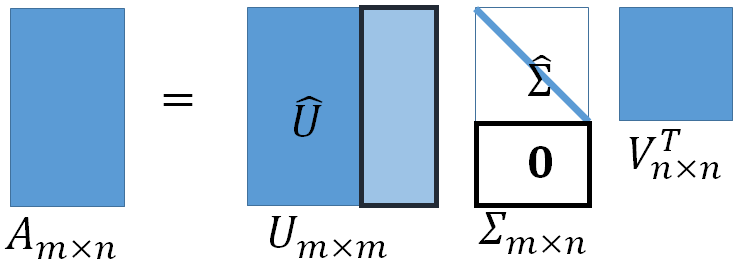
Since V is orthogonal, if we multiply by V^T on the right, we can equivalently write it as

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

The figure below shows this reduced SVD of A pictorially.



The **full SVD** is constructed in a similar way to how the full QR factorization was created from the reduced QR factorization. We can define a full SVD by adding $m - n$ more orthonormal columns to $\hat{U}$ to give a square, **orthogonal** U. Then we must also add extra empty rows to $\hat{\Sigma}$ to construct Σ. That is, replace $\hat{U} \to U$ and $\hat{\Sigma} \to \Sigma$ as shown in the figure below.



- Every matrix $A \in \mathbb{R}^{m\times n}$ has a singular value decomposition.
  - We will prove this in the next lecture.
  - Furthermore, the singular values are uniquely determined.
- Also, if A is square and σ_j are distinct, then the left and right singular vectors are unique **(up to signs)**.

18.1.2 Comparison with Eigendecomposition

The SVD is similar to the eigendecomposition we have seen previously. Consider the SVD vs the eigendecomposition

$$A = U\Sigma V^T \quad \text{vs} \quad A = X\Lambda X^{-1}.$$

- Both decompositions act to diagonalize a matrix.
- The SVD uses two bases: U and V, the left and right singular vectors.
- The eigendecomposition uses only one basis, the set of eigenvectors.
- The SVD always uses orthonormal vectors.
- The eigenvectors are not orthonormal **in general** (though for the real symmetric matrices we considered, they are).
- Finally, not all matrices have an eigendecomposition, but **all matrices** have an SVD, even rectangular matrices.

18.2 Properties of the SVD

Next we will discuss some properties of the SVD. For the following theorems let $A \in \mathbb{R}^{m\times n}$ and $r = \#$ of non-zero singular values.

Theorem 18.1.

$$\text{rank}(A) = r.$$

Proof. Rank of a diagonal matrix is the number of non-zero diagonal entries. U and V are both of full rank, by definition. Hence $\text{rank}(A) = \text{rank}(\Sigma) = r$. □

Theorem 18.2.

$$\begin{aligned} \text{range}(A) &= \text{span}\{u_1, u_2, \ldots, u_r\}, \\ \text{null}(A) &= \text{span}\{v_{r+1}, \ldots, v_n\}. \end{aligned}$$

Proof. We will not give a full proof of this theorem. Instead for the second property, we show that a vector in $\text{span}\{v_{r+1}, \ldots, v_n\}$ is in $\text{null}(A)$ (in other words, $\text{span}\{v_{r+1}, \ldots, v_n\} \subseteq \text{null}(A)$).

Let $x \in \text{span}\{v_{r+1}, \ldots, v_n\}$ be arbitrary. Then

$$x = \sum_{i=r+1}^{n} w_i v_i \qquad \text{and so} \qquad Ax = \sum_{i=r+1}^{n} w_i (Av_i).$$

Observe that

$$\begin{aligned} Av_i &= U\Sigma V^T v_i \\ &= U\Sigma e_i, \end{aligned}$$

with the last equality holding because V is an orthogonal matrix. But $\Sigma e_i = 0$ for $i \in [r+1, n]$ since the corresponding entries of Σ are zero. Therefore $Ax = 0$, so $x \in \text{null}(A)$. □

I claim that

$$\|A\|_2^2 = \lambda_{\max}\left(A^T A\right).$$

Proof.

- Recall that

$$\begin{aligned} \|A\|_2^2 &= \max_{\|x\|_2=1} \|Ax\|_2^2 \\ &= \max_{\lambda \in \Lambda(A)} |\lambda|^2 \\ &= \max_{\lambda \in \Lambda(A)} \lambda^2. \end{aligned}$$

- The SVD of A gives

$$\begin{aligned} A &= && U\Sigma V^T, \text{ so that} \\ A^T A &= && \left(U\Sigma V^T\right)^T U\Sigma V^T \\ &= && V\Sigma^T \underbrace{U^T U}_{=I} \Sigma V^T \\ &\underbrace{=}_{\Sigma \text{ is diagonal, thus symmetric}} && V\Sigma^2 V^T. \end{aligned}$$

- This is a similarity transformation, hence the eigenvalues of $A^T A$ equal the eigenvalues of Σ^2.
- But Σ is a diagonal matrix with A's singular values on its diagonal.
- Hence the eigenvalues of $A^T A$ equal the squares of the singular values of A.

□

Lemma 18.2.1. *Keeping all of the above notation, $A^T \hat{U}_{m-n} = 0$.*

Proof.

$$\begin{aligned}
A &= \hat{U}\hat{\Sigma}V^T, \text{ so that} \\
A^T &= \left(\hat{U}\hat{\Sigma}V^T\right)^T \\
&= V\hat{\Sigma}^T\hat{U}^T \\
&\underbrace{=}_{\Sigma \text{ is diagonal}} V\hat{\Sigma}\hat{U}^T, \text{ so we can compute} \\
A^T\hat{U}_{m-n} &= \left(V\hat{\Sigma}\hat{U}^T\right)\hat{U}_{m-n} \\
&= V\hat{\Sigma}\underbrace{\left(\hat{U}^T\hat{U}_{m-n}\right)}_{=0} \\
&= 0,
\end{aligned}$$

where $\hat{U}^T\hat{U}_{m-n} = 0$ holds because $\hat{U}_{m-n}$'s columns are orthogonal to $\hat{U}$'s columns. □

Notation:

$$\|A\|_F^2 = \sum_{i,j} a_{ij}^2 = \operatorname{tr}\left(A^TA\right), \text{ where } \|A\|_F \text{ is the } \textbf{Frobenius norm}.$$

Recall that

$$\operatorname{tr}(A) = \sum_{i=1}^{n} a_{ii},$$

the sum of the diagonal entries of A.

Theorem 18.3. *$\|A\|_2 = \sigma_1$ and $\|A\|_F = \sqrt{\sigma_1^2 + \cdots + \sigma_r^2}$.*

Proof. We have $\lambda_{\max}(A^TA) = \lambda_{\max}(\Sigma^2) = \sigma_1^2 \Rightarrow \|A\|_2 = \sigma_1$.

Now for the Frobenius norm we have

$$\begin{aligned}
&\|A\|_F^2 \\
= \quad &\operatorname{tr}(A^TA) \\
\underbrace{=}_{A^TA = V\Sigma^2V^T} \quad &\operatorname{tr}\left(V\Sigma^2V^T\right) \\
= \quad &\operatorname{tr}\left((V\Sigma)(V\Sigma)^T\right), \\
= \quad &\operatorname{tr}\left((V\Sigma)^T(V\Sigma)\right), \text{ trace identity } \operatorname{tr}(X^TY) = \operatorname{tr}(XY^T), \\
= \quad &\operatorname{tr}\left(\Sigma V^TV\Sigma\right), \\
= \quad &\operatorname{tr}(\Sigma^2), \text{ by the orthogonality of } V, \\
= \quad &\sigma_1^2 + \cdots + \sigma_r^2.
\end{aligned}$$

□

Theorem 18.4. *Non-zero singular values of A are the square roots of non-zero eigenvalues of AA^T or A^TA.*

Proof. A^TA and AA^T are similar to Σ^2. We proved this for A^TA above; the proof for AA^T is similar.

1. We showed above that $A^TA = V\Sigma^2V^T$.
2. Similarly,

$$\begin{aligned} AA^T &= U\Sigma V^T \left(U\Sigma V^T\right)^T \\ &= U\Sigma \underbrace{\left(V^TV\right)}_{I}\Sigma^T U^T \\ &= U\Sigma^2U^T, \text{ since } \Sigma \text{ is diagonal.} \end{aligned}$$

□

Recall Notation: $\Lambda(A)$ is the set of eigenvalues of A.

New Notation: $\sigma(A)$ is the set of the singular values of A.

Theorem 18.5. *If $A = A^T$, then $\sigma(A) = \{|\lambda| : \lambda \in \Lambda(A)\}$. In particular, if A is SPD then $\sigma(A) = \Lambda(A)$.*

Proof. Real symmetric matrices have orthogonal eigenvectors and real eigenvalues, so

$$A = Q\Lambda Q^T, \quad \text{with } Q \text{ orthogonal.}$$

Construct the SVD as

$$A = \underbrace{Q}_{U}\underbrace{|\Lambda|}_{\Sigma}\underbrace{\text{sign}(\Lambda)Q^T}_{V^T},$$

where $|\Lambda|$ and $\text{sign}(\Lambda)$ are diagonal matrices with entries $|\lambda_j|$ and $\text{sign}(\lambda_j)$, respectively. If desired one can also insert orthogonal permutation matrices to sort the σ's. □

Theorem 18.6. *The **condition number** for $A \in \mathbb{R}^{n\times n}$ is $\kappa_2(A) = \frac{\sigma_1}{\sigma_n}$.*

Proof. By the definition of κ and by Theorem 18.3, we have

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \sigma_1\|A^{-1}\|_2.$$

Since $A = U\Sigma V^T$, therefore $A^{-1} = V\Sigma^{-1}U^T$ is the SVD of A^{-1}. Therefore

$$\begin{aligned} \|A^{-1}\|_2 &= \frac{1}{\sigma_n} \\ \Rightarrow \kappa_2(A) &= \frac{\sigma_1}{\sigma_n}. \end{aligned}$$

□

18.3 Computing the SVD - 1st Attempt

We first consider a naïve approach to computing the SVD. Since $A = U\Sigma V^T$ we showed above that $A^TA = V\Sigma^2V^T$, which is an eigendecomposition of A^TA! Therefore, the eigen**values** of A^TA are **squares** of the singular values of A. The eigen**vectors** of A^TA are the **right singular** vectors of A.

This suggests a (naïve) method for computing the SVD:

1. Form A^TA (it's symmetric and positive semi-definite, so its eigenvalues are real and non-negative),
2. Compute eigendecomposition of $A^TA = V\Lambda V^T$,
3. Compute $\Sigma = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{bmatrix}$, where $\sigma_i = \sqrt{\lambda_i}$ and $\Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}$,
4. Solve $U\Sigma = AV$ for orthogonal U (e.g., by QR factorization).

Recovering U from the above algorithm involves (note, we already have Σ, A, V):

- Multiply AV to get A',
- QR factor $A' = QR$,
- Identify $U = Q, \Sigma = R$.

This ensures that $U = Q$ is properly orthogonal. Conveniently, $R = \Sigma$ will be diagonal.

Unfortunately, this naïve method is inaccurate; the error satisfies

$$|\tilde{\sigma}_k - \sigma_k| = O\left(\frac{\epsilon \|A\|^2}{\sigma_k}\right),$$

which can be very bad for small singular values! (**Conceptually**, this is similar to how solving least squares by normal equations used A^TA. Effectively this "squares the condition number", therefore making it less accurate than QR factorization). In the next lecture we will discuss a better alternative for computing the SVD.

18.3.1 Example

We can find the SVD of $A = \begin{bmatrix} 0 & -\frac{1}{2} \\ 3 & 0 \\ 0 & 0 \end{bmatrix}$ in a few different ways.

1. **Method 1:**

$$\begin{aligned} A^TA &= \begin{bmatrix} 0 & 3 & 0 \\ -\frac{1}{2} & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & -\frac{1}{2} \\ 3 & 0 \\ 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 9 & 0 \\ 0 & \frac{1}{4} \end{bmatrix} \\ &= V\Sigma^2V^T \\ &= Q\Lambda Q^T \end{aligned}$$

Therefore $\lambda_1 = 9, \lambda_2 = \frac{1}{4}, v_1 = \begin{bmatrix}1\\0\end{bmatrix}, v_2 = \begin{bmatrix}0\\1\end{bmatrix}$ since $Q = I$. Therefore $\sigma_1 = 3, \sigma_2 = \frac{1}{2}$, so

$$\hat{\Sigma} = \begin{bmatrix}3 & 0\\ 0 & \frac{1}{2}\end{bmatrix} \quad \text{and} \quad V = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}.$$

Then find U from $U\Sigma = AV$

$$\begin{bmatrix}u_1 & u_2\end{bmatrix}\begin{bmatrix}3 & 0\\ 0 & \frac{1}{2}\end{bmatrix} = \begin{bmatrix}0 & -\frac{1}{2}\\ 3 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}0 & -\frac{1}{2}\\ 3 & 0\\ 0 & 0\end{bmatrix}$$

Hence

$$\begin{aligned} 3u_1 &= \begin{bmatrix}0\\3\\0\end{bmatrix} \text{ therefore } u_1 = \begin{bmatrix}0\\1\\0\end{bmatrix} \\ \frac{1}{2}u_2 &= \begin{bmatrix}-\frac{1}{2}\\0\\0\end{bmatrix} \text{ therefore } u_2 = \begin{bmatrix}-1\\0\\0\end{bmatrix} \end{aligned}$$

Thus $\hat{U} = \begin{bmatrix}0 & -1\\ 1 & 0\\ 0 & 0\end{bmatrix}$.

2. **Method 2:** Use AA^T instead, same idea.
3. **Method 3:** Let's exploit intuition about SVD and the simple structure of this matrix. By inspection, $\text{range}(A) = \text{span}\{u_1, u_2\}$ for $u_1 = \begin{bmatrix}0\\1\\0\end{bmatrix}$ and $u_2 = \begin{bmatrix}1\\0\\0\end{bmatrix}$, u_1 and u_2 are orthonormal. The lengths of the principal axes are 3 and $\frac{1}{2}$.
Then by the definition of SVD

$$\begin{aligned} Av_1 &= \sigma_1 u_1 \\ \begin{bmatrix}0 & -\frac{1}{2}\\ 3 & 0\\ 0 & 0\end{bmatrix} v_1 &= 3\begin{bmatrix}0\\1\\0\end{bmatrix} \\ \Rightarrow v_1 &= \begin{bmatrix}1\\0\end{bmatrix} \\ Av_2 &= \sigma_2 u_2 \\ \begin{bmatrix}0 & -\frac{1}{2}\\ 3 & 0\\ 0 & 0\end{bmatrix} v_2 &= \frac{1}{2}\begin{bmatrix}1\\0\\0\end{bmatrix} \\ \Rightarrow v_2 &= \begin{bmatrix}0\\-1\end{bmatrix} \end{aligned}$$

The details of solving both systems follow.

$$
\begin{array}{rcl}
 & & \left[\begin{array}{cc|c} 0 & -\frac{1}{2} & 0 \\ 3 & 0 & 3 \\ 0 & 0 & 0 \end{array}\right] \\
 & \sim & \left[\begin{array}{cc|c} 3 & 0 & 3 \\ 0 & -\frac{1}{2} & 0 \\ 0 & 0 & 0 \end{array}\right] \begin{array}{l} R_1 \leftarrow R_2 \\ R_2 \leftarrow R_1 \end{array} \\
 & \sim & \left[\begin{array}{cc|c} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \begin{array}{l} R_1 \leftarrow \frac{1}{3} R_1 \\ R_2 \leftarrow -2R_2 \end{array} \\
\Rightarrow v_1 & = & \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\
 & & \left[\begin{array}{cc|c} 0 & -\frac{1}{2} & \frac{1}{2} \\ 3 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \\
 & \sim & \left[\begin{array}{cc|c} 3 & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 \end{array}\right] \begin{array}{l} R_1 \leftarrow R_2 \\ R_2 \leftarrow R_1 \end{array} \\
 & \sim & \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{array}\right] \begin{array}{l} R_1 \leftarrow \frac{1}{3} R_1 \\ R_2 \leftarrow -2R_2 \end{array} \\
\Rightarrow v_2 & = & \begin{bmatrix} 0 \\ -1 \end{bmatrix}
\end{array}
$$

So $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ i.e. same solution, up to signs in U and V.

19 Lecture 19: Singular Value Decompositions Versus Eigendecomposition

Outline

1. Alternative Formulation
   (a) Alternate Approach Example
2. Proof of Existence of SVD
3. Stability Comparison
4. Golub-Kahan Bidiagonalization

Recall that SVD is the decomposition of any matrix A into $U\Sigma V^T$, where Σ is diagonal with non-negative entries, and U, V are orthogonal. In the previous lecture we seen that the SVD can be found from the eigendecomposition of A^TA or AA^T.

In this lecture we will see a more stable method using the eigendecomposition of

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

We will also prove the existence of the SVD, discuss stability, and discuss how to compute the SVD efficiently.

19.1 Alternative Formulation

Assume A is square, i.e. $A \in \mathbb{R}^{n\times n}$. Consider the $2n \times 2n$ symmetric matrix

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

By computing the eigendecomposition of $H = Q\Lambda Q^T$ we can extract the singular values and vectors. We have that $\sigma_A = |\lambda_H|$, and U, V can be recovered from the eigenvectors. Let us see why all of this holds.

Write $A = U\Sigma V^T$, then we have $AV = U\Sigma$ because V is orthogonal. Likewise,

$$\begin{aligned} A^T &= (U\Sigma V^T)^T \\ &= V\Sigma^T U^T \\ &= V\Sigma U^T, \text{ because } \Sigma \text{ is diagonal.} \end{aligned}$$

Thus $A^TU = V\Sigma$ because U is orthogonal. Hence we have

$$\begin{aligned} \underbrace{\begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}}_{H} \underbrace{\begin{bmatrix} V & V \\ U & -U \end{bmatrix}}_{Q} &= \begin{bmatrix} A^TU & -A^TU \\ AV & AV \end{bmatrix} \\ &= \begin{bmatrix} V\Sigma & -V\Sigma \\ U\Sigma & U\Sigma \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} V & V \\ U & -U \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{bmatrix}}_{\Lambda} \end{aligned}$$

Therefore, $HQ = Q\Lambda$, equivalently $H = Q\Lambda Q^T$, gives an **eigendecomposition** of H. Note, we need to normalize the columns of Q, to make Q an orthogonal matrix.

Explanation Of Why Q's Columns Are Orthogonal, Given U, V Are Orthogonal Matrices

- Let $1 \leq i < j \leq n$ be arbitrary.
- Then we have

$$\begin{aligned} & \begin{bmatrix} v_i \\ u_i \end{bmatrix}^T \begin{bmatrix} v_j \\ \pm u_j \end{bmatrix} \\ = \quad & \begin{bmatrix} v_i^T & u_i^T \end{bmatrix} \begin{bmatrix} v_j \\ \pm u_j \end{bmatrix} \\ = \quad & v_i^T v_j \pm u_i^T u_j \\ \underbrace{=}_{U,V \text{ are orthogonal}} \quad & 0 \pm 0 \\ = \quad & 0. \end{aligned}$$

To summarize we have the following steps:

1. Form $H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$,
2. Compute eigendecomposition $HQ = Q\Lambda$,
3. Set $\sigma_A = |\lambda_H|$,
4. Extract U, V from Q (normalizing for orthogonality).

This algorithm is preferable with respect to stability (see the more detailed section below). The error in the singular values satisfies $|\tilde{\sigma}_k - \sigma_k| = O(\epsilon \|A\|)$, compared to $O(\epsilon \|A\|^2 / \sigma_k)$ for the algorithm using $A^T A$. This approach can be extended to non-square matrices too. Practical algorithms are based on this premise, but without explicitly forming the (large) matrix H.

19.1.1 Alternate Approach Example

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

Therefore

$$H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 3 \\ 0 & 0 & -\frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 0 & 0 \\ 3 & 0 & 0 & 0 \end{bmatrix}$$

MATLAB eigendecomposition gives

$$Q = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix}, \Lambda = \begin{bmatrix} -3 & & & \\ & -\frac{1}{2} & & \\ & & \boxed{\frac{1}{2}} & \\ & & & \boxed{3} \end{bmatrix}$$

Order may be different (of cols) so read off desired cols, for positive Σ entries.

One can verify that the following eigendecomposition (permuting the columns of the previous one, to change the order of the eigenvalues) is also correct, and perfectly fits the shape required of our setup:

$$Q = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & -1 & 0 & 1 \\ 1 & 0 & -1 & 0 \end{bmatrix}, \Lambda = \begin{bmatrix} \boxed{3} & & & \\ & \boxed{\frac{1}{2}} & & \\ & & -3 & \\ & & & -\frac{1}{2} \end{bmatrix}$$

Even better, this modified eigendecomposition fits in perfectly with the remainder of the computation.

Therefore

$$\begin{aligned} \sigma_1 &= 3 \\ v_1 &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ u_1 &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ \sigma_2 &= \frac{1}{2} \\ v_2 &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ u_2 &= \begin{bmatrix} -1 \\ 0 \end{bmatrix} \end{aligned}$$

So

$$\begin{aligned} U &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \\ V &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \Sigma &= \begin{bmatrix} 3 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \end{aligned}$$

Check:

$$\begin{aligned} U\Sigma V^T &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -\frac{1}{2} \\ 3 & 0 \end{bmatrix} \\ &= A \end{aligned}$$

19.2 Proof of Existence of SVD

We claimed in Lecture 18 that every matrix $A \in \mathbb{R}^{m \times n}$ has a singular value decomposition. We will now prove this result.

Proof. Let A be an arbitrary $m \times n$ matrix. The proof is by induction on $n \geq 1$.

Recall that the induced matrix norm is defined as

$$\|A\| := \max_{\|x\|=1} \|Ax\| .$$

Let $\sigma_1 = \|A\|_2$. Let v_1 have $\|v_1\|_2 = 1$ and a direction such that $\|Av_1\|_2 = \|A\|_2 = \sigma_1$. Also, let $u_1 = \frac{Av_1}{\sigma_1}$, so that $Av_1 = \sigma_1 u_1$.

Consider any extensions of vectors u_1 and v_1 to orthonormal bases U_1 and V_1:

$$\begin{aligned} U_1 &= \left[u_1 | \cdots\right], \\ V_1 &= \left[v_1 | \cdots\right]. \end{aligned}$$

Then we have

$$U_1^T A V_1 \underbrace{=}_{\text{definition}} S = \begin{bmatrix} \sigma_1 & w^T \\ 0 & B \end{bmatrix},$$

where 0 is the $m-1$ column vector, w^T is a $n-1$ row vector, and B has dimensions $(m-1) \times (n-1)$.

Note, the top-left comes from

$$\begin{aligned} Av_1 &= \sigma_1 u_1 \\ u_1^T A v_1 &= \sigma_1 \underbrace{u_1^T u_1}_{=1} \\ &= \sigma_1. \end{aligned}$$

The bottom-left is zero because

$$\begin{aligned} u_i^T A v_1 &= u_i^T (\sigma_1 u_1), \\ &= \sigma_1 u_i^T u_1, \\ &= 0, \ \forall i > 1. \end{aligned}$$

Now, we can show $w = 0$ as follows: Consider

$$\begin{aligned}
& \left\| \begin{bmatrix} \sigma_1 & w^T \\ 0 & B \end{bmatrix} \begin{bmatrix} \sigma_1 \\ w \end{bmatrix} \right\|_2 \\
= & \left\| \begin{bmatrix} \sigma_1^2 + w^T w \\ Bw \end{bmatrix} \right\|_2 \\
= & \sqrt{(\sigma_1^2 + w^T w)^2 + (Bw)^T Bw}, \\
= & \sqrt{(\sigma_1^2 + w^T w)^2 + w^T \underbrace{B^T B}_{\text{symm, + semi-def}} w}, \\
\geq & \sigma_1^2 + w^T w \\
= & \sqrt{\sigma_1^2 + w^T w} \left\| \begin{bmatrix} \sigma_1 \\ w \end{bmatrix} \right\|_2 .
\end{aligned}$$

By the induced matrix norm definition, this implies $\|S\|_2 \geq \sqrt{\sigma_1^2 + w^T w}$. However, since U_1 and V_1 are orthogonal, therefore $\|S\|_2 = \|A\|_2 = \sigma_1$. Thus we must have $w = 0$, and so

$$U_1^T A V_1 = \begin{bmatrix} \sigma_1 & 0 \\ 0 & B \end{bmatrix}.$$

For $n = 1$ (base case) this completes the proof. For $n > 1$ (induction case), by the inductive hypothesis, the SVD of B exists. Write $B = U_2 \Sigma_2 V_2^T$. Then if we let

$$A = \underbrace{U_1 \begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} \sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix}}_{\Sigma} \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & V_2 \end{bmatrix} V_1^T}_{V^T},$$

it is easy to verify that this is an SVD of A. (Some explanation is given below.)

Therefore, in either case, the SVD of A always exists. □

Additional Explanation: Earlier, we had

$$U_1^T A V_1 = \begin{bmatrix} \sigma_1 & 0 \\ 0 & B \end{bmatrix}.$$

Recall that U_1 and V_1 are orthogonal. Therefore left multiplying by U_1 and right multiplying by V_1 yields

$$A = U_1 \begin{bmatrix} \sigma_1 & 0 \\ 0 & B \end{bmatrix} V_1^T.$$

Thus it suffices to prove that

$$\begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & V_2 \end{bmatrix} = \begin{bmatrix} \sigma_1 & 0 \\ 0 & B \end{bmatrix},$$

where $B = U_2 \Sigma_2 V_2^T$.

19.3 Stability Comparison

This section gives a more detailed comparison of the stability of the two approaches to compute the SVD. Assume a **stable** algorithm is used for finding eigenvalues (e.g., QR Iteration) such that

$$|\tilde{\lambda}_k - \lambda_k| = O\left(\epsilon_{\text{machine}}\|A\|\right),$$

where $\tilde{\lambda}_k$ denotes the numerical approximation. This satisfies $\tilde{\lambda}_k = \lambda_k(A + \delta A)$, with $\frac{\|\delta A\|}{\|A\|} = O\left(\epsilon_{\text{machine}}\right)$. That is, we compute the exact eigenvalues for a slightly **perturbed** matrix, $A + \delta A$.

Applying this to $H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$, we can get the singular values with

$$|\tilde{\sigma}_k - \sigma_k| = |\tilde{\lambda}_k - \lambda_k| = O\left(\epsilon_{\text{machine}}\|H\|\right) = O\left(\epsilon_{\text{machine}}\|A\|\right).$$

If we instead applied our eigenvalue routine to $A^T A$

$$|\tilde{\lambda}_k - \lambda_k| = O\left(\epsilon_{\text{machine}}\|A^T A\|\right) \approx O\left(\epsilon_{\text{machine}}\|A\|^2\right).$$

Taking the square roots (i.e., divide by $\sqrt{\lambda_k}$) to get σ_k gives

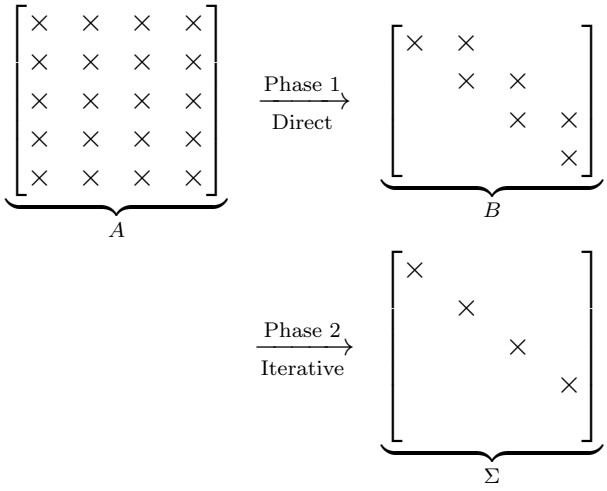
$$|\tilde{\sigma}_k - \sigma_k| = O\left(\frac{|\tilde{\lambda}_k - \lambda_k|}{\sqrt{\lambda_k}}\right) = O\left(\frac{\epsilon_{\text{machine}}\|A\|^2}{\sigma_k}\right).$$

This is quite inaccurate for $\sigma_k \ll \|A\|$.

19.4 Golub-Kahan Bidiagonalization

This section describes a way of accelerating the SVD computation. We apply another two-phase process, as we did for QR Iteration. We **pre-process** the matrix A to reduce the total cost of computing the SVD.

The idea is to first convert to a **bidiagonal** matrix and then extract the SVD! This process is depicted below.



Why bidiagonal? We do not have to maintain a similarity transformation for SVD (unlike for the eigendecomposition). Therefore, we can apply **different** Householder reflectors on left and right to introduce zeros:

$$\underset{A}{\begin{bmatrix}\times&\times&\times&\times\\ \times&\times&\times&\times\\ \times&\times&\times&\times\\ \times&\times&\times&\times\\ \times&\times&\times&\times\end{bmatrix}} \longrightarrow \underset{U_1^TA}{\begin{bmatrix}\times&\times&\times&\times\\ 0&\times&\times&\times\\ 0&\times&\times&\times\\ 0&\times&\times&\times\\ 0&\times&\times&\times\end{bmatrix}} \longrightarrow \underset{U_1^TAV_1}{\begin{bmatrix}\times&\times&0&0\\ 0&\times&\times&\times\\ 0&\times&\times&\times\\ 0&\times&\times&\times\\ 0&\times&\times&\times\end{bmatrix}} \longrightarrow$$

$$\underset{U_2^TU_1^TAV_1}{\begin{bmatrix}\times&\times&0&0\\ 0&\times&\times&\times\\ 0&0&\times&\times\\ 0&0&\times&\times\\ 0&0&\times&\times\end{bmatrix}} \longrightarrow \underset{U_2^TU_1^TAV_1V_2}{\begin{bmatrix}\times&\times&0&0\\ 0&\times&\times&0\\ 0&0&\times&\times\\ 0&0&\times&\times\\ 0&0&\times&\times\end{bmatrix}} \longrightarrow \text{etc...}$$

Bidiagonalization ultimately uses n reflectors on the left, $n-2$ on the right. Therefore, the cost of bidiagonalization is

$$\text{flops(bidagonalization)} \approx 2 \times \text{flops(QR)} \approx 4mn^2 - \frac{4}{3}m^3.$$

For the case of $m \gg n$, there exist faster algorithms (see Trefethen & Bau, Lecture 31 if curious).

For computing the SVD the cost is as follows. In practice, the cost of bidiagonalization phase $\approx O(mn^2)$ dominates over the eigendecomposition phase $\approx O(n^2)$. This cost is typically more expensive than other factorizations we have seen previously. However, the SVD computation is more numerically stable (i.e., preferable for ill-conditioned/rank-deficient matrices).

- Here, **rank-deficient** simply means not of full rank.

20 Lecture 20: Application - Image Compression

Outline

1. Best Approximation to A
2. Application of SVD to Image Compression
   (a) Image Compression Demo

20.1 Best Approximation to A

The singular value decomposition (SVD) can be thought of as representing A as the sum of rank-one matrices. In this lecture, we discuss approximating A using a **truncation** of this sum (i.e. omitting some terms at the end).

Theorem 20.1. *Let $m \geq n$. Let A be an $m \times n$ matrix, having rank r. Then A is the sum of r rank-one matrices, i.e. for $1 \leq j \leq r$, there exist scalars σ_j and vectors $u_j \in \mathbb{R}^m, v_j \in \mathbb{R}^n$ such that each $u_j v_j^T$ has rank 1, and*

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

Proof. From the definition of the SVD of A:

$$\begin{aligned} A &= \begin{bmatrix} u_1 & \cdots & u_m \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_r & \\ & & & 0 \end{bmatrix} \begin{bmatrix} v_1^T \\ \vdots \\ v_n^T \end{bmatrix} \\ &= \begin{bmatrix} u_1 & \cdots & u_m \end{bmatrix} \begin{bmatrix} \sigma_1 v_1^T \\ \vdots \\ \sigma_r v_r^T \\ 0 \end{bmatrix}, \\ &= \sigma_1 u_1 v_1^T + \cdots + \sigma_r u_r v_r^T = \sum_{j=1}^{r} \sigma_j u_j v_j^T. \end{aligned}$$

□

Exercise: What is the reduced SVD of the rank-1 matrix $A = xy^T$?

We can construct an approximate version of A, denoted A_k, using only the first k singular

values as follows:

$$\begin{aligned} A_k &= \begin{bmatrix} u_1 & \cdots & u_m \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_k & \\ & & & 0 \end{bmatrix} \begin{bmatrix} v_1^T \\ \vdots \\ v_n^T \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} u_1 & \cdots & u_k \end{bmatrix}}_{U_k} \underbrace{\begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_k \end{bmatrix}}_{\Sigma_k} \underbrace{\begin{bmatrix} v_1^T \\ \vdots \\ v_k^T \end{bmatrix}}_{V_k}. \end{aligned}$$

So, as above, we may write $A_k = U_k \Sigma_k V_k^T$. Theorem 20.2 gives the following results:

1. Among all matrices B with rank $\leq k$, A_k minimizes $\|A - B\|_2$. In other words, A_k provides the best rank k approximation of A.
2. The approximation error is given by the singular value σ_{k+1}.

Theorem 20.2. *Let $m \geq n$. Let A be an $m \times n$ matrix, having rank r, and SVD:*

$$A = \begin{bmatrix} u_1 & \cdots & u_m \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_r & \\ & & & 0 \end{bmatrix} \begin{bmatrix} v_1^T \\ \vdots \\ v_n^T \end{bmatrix}.$$

For any $1 \leq k \leq r$, define

$$A_k = \sum_{j=1}^{k} \sigma_j u_j v_j^T.$$

Then

$$\begin{aligned} \|A - A_k\|_2 &= \inf_{\text{rank}(B) \leq k} \|A - B\|_2 \\ &= \sigma_{k+1}. \end{aligned}$$

Proof. $\underline{\|A - A_k\|_2 = \sigma_{k+1}}$:

We will establish $\|A - A_k\|_2 = \sigma_{k+1}$, using the definition of SVD. We know that

$$\begin{aligned} A - A_k &= \left(\sum_{j=1}^{r} \sigma_j u_j v_j^T \right) - \left(\sum_{j=1}^{k} \sigma_j u_j v_j^T \right) \\ &= \sum_{j=k+1}^{r} \sigma_j u_j v_j^T \\ &= \begin{bmatrix} u_1 & \cdots & u_m \end{bmatrix} \begin{bmatrix} 0 & & & \\ & \sigma_{k+1} & & \\ & & \ddots & \\ & & & \sigma_r \\ & & & 0 \end{bmatrix} \begin{bmatrix} v_1^T \\ \vdots \\ v_n^T \end{bmatrix}, \end{aligned}$$

gives an SVD for $A - A_k$ (subject to reordering). We showed in Theorem 18.3 that $\|A\|_2 = \sigma_1$, so we have $\|A - A_k\|_2 = \sigma_{k+1}$ (i.e. the largest remaining singular value).

Part Two: Optimality:

We will show $\|A - A_k\|_2 = \inf_{\text{rank}(B) \leq k} \|A - B\|_2$ using a proof by contradiction. Towards a contradiction, suppose there exists B such that $\text{rank}(B) \leq k$ and $\|A - B\|_2 < \sigma_{k+1}$. That is, B is a strictly better approximation to A, with rank $\leq k$.

Recall that B is $m \times n$, i.e. we can view left multiplication by B as a linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$. By the rank-nullity theorem,

$$\begin{aligned} rank(B) + nullity(B) &= n \\ \Rightarrow nullity(B) &= n - rank(B). \end{aligned}$$

So $null(B)$ has dimension $\geq n - k$, and contains non-zero vectors v (such that $Bv = 0$, by definition).

If there are non-zero vectors in $null(B)$, then B kills them.

Further, if $k = r = n$, then $A_k = A$. ($\|A - A_k\| = 0$, as small as possible).

Observe that $null(B)$ and $span\{v_1, \ldots, v_{k+1}\}$ are subspaces of $\mathbb{R}^n$, with

- $nullity(B) \geq n - k$, and
- $\dim(span\{v_1, \ldots, v_{k+1}\}) = k + 1$.

Since $(n - k) + (k + 1) > n$, therefore $null(B)$ and $\text{span}\{v_1, \ldots, v_{k+1}\}$ must have a non-zero intersection, i.e., $\exists z \neq 0$ such that

$$z \in null(B) \cap \text{span}\{v_1, \ldots, v_{k+1}\}.$$

Without loss of generality, let $\|z\|_2 = 1$. We will obtain a contradiction by showing $\|A - B\|_2 \geq \sigma_{k+1}$.

Note $\|A - B\|_2^2 \geq \|(A - B)z\|_2^2$ (Recall the definition of the matrix 2-norm, $\|A\|_2 = \max \|Ax\|_2$ with $\|x\|_2 = 1$). Since $z \in \text{null}(B)$, $Bz = 0$, and therefore

$$\begin{aligned} \|(A - B)z\|_2^2 &= \|Az - Bz\|_2^2 \\ &= \|Az - 0\|_2^2 \\ &= \|Az\|_2^2 \\ &= \left\| \left(\sum_{i=1}^{n} \sigma_i u_i v_i^T \right) z \right\|_2^2 . \end{aligned}$$

For an arbitrary $0 \leq i \leq n$, the i^{th} term of the sum equals $\sigma_i u_i v_i^T z$.

Recall, $\|A\| \geq 0$, for any A, (by properties of $\| \cdot \|$).

We also have $z \in \text{span}\{v_1, \ldots, v_{k+1}\} \subseteq \mathbb{R}^n$. The above i^{th} term therefore equals

$$\begin{aligned} & \sigma_i u_i \underbrace{\left(v_i^T z\right)}_{\text{scalar}} \\ = \quad & \sigma_i \left(v_i^T z\right) u_i, \end{aligned}$$

and therefore the above squared 2-norm expression equals

$$\begin{aligned} & \left(\sum_{i=1}^{n} \sigma_i \left(v_i^T z\right) u_i\right)^T \left(\sum_{j=1}^{n} \sigma_j \left(v_j^T z\right) u_j\right) \\ = \quad & \sum_{i=1}^{n} \sigma_i^2 \left(v_i^T z\right)^2, \text{ using orthogonality of the } u_i s \\ = \quad & \sum_{i=1}^{k+1} \sigma_i^2 \left(v_i^T z\right)^2, \text{ since } z \in \text{span}\{v_1, \ldots, v_{k+1}\} \\ \geq \quad & \sigma_{k+1}^2 \sum_{i=1}^{k+1} \left(v_i^T z\right)^2, \text{ by the ordering of the } \sigma_i s. \end{aligned}$$

Now I claim that $\sum_{i=1}^{k+1} \left(v_i^T z\right)^2 = 1$. We have assumed that $\|z\|_2 = 1$. Since $z \in \text{span}\{v_1, \ldots, v_{k+1}\}$, we may write $z = \sum_{\ell=1}^{k+1} c_\ell v_\ell$, for some c_ℓs. Then we have

$$\begin{aligned} 1 \quad = \quad & \|z\|_2 \\ = \quad & \sqrt{z^T z} \\ = \quad & \sqrt{\left(\sum_{\ell=1}^{k+1} c_\ell v_\ell\right)^T \left(\sum_{j=1}^{k+1} c_j v_j\right)} \\ = \quad & \sqrt{\sum_{\ell=1}^{k+1} c_\ell^2 \left(v_\ell^T v_\ell\right)}, \text{ by orthogonality of the } v_\ell s \\ = \quad & \sqrt{\sum_{\ell=1}^{k+1} c_\ell^2}, \text{ since each } \|v_\ell\|_2 = 1, \text{ and so} \\ 1 \quad = \quad & \sum_{\ell=1}^{k+1} c_\ell^2. \end{aligned}$$

Now we can compute

$$\begin{aligned}
\sum_{i=1}^{k+1} \left(v_i^T z\right)^2 &= \sum_{i=1}^{k+1} \left(v_i^T \left(\sum_{\ell=1}^{k+1} c_\ell v_\ell\right)\right)^2 \\
&= \sum_{i=1}^{k+1} c_i^2 \left(v_i^T v_i\right)^2, \text{ by orthogonality of the } v_i s \\
&= \sum_{i=1}^{k+1} c_i^2, \text{ since each } \|v_i\|_2 = 1 \\
&= 1, \text{ as claimed.}
\end{aligned}$$

Putting everything together, we finally get

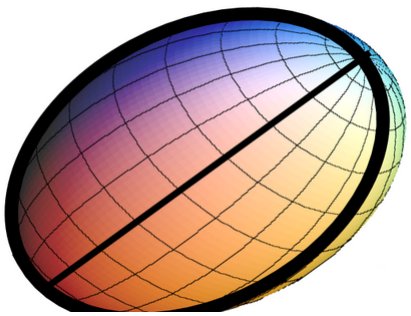
$$\begin{aligned}
\|A - B\|_2^2 &\geq \|(A - B)z\|_2^2 \\
&\geq \sigma_{k+1}^2 \sum_{i=1}^{k+1} \left(v_i^T z\right)^2 \\
&= \sigma_{k+1}^2,
\end{aligned}$$

implying $\|A - B\|_2 \geq \sigma_{k+1}$ and contradicting the fact that $\|A - B\|_2 < \sigma_{k+1}$. Hence no such B can exist. Note that the analogous statement holds true for the Frobenius norm

$$\|A - A_k\|_F = \inf_{\text{rank}(B) \leq k} \|A - B\|_F = \sqrt{\sigma_{k+1}^2 + \sigma_{k+2}^2 + \cdots + \sigma_r^2}.$$

□

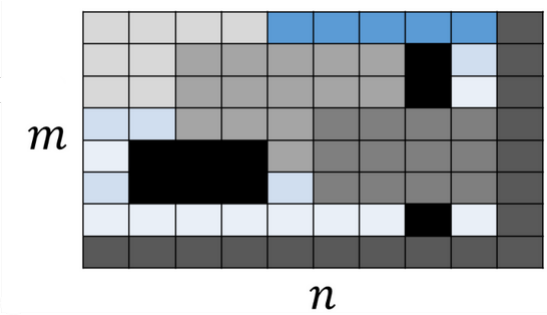
A geometric interpretation of the low rank approximation is as follows. Consider trying to determine the line segment that "best" approximates a (hyper)ellipsoid. The best approximation is the line segment along the longest axis of the (hyper)ellipsoid. This example corresponds to approximating A with $k = 1$. With $k = 2$, we can ask what ellipse gives "best" approximation of the (hyper)ellipsoid? The best ellipse is the one spanning the two longest axes (as shown with black curves in the figure below for the ellipsoid in $\mathbb{R}^3$). With larger k the same idea holds.



20.2 Application of SVD to Image Compression

The SVD can be used to produce a cheaper approximate version of an image (or other dataset) that captures the "most important" parts.

Consider and $m \times n$ pixel (grayscale) image as an $m \times n$ matrix A where A_{ij} is the intensity of the pixel (i, j). If we can store fewer than mn entries, we have a compressed representation.



Let $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$ be the best rank-k approximation of A. Then A_k gives a compressed version of the image A using the first k singular values. For example, given an input image with $m = 320, n = 200$. For A_k, we need to only store vectors $u_1, \ldots, u_k$ and $\sigma_1 v_1, \ldots, \sigma_k v_k$. Thus, we have $(m + n)k$ entries to store in total.

This gives a compression ratio of $\frac{(m+n)k}{mn}$. In our specific example we have

$$\frac{(320 + 200)k}{320 \cdot 200} \approx \frac{k}{123}.$$

The table below gives values for different k. As can be seen from the relative error and the compression ratios, this is an effective approach with small k.

k	Rel. err. σ_{k+1}/σ_1	Comp. ratio
3	0.155	2.4%
10	0.077	8.1%
20	0.04	16.3%

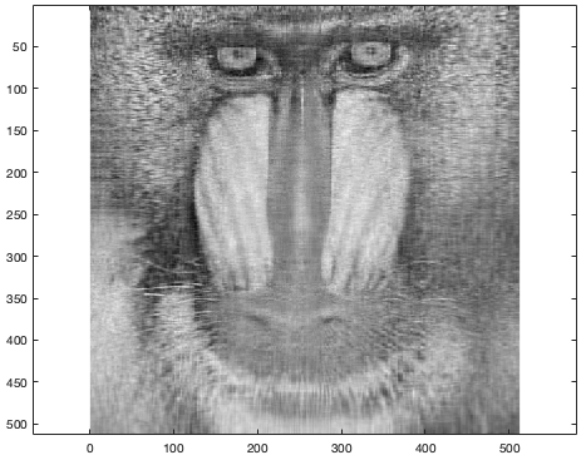
20.2.1 Image Compression Demo

An example of code for image compression using the above ideas is given in Algorithm 20.2.1. Note that sample code for colour images is given in SVDimageCompression.m (which just computes an SVD for each colour channel separately).

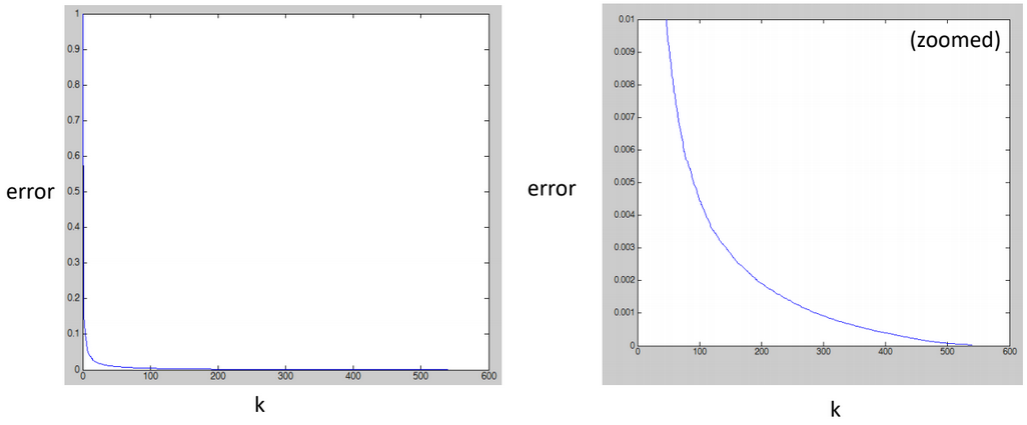
Algorithm 20.22 : Grayscale Image Compression

```
A=rgb2gray(imread('baboon.png'));
A=double(A);
[U,S,V]=svd(A);
k = 30;                                   ▷ try different choices
Ak=U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
colormap('gray');
imagesc(Ak);
axis equal;
```

Given the input figure on the left below, the code computes the compressed grayscale output on the right.



We had previously shown that $\|A - A_k\|_2 = \sigma_{k+1}$ gives the approximation error. So we can plot the relative error as $\frac{\sigma_{k+1}}{\sigma_1}$ against the choice of k (see below).



As depicted in the plot, the greater the k, the closer to the original image, thus a lower approximation error.

21 Lecture 21: Convergence of Iterative Methods

Outline

1. Introduction
2. Richardson Convergence
   (a) Choosing Optimal θ
3. Jacobi, Gauss-Seidel, & SOR Convergence
4. Convergence on Discrete Poisson Equation
   (a) Richardson
   (b) Jacobi
   (c) GS and SOR

21.1 Introduction

We will now revisit iterative schemes to analyze aspects of their convergence behaviour in detail. In this lecture we will study the stationary iterative methods:

1. Richardson,
2. Jacobi,
3. Gauss-Seidel, and
4. Successive-Over-Relaxation (SOR).

These methods were first discussed in Lecture 08.

Recall that the stationary iterative methods amount to different choices of M when splitting $A = M - N$. The generic iteration is

$$x^{k+1} = x^k + M^{-1}(b - Ax^k).$$

For each method we have the following splittings of the matrix A:

1. Richardson: $M = \frac{1}{\theta} I$ for scalar $\theta > 0$,
2. Jacobi: $M = D$,
3. Gauss-Seidel: $M = D - L$,
4. SOR: $\frac{1}{\omega} D - L$ for scalar $\omega > 0$.
   - $0 < \omega < 1$ indicates under-relaxation;
   - $1 < \omega$ indicates over-relaxation.

$$A = \begin{bmatrix} \ddots & & -U \\ & D & \\ -L & & \ddots \end{bmatrix}$$

We can rewrite the generic iteration as

$$x^{k+1} = \boxed{(I - M^{-1}A)}\, x^k + M^{-1}b.$$

Then we call $G = I - M^{-1}A$ the **iteration matrix** for the scheme. The method converges if and only if $\rho(I - M^{-1}A) < 1$, where $\rho(\cdot)$ denotes the **spectral radius** of a matrix (i.e., maximum eigenvalue magnitude – see Theorem 8.2). Note that a smaller ρ implies faster convergence to the solution. We will now consider the convergence behaviour for SPD matrices.

21.2 Richardson Convergence

The iteration matrix for the Richardson iteration is

$$\begin{aligned} G^{Rich} &= I - M_{Rich}^{-1}A \\ &= I - \theta A, \end{aligned}$$

for scalar $\theta > 0$. Let (λ, v) be an eigenpair of A. Then,

$$\begin{aligned} G^{Rich}v &= (I - \theta A)v \\ &= v - \theta\lambda v \\ &= (1 - \theta\lambda)v. \end{aligned}$$

Therefore, $\mu = 1 - \theta\lambda$ is an eigenvalue for G^{Rich}.

Lemma 21.1. *Let λ_{min} and λ_{max} satisfy $\lambda_{min} \leq \lambda_i \leq \lambda_{max}, \forall i$. Then $\rho(G^{Rich}) = \max\{|1 - \theta\lambda_{min}|, |1 - \theta\lambda_{max}|\}$.*

Proof.

$$\begin{aligned} \lambda_{min} \leq \lambda \leq \lambda_{max}, \\ 1 - \theta\lambda_{min} \geq 1 - \theta\lambda \geq 1 - \theta\lambda_{max}, \end{aligned}$$

$$\Rightarrow |\mu| \leq \max\{|1 - \theta\lambda_{min}|, |1 - \theta\lambda_{max}|\}.$$

□

Note, if $\lambda_{min} < 0$ and $\lambda_{max} > 0$ then either

$$\begin{aligned} 1 - \theta\lambda_{min} > 1 \text{ if } \theta > 0 \text{ or,} \\ 1 - \theta\lambda_{max} > 1 \text{ if } \theta < 0. \end{aligned}$$

Hence, $\rho(G^{Rich}) > 1$ for this case and Richardson will diverge for such matrices. (Recall the condition on ρ was necessary and sufficient for convergence – Theorem 8.2).

If we assume that A is SPD, then its eigenvalues **cannot** be negative.

Also, we usually assume that $\theta > 0$.

Theorem 21.1. *Assume all eigenvalues of A are positive (i.e., A is positive definite). Then Richardson converges iff $0 < \theta < \frac{2}{\lambda_{max}}$.*

Proof. If $0 < \theta < \frac{2}{\lambda_{max}}$, then multiplying through by λ_{max} (and inserting the obvious $\theta\lambda_{min} \leq \theta\lambda_{max}$) yields

$$\begin{aligned} 0 < \theta\lambda_{min} &\leq \theta\lambda_{max} < 2, \\ -2 < -\theta\lambda_{max} &\leq -\theta\lambda_{min} < 0, \quad \text{(multiply by -1)} \\ -1 < 1 - \theta\lambda_{max} &\leq 1 - \theta\lambda_{min} < 1 \quad \text{(add one)} \end{aligned}$$

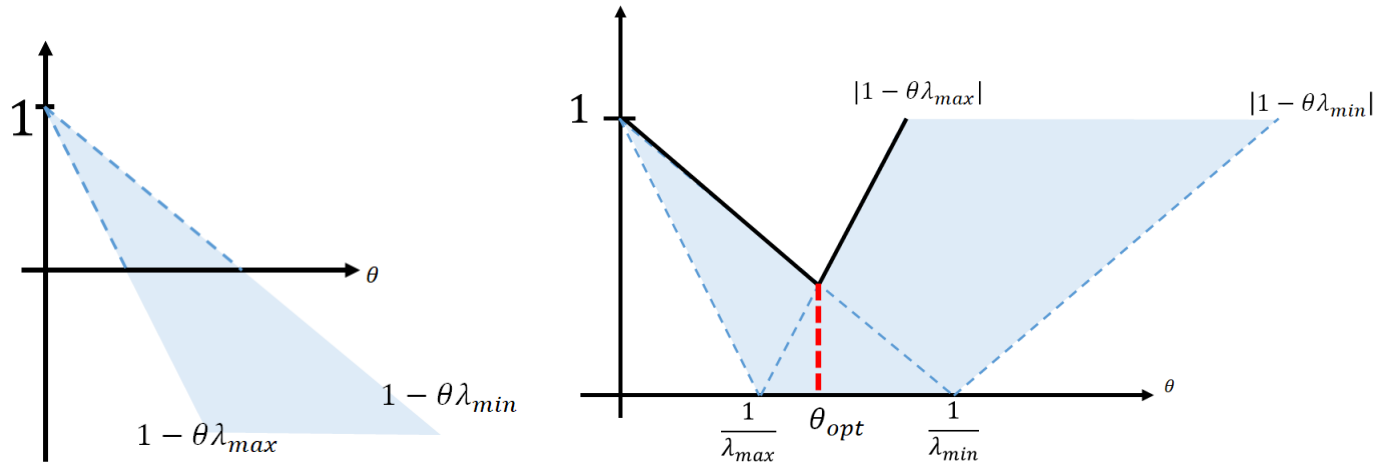


Figure 21.32: Finding the optimal θ for Richardson iteration.

Therefore, $|1-\theta\lambda_{max}|<1$ and $|1-\theta\lambda_{min}|<1 \Rightarrow \rho(G^{Rich})<1$.

For the other direction assume $\rho(G^{Rich})<1$, then

$$-1<1-\theta\lambda_{max}\leq\mu\leq 1-\theta\lambda_{min}<1. \tag{21.39}$$

From the left inequality of (21.39) we have

$$\begin{aligned} -1 &< 1-\theta\lambda_{max}, \\ -2 &< -\theta\lambda_{max}, \\ \Rightarrow \theta &< \frac{2}{\lambda_{max}}. \end{aligned}$$

The right inequality of (21.39) gives

$$\begin{aligned} 1-\theta\lambda_{min} &< 1, \\ -\theta\lambda_{min} &< 0, \\ \Rightarrow \theta &> 0. \qquad (\text{since } \lambda_{min}>0) \end{aligned}$$

So $0<\theta<\frac{2}{\lambda_{max}}$. □

21.2.1 Choosing Optimal θ

Assume A is PD. Assume $\theta>0$. To optimize convergence speed we must minimize $\rho(G^{Rich})$. Eigenvalues of $A\in[\lambda_{min},\lambda_{max}]$, so eigenvalues of Richardson iteration matrix $I-\theta A$ are in $[1-\theta\lambda_{max}, 1-\theta\lambda_{min}]$. Plotting this range gives the blue region in Figure 21.32 (left). But to get the minimum spectral radius, we need the absolute value. Reflecting negative parts over the x-axis gives Figure 21.32 (right).

For any choice of θ, the largest magnitude eigenvalue will sit at the top of the blue band, shown by the black line in Figure 21.32 (right). Thus ρ is minimized where the two lines

$|1-\theta\lambda_{min}|$ and $|1-\theta\lambda_{max}|$ intersect. Hence, we must find where $|1-\theta\lambda_{max}| = |1-\theta\lambda_{min}|$ since this is where the largest μ "switches" lines. That is, the optimal θ is when

$$\begin{aligned}
-(1-\theta_{opt}\lambda_{max}) &= 1-\theta_{opt}\lambda_{min} \\
-1+\theta_{opt}\lambda_{max} &= 1-\theta_{opt}\lambda_{min} \\
\theta_{opt}\lambda_{max}+\theta_{opt}\lambda_{min} &= 2 \\
\theta_{opt}(\lambda_{max}+\lambda_{min}) &= 2 \\
\theta_{opt} &= \frac{2}{\lambda_{min}+\lambda_{max}}.
\end{aligned}$$

Plugging θ_{opt} back in to find corresponding ρ gives

$$\begin{aligned}
\rho_{opt} &= 1-\theta_{opt}\lambda_{min}, \\
&= 1-\frac{2\lambda_{min}}{\lambda_{min}+\lambda_{max}}, \\
&= \left(\frac{\lambda_{max}+\lambda_{min}-2\lambda_{min}}{\lambda_{max}+\lambda_{min}}\right) \\
&= \left(\frac{\lambda_{max}-\lambda_{min}}{\lambda_{max}+\lambda_{min}}\right)\left(\frac{\frac{1}{\lambda_{min}}}{\frac{1}{\lambda_{min}}}\right) \\
&= \frac{\frac{\lambda_{max}}{\lambda_{min}}-1}{\frac{\lambda_{max}}{\lambda_{min}}+1} \\
&= \frac{\kappa_2(A)-1}{\kappa_2(A)+1}.
\end{aligned}$$

Recall that $\kappa_2(A) = \frac{|\lambda_{max}|}{|\lambda_{min}|}$ and $\lambda > 0$ was assumed in Theorem 21.1. Note that

1. we need eigenvalues (or estimates) to choose optimal θ, and
2. convergence can be slow, depending on λ's.

21.3 Jacobi, Gauss-Seidel, & SOR Convergence

In this section we give convergence results for Jacobi, Gauss-Seidel, and SOR.

Theorem 21.2. *If A and $2D-A$ are SPD, then the Jacobi iteration converges.*

Proof. Let μ be an eigenvalue of $G^J = I - M_J^{-1}A = I - D^{-1}A$, with eigenvector v. Then

$$\begin{aligned}
(I - D^{-1}A)v &= \mu v, \\
D^{-1}(D - A)v &= \mu v, \\
(D - A)v &= \mu Dv, \\
v^T(D - A)v &= \mu v^T Dv, \\
v^T Dv - v^T Av &= \mu v^T Dv, \\
v^T Dv - \mu v^T Dv &= v^T Av \\
(1 - \mu)v^T Dv &= v^T Av \\
&> 0, \text{ since } A \text{ is SPD.}
\end{aligned}$$

So $(1 - \mu)v^T Dv > 0$, which implies $\mu < 1$, because $v^T Dv > 0$, since A is SPD and hence D is SPD. Similarly, since $2D - A$ is SPD,

$$\begin{aligned}
v^T(2D - A)v &> 0 \\
v^T Dv - v^T Av &> -v^T Dv \\
v^T(D - A)v &> -v^T Dv.
\end{aligned}$$

Now note that

$$\begin{aligned}
\left(I - D^{-1}A\right)v &= \mu v \\
(D - A)\,v &= \mu Dv \\
v^T\,(D - A)\,v &= \mu v^T Dv,
\end{aligned}$$

and thus we can continue the above sequence of inequalities:

$$\begin{aligned}
\mu v^T Dv &> -v^T Dv \\
(\mu + 1)v^T Dv &> 0 \\
\Rightarrow \mu &> -1, \text{ since } D \text{ is SPD.}
\end{aligned}$$

Hence, $-1 < \mu < 1 \Rightarrow \rho\left(G^J\right) < 1$, i.e. a Jacobi iteration converges. □

Theorem 21.3. *If A is SPD then GS and SOR (for $0 < \omega < 2$) both converge.*

The optimal value of ω for SOR is not known in general. It is only known for special cases, e.g., for A that are tridiagonal and SPD we have

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho(G^J)^2}},$$

where G^J is Jacobi's iteration matrix.

Gauss-Seidel and Jacobi also converge for another class of matrices, called M-matrices.

Definition 21.1. *A is an M-matrix if*

(i) $a_{ii} > 0$,

(ii) $a_{ij} \leq 0$ *for* $i \neq j$,
(iii) A^{-1} *exists and* $(A^{-1})_{ij} \geq 0, \forall i, j$.

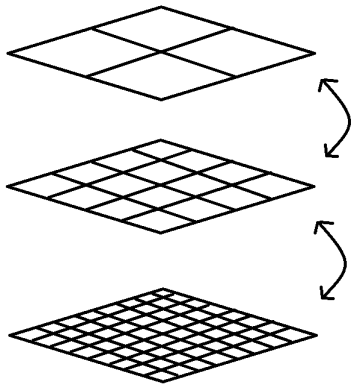
Theorem 21.4. *If A is an M-matrix then Jacobi and GS converge and*

$$\rho(I - M_{GS}^{-1}A) \leq \rho(I - M_J^{-1}A) < 1,$$

i.e., GS converges at least as rapidly as Jacobi.

Stationary iteration methods are often slow for low-frequency error components. Their modern use is often in "multigrid" methods, that use multiple grid levels to find a solution more quickly.

Since low frequency data has relatively high frequency on coarser grids, those errors can be eliminated more quickly, by transferring the solution between levels.



21.4 Convergence on Discrete Poisson Equation

Analytical expressions can be found for eigendecompositions of certain matrices. This can give us a sense for how our iterative schemes fare in practice. We will consider the familiar 2D finite difference Laplacian matrix for the Poisson equation:

$$\begin{aligned} -\nabla \cdot \nabla u &= f, \\ -u_{xx} - u_{yy} &= f. \end{aligned}$$

Note that the negative sign in front of the Laplacian makes the finite difference matrix positive definite (otherwise it is negative definite).

Theorem 21.5. *Let A be the negative of a 2D Laplacian matrix with cell size h and m grid points in each axis. Then the exact eigenvalues are*

$$\lambda_{ij} = \frac{4}{h^2}\left[\sin^2\left(\frac{\pi h i}{2}\right) + \sin^2\left(\frac{\pi h j}{2}\right)\right] \text{ for } 1 \leq i, j \leq m.$$

Note that the smallest eigenvalue is

$$\lambda_{min} = \frac{8}{h^2}\sin^2\left(\frac{\pi h}{2}\right),$$

and the largest eigenvalue is

$$\begin{aligned} \lambda_{max} &= \frac{8}{h^2}\sin^2\left(\frac{m\pi h}{2}\right), \\ &= \frac{8}{h^2}\sin^2\left(\frac{\pi}{2}(1-h)\right) \text{ , using } h = \frac{1}{m+1} \text{ or } mh = 1 - h, \\ &= \frac{8}{h^2}\cos^2\left(\frac{\pi h}{2}\right) \text{ , using } \sin\left(\frac{\pi}{2} - u\right) = \cos(u). \end{aligned}$$

The matrix A is both SPD and an M-matrix. Furthermore, the conditioning of A gets worse with finer grids, e.g., consider two grid resolution cases

$$\begin{aligned} 1)\ h &= \frac{1}{10},\ m = 9, \\ \lambda_{min} &\approx 19.6, \\ \lambda_{max} &\approx 780, \\ \kappa_2 &= \frac{\lambda_{max}}{\lambda_{min}} \\ &\approx 40. \end{aligned} \qquad \begin{aligned} 2)\ h &= \frac{1}{100},\ m = 99, \\ \lambda_{min} &\approx 19.7, \\ \lambda_{max} &\approx 80000, \\ \kappa_2 &= \frac{\lambda_{max}}{\lambda_{min}} \\ &\approx 4000. \end{aligned}$$

Finer Resolution → Worse conditioning.

21.4.1 Richardson

We will finish this lecture by showing convergence for the Poisson equation with the stationary iterative methods. For the Richardson iteration we have

$$\rho(I - \theta A) = \max\left\{\left|1 - \theta\frac{8}{h^2}\sin^2\left(\frac{\pi h}{2}\right)\right|, \left|1 - \theta\frac{8}{h^2}\cos^2\left(\frac{\pi h}{2}\right)\right|\right\}.$$

Hence, Richardson converges for

$$0 < \theta < \frac{2}{\lambda_{max}} = \frac{h^2}{4\cos^2\left(\frac{\pi h}{2}\right)}.$$

The optimal θ is

$$\theta_{opt} = \frac{2}{\lambda_{max} + \lambda_{min}} = \frac{2}{\frac{8}{h^2}\left(\cos^2\left(\frac{\pi h}{2}\right) + \sin^2\left(\frac{\pi h}{2}\right)\right)^1} = \frac{h^2}{4},$$

which gives optimal convergence with

$$\begin{aligned} \rho_{opt} &= \frac{\lambda_{max} - \lambda_{min}}{\lambda_{max} + \lambda_{min}} \\ &= \frac{\frac{8}{h^2}\left(\cos^2\left(\frac{\pi h}{2}\right) - \sin^2\left(\frac{\pi h}{2}\right)\right)}{\frac{8}{h^2}} \\ &= \cos^2\left(\frac{\pi h}{2}\right) - \sin^2\left(\frac{\pi h}{2}\right) \\ &= 1 - 2\sin^2\left(\frac{\pi h}{2}\right), \text{ since } \cos^2 x = 1 - \sin^2 x \\ &= \cos(\pi h), \text{ since } 1 - 2\sin^2(u) = \cos(2u). \end{aligned}$$

21.4.2 Jacobi

Since

$$\begin{aligned} D &= \frac{4}{h^2} I \\ &= \theta_{opt}^{-1} I, \end{aligned}$$

therefore we have that

$$\begin{aligned} G^J &= I - D^{-1} A \\ &= I - \theta_{opt} A. \end{aligned}$$

Therefore, Jacobi iteration is equivalent to the optimal Richardson iteration for this case, and hence

$$\rho = \cos(\pi h).$$

The Taylor expansion of $\cos(x)$ gives

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{4} + \dots.$$

Therefore,

$$\begin{aligned} \rho\left(I - D^{-1} A\right) &= \cos(\pi h) \\ &= 1 - \frac{\pi^2 h^2}{2} + O\left(h^4\right). \end{aligned}$$

For small h, Jacobi (and optimal Richardson) has slow convergence, since $\rho\left(G^J\right) \approx 1$.

21.4.3 GS and SOR

The spectral radius for Gauss-Seidel is the square of that for Jacobi [**Proof still needed**], and so we have

$$\begin{aligned} \rho\left(I - M_{GS}^{-1} A\right) &= \left[\rho(I - M_J^{-1} A)\right]^2 \\ &= \cos^2(\pi h), \\ &= 1 - \sin^2(\pi h), \\ &= 1 - \pi^2 h^2 + O(h^4). \text{ (Taylor expansion)} \end{aligned}$$

Notice there is no division by 2 compared to Jacobi, so GS convergence is 2 times better than Jacobi. However, this is only a constant factor, therefore GS is still slow for small h. This relationship is typical for SPD systems.

For SOR we have that

$$\omega_{opt} = \frac{2}{1 + \sin(\pi h)},$$

and thus

$$\begin{aligned}\rho_{opt} &= \omega_{opt} - 1 \\ &= \frac{1-\sin(\pi h)}{1+\sin(\pi h)}, \\ &= 1 - 2\pi h + O(h^2).\end{aligned}$$

Therefore, optimal SOR is much faster than GS/Jacobi/Richardson since the h factor is not squared. For example, with $h = 0.1$ we have

$$\begin{aligned}\rho(G^J) &= 0.95, \\ \rho(G^{GS}) &= 0.9, \\ \rho(G^{SOR}) &= 0.37.\end{aligned}$$

Run the demo code `StationaryIterativeConvergence.m` to see a comparison of Jacobi, GS, SOR, and optimal SOR iterations for solving the Laplace equation (i.e., the Poisson equation with $f = 0$). It is apparent from the demo that optimal SOR converges much faster.

22 Lecture 22: Convergence of Iterative Methods

Outline

1. Conjugate Gradient Convergence
2. Preconditioning Idea
   (a) Symmetric Preconditioning
3. Common Preconditioners
   (a) SGS Implementation
   (b) "Incomplete" Cholesky preconditioning
4. Extensions
5. (Last) Graphics Application

In this final lecture of new material for which you will be responsible, we will examine the convergence of the conjugate gradient method in more detail. We then introduce the idea of **preconditioning** to accelerate convergence of the conjugate gradient method. The conjugate gradient method was first introduced in Lecture 09.

22.1 Conjugate Gradient Convergence

Assumption: A is SPD.

We will need the following fact soon. For any k,

$$\begin{aligned} e^{(k)} &= x - x^{(k)}, \text{ so that} \\ e^{(0)} &= x - x^{(0)}, \text{ and} \\ Ae^{(0)} &= A\left(x - x^{(0)}\right) \\ &= Ax - Ax^{(0)} \\ &= b - Ax^{(0)} \\ &= r^{(0)}. \end{aligned}$$

Recall, at each step, CG finds the best solution $x^{(k)}$ in the span of search vectors so far, under the A-norm. That is, the CG method minimizes

$$\left\|e^{(k)}\right\|_A = \left\|x - x^{(k)}\right\|_A = \min_{x' \in \mathcal{K}_k(A)} \|x - x'\|_A ,$$

where x is the true solution and $\mathcal{K}_k$ is a Krylov subspace (see Lecture 09). We can write

$$\begin{aligned}
\left\|e^{(k)}\right\|_A &= \min \left\| x - \underbrace{\left(x^{(0)} + \sum_{i=0}^{k-1} \alpha_i p^{(i)} \right)}_{x' \in \mathcal{K}_k(A)} \right\|_A, \text{ for } \alpha_i \in \mathbb{R} \\
&= \min \left\| x - x^{(0)} - \sum_{i=0}^{k-1} \alpha_i p^{(i)} \right\|_A \\
&= \min \left\| e^{(0)} - \sum_{i=0}^{k-1} \alpha_i p^{(i)} \right\|_A \\
&= \min \left\| e^{(0)} + \sum_{i=0}^{k-1} \gamma_i A^{(i)} r^{(0)} \right\|_A, \text{ for } \gamma_i \in \mathbb{R},
\end{aligned}$$

since $\text{span}\{p^{(0)}, p^{(1)}, \ldots, p^{(k-1)}\} = \text{span}\{r^{(0)}, Ar^{(0)}, A^2 r^{(0)}, \ldots, A^{k-1} r^{(0)}\}$, as in Lecture 09.

Now define the following polynomial function

$$Q_{k-1}(x) = \gamma_0 + \gamma_1 x + \gamma_2 x^2 + \ldots + \gamma_{k-1} x^{k-1}.$$

Then taking the matrix A as the argument we have

$$\begin{aligned}
Q_{k-1}(A) &= \gamma_0 + \gamma_1 A + \gamma_2 A^2 + \ldots + \gamma_{k-1} A^{k-1}, \\
&= \sum_{i=0}^{k-1} \gamma_i A^i.
\end{aligned}$$

Now we can rewrite the error as

$$\begin{aligned}
e^{(k)} &= e^{(0)} + \sum_{i=0}^{k-1} \gamma_i A^i r^{(0)} \\
&= e^{(0)} + Q_{k-1}(A) r^{(0)} \\
&= e^{(0)} + Q_{k-1}(A) A e^{(0)}, \text{ since } r^{(0)} = A e^{(0)} \text{ as above} \\
&= (\underbrace{I + Q_{k-1}(A) A}_{\text{Another polynomial}}) e^{(0)}.
\end{aligned}$$

Define $P_k(x) = 1 + Q_{k-1}(x)x$, then $\deg(P_k) \leq k$ and $P_k(0) = 1$. So, we have $e^{(k)} = P_k(A) e^{(0)}$ and therefore

$$\|e^{(k)}\|_A = \min \left\{ \left\| P_k(A) e^{(0)} \right\|_A : P_k(A) = \text{poly. of deg. } \leq k \text{ with } P_k(0) = 1 \right\}.$$

That is, if $\tilde{P}_k(x)$ is any polynomial of degree $\leq k$ with $\tilde{P}_k(A) = 1$ then $\left\|e^{(k)}\right\|_A \leq \left\| \tilde{P}_k(A) e^{(0)} \right\|_A$. So CG finds (implicitly) the optimal polynomial to minimize the error in the A-norm. By choosing a particular polynomial we can obtain a bound on the error, given in the next theorem (see Shewchuk's article for expanded derivation).

Theorem 22.1.

$$\left\|e^{(k)}\right\|_A \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \left\|e^{(0)}\right\|_A.$$

[Proof still needed]

The actual CG convergence depends on all eigenvalues and is often better. The above is a worst case upper bound. Let us consider A with only 3 distinct eigenvalues

$$\lambda_1 < \lambda_2 < \lambda_3 \quad \text{, some eigenvalue may have mulitplicity } > 1.$$

We can show convergence in 3 iterations by choosing a "good" polynomial! Write the initial error in terms of unit orthogonal, eigenvectors v_j, of A

$$e^{(0)} = \sum_{j=1}^{n} \xi_j v_j, \text{ for some coefficients } \xi_i.$$

Also observe $P_k(A)v_j = P_k(\lambda_j)v_j$. Now, form a **Lagrange polynomial** $P_3(x)$ with degree ≤ 3 such that $P_3(0) = 1$ and $P_3(\lambda_j) = 0$ for $j = 1, 2, 3$.

Note that

$$\begin{aligned}
e^{(k)} &= P_k(A)\sum_{j=1}^{n} \xi_i v_j, \\
&= \sum_{j=1}^{n} \xi_j P_k(\lambda_j) v_j, \text{ since } v_j \text{ is an eigenvector} \\
\Rightarrow Ae^{(k)} &= \sum_{j=1}^{n} \xi_j P_k(\lambda_j)\lambda_j v_j, \text{ since } v_j \text{ is an eigenvector} \\
\Rightarrow \left\|e^{(k)}\right\|_A^2 &= \sum_{j=1}^{n} \xi_j^2 \left[P_k(\lambda_j)\right]^2 \lambda_j. \text{ because of orthogonality}
\end{aligned}$$

Hence,

$$\left\|e^{(3)}\right\|_A \leq \left\|P_3(A)e^{(0)}\right\|_A^2 = \sum_{j=1}^{3} \xi_j^2 \left[P_3(\lambda_j)\right]^2 \lambda_j = 0,$$

which means the error will be zero after 3 iterations. Since CG picks the optimal polynomial, it converges at least as fast as this, i.e., in 3 steps regardless of $\kappa(A)$.

22.2 Preconditioning Idea

We have seen that depending on eigenvalues of A, convergence of CG may still be rather slow. For the discretization of the Poisson equation, the asymptotic convergence of SOR (with optimal ω) matches CG. (Note that CG has the advantage of not having to find any

optimal parameter.) How can we improve the speed of convergence for CG even further? We speed up convergence using the idea of **preconditioning**.

We want to find a **modified** linear system, with **nicer properties**, but has the **same solution**. We want a better condition number $\kappa(A)$ (or more clustered eigenvalues), to achieve faster convergence, i.e., fewer CG iterations.

Consider the system

$$(M^{-1}A)x = M^{-1}b \quad \text{vs.} \quad Ax = b.$$

The same x is a solution to both problems, but we would prefer to solve the "easier" one. In this situation the matrix M is called a **preconditioner**. Similar to splitting in stationary iterative methods, we desire

1. $M \approx A$, (exercise: why?)
2. M^{-1} to be easy to build, or rather, $My = c$ to be cheap to solve. (exercise: why?)

22.2.1 Symmetric Preconditioning

To use CG, we need our modified system to also be SPD. Note that $M^{-1}A$ is not necessarily SPD, even if M and A are. If we let M be SPD, then a Cholesky factorization $M = LL^T$ exists. We instead form a new modified system as

$$\underbrace{L^{-1}AL^{-T}}_{\tilde{A}}\underbrace{L^Tx}_{\tilde{x}} = \underbrace{L^{-1}b}_{\tilde{b}}.$$

Notice $\tilde{A}$ is SPD by construction. The preconditioner is effectively **split** into left and right parts.

Claim: $\tilde{A}$ is similar to $M^{-1}A$.

Proof: Observe that

$$\begin{aligned} M^{-1}A &= \left(LL^T\right)^{-1}A \\ &= L^{-T}L^{-1}A, \text{ so that} \\ L^T\left(M^{-1}A\right)L^{-T} &= L^T\left(L^{-T}L^{-1}A\right)L^{-T} \\ &= L^{-1}AL^{-T} \\ &= \tilde{A}, \end{aligned}$$

and hence $\tilde{A}$ is a similarity transform of $M^{-1}A$.

Moral: This **could** now be solved by CG and this system has the same convergence behavior, since $M^{-1}A$ and $L^{-1}AL^{-T}$ have the same eigenvalues!

This leads to a naïve approach for preconditioning CG. The naïve approach is:

- form the modified system, $L^{-1}AL^{-T}\tilde{x} = \tilde{b}$,
- apply basic CG,
- transform solution $\tilde{x}$ to recover x (solve $L^Tx = \tilde{x}$).

The downside of this naïve approach is that it requires factoring M, which is potentially very costly. We must also run the full LL^T process on it, possibly leading to a large fill.

22.3 Common Preconditioners

A better approach would be to not form $\tilde{A}$ explicitly. Instead we modify the CG algorithm itself (via change of variables) to include a single new "preconditioning step"

$$x^k = M^{-1}r^k.$$

The theory only requires M to be SPD and we must be able to solve $Mz^k = r^k$ (hopefully cheaply). With this better approach, there is no need for factorization of $M = LL^T$. The preconditioned CG method is given in Algorithm 22.23. Note that we essentially add one extra line; if $M = I$, we recover basic CG.

Algorithm 22.23 Preconditioned CG Algorithm

$x^0 =$ initial guess
$r^0 = b - Ax^0$
for $k = 0, 1, 2, \ldots, n-1$
  $z^k = M^{-1}r^k$ (or preferably solve $Mz^k = r^k$)
  $\beta^k = \begin{cases} 0 & \text{if } k = 0 \\ \frac{(z^k, r^k)}{(z^{k-1}, r^{k-1})} & \text{otherwise} \end{cases}$
  $p^k = \begin{cases} z^k & \text{if } k = 0 \\ z^k + \beta^k p^{k-1} & \text{otherwise} \end{cases}$
  $\alpha^k = \frac{(z^k, r^k)}{(p^k, Ap^k)}$
  $x^{k+1} = x^k + \alpha^k p^k$
  $r^{k+1} = r^k - \alpha^k Ap^k$
end for

Common preconditioners for M often are related to our stationary iterative methods:

- Jacobi preconditioning:

$$M_J = D \qquad \text{(easiest! but not great)},$$

- Symmetric Gauss-Seidel:

$$M_{SGS} = (D - L)D^{-1}(D - U),$$

- Symmetric SOR:

$$M_{SSOR} = (D - \omega L)D^{-1}(D - \omega U).$$

22.3.1 SGS Implementation

One can express

$$M_{SGS} = (D - L)D^{-1}(D - U) = L_M U_M,$$

as an LU factorization where

$$\begin{aligned} L_M &= (D - L)D^{-1}, \\ U_M &= (D - U). \end{aligned}$$

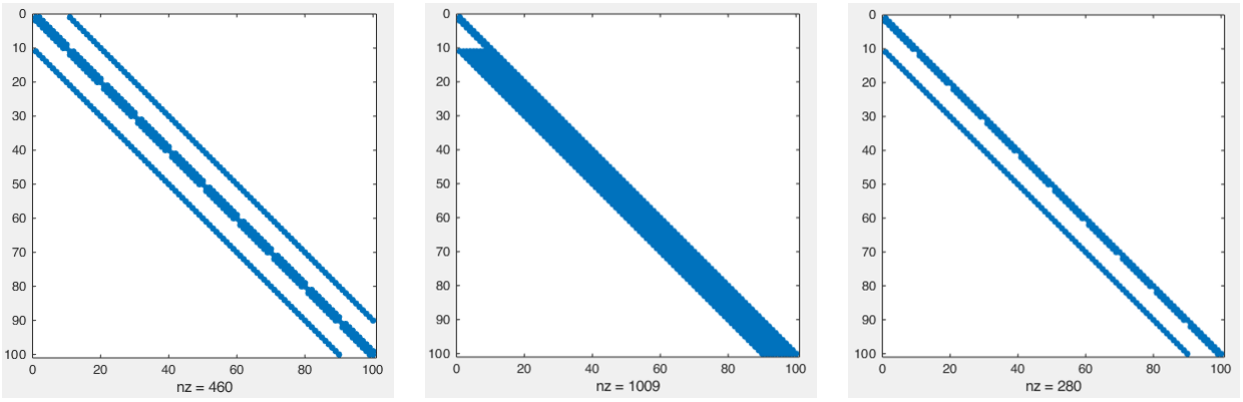
Since L_M is lower triangular and U_M is upper triangular, SGS preconditioning $z^k = M^{-1}r^k$ just requires two triangular solves:

$$\begin{aligned} (I - LD^{-1})y &= r^k, \\ (D - U)z^k &= y. \end{aligned}$$

22.3.2 "Incomplete" Cholesky preconditioning

The $L_M U_M$ factorization above gives us the hint for using other factorizations. With incomplete Cholesky (IC) preconditioning we find a partial Cholesky factorization where $LL^T \approx A$ (only **approximately**). We construct L via a Cholesky-like process, but skip (some or all) steps that would introduce new non-zero entries.

The IC preconditioner is not guaranteed to exist except in special cases (e.g., Laplacian, other M-matrices, etc.). The figure below shows an example of the sparsity pattern of the Cholesky and IC factorizations of the discrete Laplacian.



Laplacian A | Cholesky Factor | Incomplete Cholesky Factor

The sparsity pattern of L in the IC factorization stays close to A's compared to the full Cholesky factorization. Therefore, memory/speed cost remains low, but eigenvalues improve enough to accelerate CG convergence significantly.

For example, with the discrete Laplacian for $m = 14$

$$\kappa(A) \approx 90.5, \lambda_{max} \approx 1780, \lambda_{min} \approx 19.7, 23 \text{ CG iterations for } tol = 10^{-7}.$$

However, setting $L = \texttt{ichol}(A)$ and $\tilde{A} = L^{-1}AL^{-T}$ we have

$$\kappa(\tilde{A}) \approx 8.9, \lambda_{max} \approx 1.2, \lambda_{min} \approx 0.135, 14 \text{ PCG iterations for } tol = 10^{-7}.$$

Notice, we now have a better condition number, smaller eigenvalues, and fewer iterations for convergence.

MATLAB's CG routine is `pcg` for preconditioned conjugate gradient. It accepts preconditioner(s) as extra arguments. Incomplete Cholesky preconditioning is supported via `ichol`. The demo code PCGDemo.m compares CG, PCG, and optimal SOR for solving the Laplace equation. It can be seen from running this code that PCG converges much faster than the other two methods.

22.4 Extensions

A big limitation to the CG method is that it only applies to SPD matrices. However, many matrices encountered "in the wild" are not of this form. We briefly discuss how non-SPD matrices are handled in this section.

Option #1: One could solve the linear system as a least-squares problem. The solution to $\min_x \|Ax - b\|_2^2$ for square A satisfies $Ax = b$. So we just need to solve the normal equations $A^TAx = A^Tb$ with CG ("CGNR"). For this approach:

- Simple to code and A^TA is SPD!
- The condition is much worse ($\approx$ squared).

Option #2: Extensions of CG ideas ("Krylov solvers") exist for general systems:

- Symmetric **indefinite** systems: MINRES, SYMMLQ, ...,
- General **non-symmetric**: GMRES, BiCGSTAB, ...

Similar to CG, these aim to satisfy certain optimality properties. For example, MINRES seeks to minimize the norm of the residual.

For preconditioning there are also many others such as:

- (sparse) approximate inverse preconditioners,
- multilevel/multigrid preconditioners,
- parallel preconditioners,
- domain decomposition and block preconditioners, etc.

Preconditioners can also be applied to Krylov methods for indefinite and non-symmetric linear systems (MINRES, GMRES, etc.). Finding effective preconditioners for Krylov methods can depend heavily on the specific application problem/domain and its matrix structure. For more, see Preconditioning Techniques for Large Linear Systems: A Survey [Benzi 2002].

22.5 (Last) Graphics Application

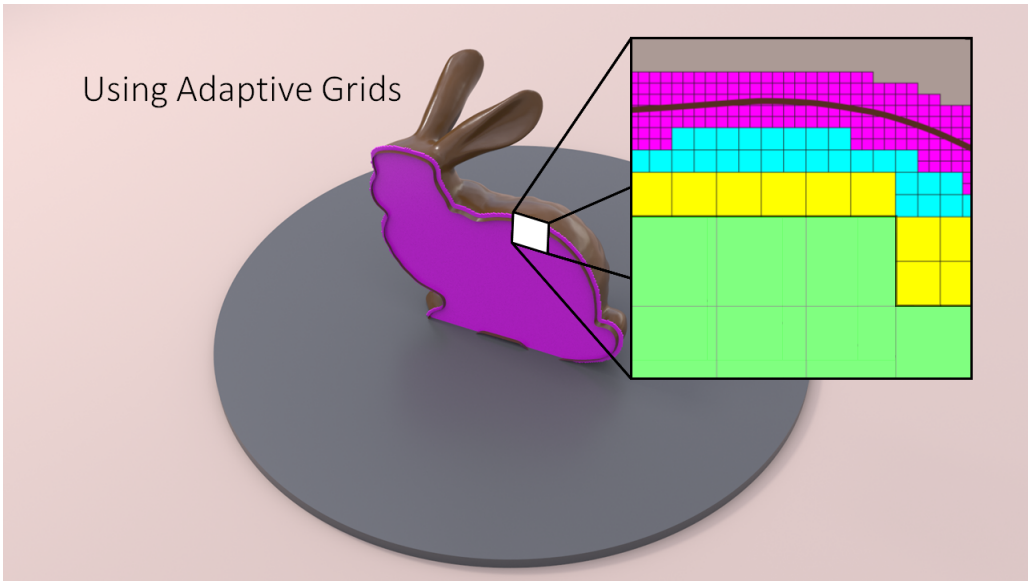
Dr. Christopher Batty here at UWaterloo (and some of his students) enjoy animating viscous liquids. However, the linear systems are very large and denser than Poisson/Laplacian. In

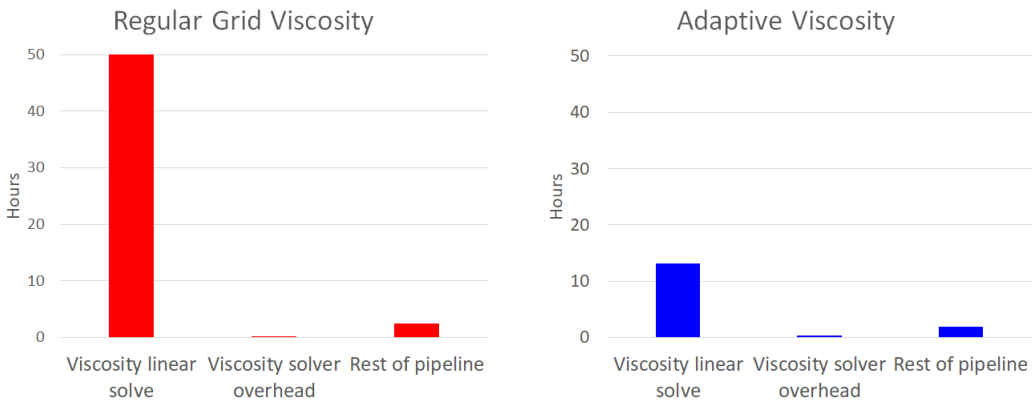
one example, of melting the Stanford Bunny, the cost of solving the linear system for the viscosity costs **way more** ($\approx 95\%$ of total) than any other step of fluid animation!

Reducing the cost has been tackled in two ways:

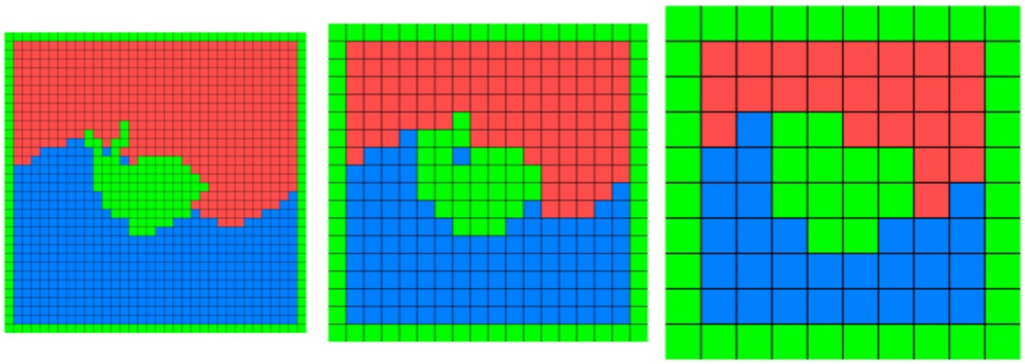
- Adaptive grid structures,
- Specialized multigrid preconditioners.

Adaptive grid structures reduce the cost by adding fine grids only where fine detail is necessary. In graphics, the interesting visual details are usual near the surface. Therefore, fine grids are used near the surface and larger grids are used far away from the surface. The size of the overall linear system is therefore smaller compared to using the fine grid throughout the whole domain.





The idea of using a **multigrid preconditioner** is as follows. One creates a multi-level approximation of the physical domain, perform "smoothing" at each level (using local Cholesky factorizations), and use the whole process as part of a CG preconditioner.



Solver	128^3 grid 3,035,346 DOFs	256^3 grid 24,303,388 DOFs	512^3 grid 194,503,048 DOFs
Multigrid (Naïve baseline)	20 iterations 377.3 s 0.256 GB 2 levels	34 iterations 975.3 s 2.18 GB 3 levels	39 iterations 2920.4 s 15.94 GB 4 levels
Multigrid (Improved baseline)	21 iterations 90.1 s 0.256 GB 2 levels	36 iterations 333.8 s 2.18 GB 3 levels	41 iterations 1512.4 s 15.94 GB 4 levels
Multigrid (Ours)	3 iterations 27.3 s 0.256 GB 2 levels	3 iterations 119.6 s 2.18 GB 3 levels	4 iterations 834.7 s 15.94 GB 4 levels
Eigen	377 iterations 35.6 s 1.536 GB	707 iterations 619.4 s 11.78 GB	out of memory
ICPCG	39 iterations 6.7 s 1.024 GB	53 iterations 95.3 s 8.32 GB	out of memory
PARDISO	1626.9 s 24.96 GB	N/A	N/A

The moral of the story is that familiarity with numerical linear algebra can enable huge speedups by:

a) Using existing algorithms more wisely,
b) Developing specialized algorithms for your problem/matrix.

23 Lecture 23: Principle Component Analysis (Optional)

Outline

1. Principle Component Analysis
   (a) PCA via Eigendecomposition
   (b) PCA via SVD
2. Applications
   (a) Dimensionality Reduction
   (b) Eigenfaces

23.1 Principle Component Analysis

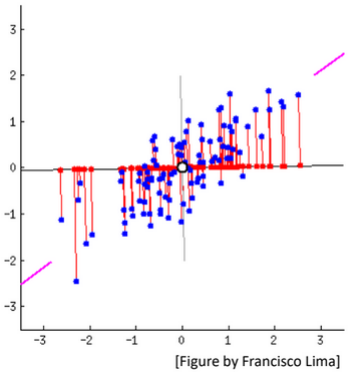
Another application of the SVD is for **principle component analysis (PCA)** of data. PCA is a method of extracting/expressing the important components in a data set. For example, consider an $m \times n$ matrix X (consisting of n column vectors x_i, each of length m).

- Each column represents a **sample**, and
- each row represents a **variable**.

Let's find an orthogonal transformation $PX = Y$ into a new basis, given by P, that "better" expresses the data. That is, we want to find the orthogonal direction vectors that express the most variation in the data (in descending order of importance).

This video explains the setup of the next diagram: https://youtu.be/FgakZw6K1QQ

The figure below (for $m = 2$) shows data samples (blue points) and their variation (red lines) from the line of best fit (black line). The displayed data is already centred around the origin. The black line is close to horizontal, so the difference lines are close to vertical.



[Figure by Francisco Lima]

First, we compute the **mean vector**

$$\langle x \rangle \equiv \frac{1}{n} \sum_{i=1}^{n} x_i$$

We will subtract the mean out from each sample to center the data around zero. Our **covariance matrix** is defined by

$$C \equiv \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \langle x \rangle)(x_i - \langle x \rangle)^T$$

After subtracting the mean vector off from our data, we can express it as

$$C \equiv \frac{1}{n-1}XX^T$$

This essentially measures how correlated entries in the sample vectors are to each other (i.e., how much they agree).

23.1.1 PCA via Eigendecomposition

The desired "principle components" are simply the **eigenvectors** of the covariance matrix

$$\frac{1}{n-1}XX^T = Q\Lambda Q^T,$$

so the orthogonal transformation is $P = Q^T$. The transformed data vectors in the new basis are $Y = PX$. Entries of y_i vectors express (orthogonal) contributions of principal components to x_i

$$y_i = \begin{bmatrix} p_1 \cdot x_i \\ p_2 \cdot x_i \\ p_3 \cdot x_i \\ \vdots \\ p_m \cdot x_i \end{bmatrix}.$$

We find y_i for a given x_i using dot products with the principle component vectors p_j.

23.1.2 PCA via SVD

Actually forming XX^T is not ideal numerically. Therefore, we apply a SVD approach that works directly with X instead. We can form the SVD of

$$\frac{1}{\sqrt{n-1}}X^T = U\Sigma V^T.$$

Then the columns of V are the desired principal components, i.e., $P = V$. Sample MATLAB code for PCA via the SVD is below, and in the `PCA.m` file.

```
function output = PCA(data)
    mn=mean(data,2);        %compute the mean
                            %for each dimension
    [M,N]=size(data);
    mns=repmat(mn,1,N);
    data=data-mns;          %subtract off the mean
```

```
                        %for each dimension
    Y=data'/sqrt(N-1);  %construct the matrix Y
    [U,S,PC]=svd(Y);    %extract principal components
    signals=PC'*data;   %project the original data
    output=signals;
end
```

For more information on PCA see "A Tutorial on Principle Component Analysis" [Schlens 2014].

23.2 Applications

23.2.1 Dimensionality Reduction

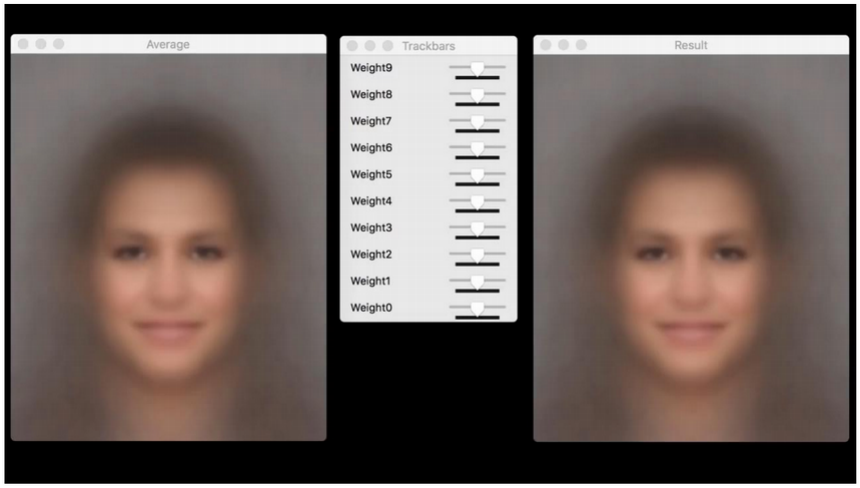
Recall the low-rank approximation property of SVD. We know that larger singular values contribute more to the data. **Dimensionality reduction** of data simply means discarding the principal components associated with smaller singular values. That is, take

$$Y_k = P_k X,$$

where we keep only the components corresponding to the k largest singular values. Now fewer variables are needed to express the data. The data becomes cheaper to store and manipulate, similar to the earlier image compression example.

23.2.2 Eigenfaces

PCA was used in an early approach to facial recognition [Turk and Pentland 1991]. Faces were treated as data vectors and assembled into a matrix X. Then, PCA/SVD/eigendecomposition was performed to find the basis (eigen)vectors. Weighted combinations of eigenvector/images ("eigenfaces") can recover the input faces. The video https://www.youtube.com/watch?v=J0arU2PAMls demonstrates this idea.



The steps are as follows. Given an input image:

1. Subtract out the mean face,
2. Find a weight vector by projecting it onto the eigenvectors (eigenface),
3. Select the person/face with the closest matching set of weights.

Here are some drawbacks of this approach:

- Very dependent on the data set (only works on faces belonging to the input),
- Many eigenvectors capture lighting effects rather than facial features,
- Based only on 2D images, so does not understand the 3D shape of faces/heads.

24 Lecture 24: Course Review and Wrap-Up

Outline

1. Final Exam Details
2. Course Review
3. Course Wrap-up
4. Student Perception Surveys

24.1 Final Exam Details

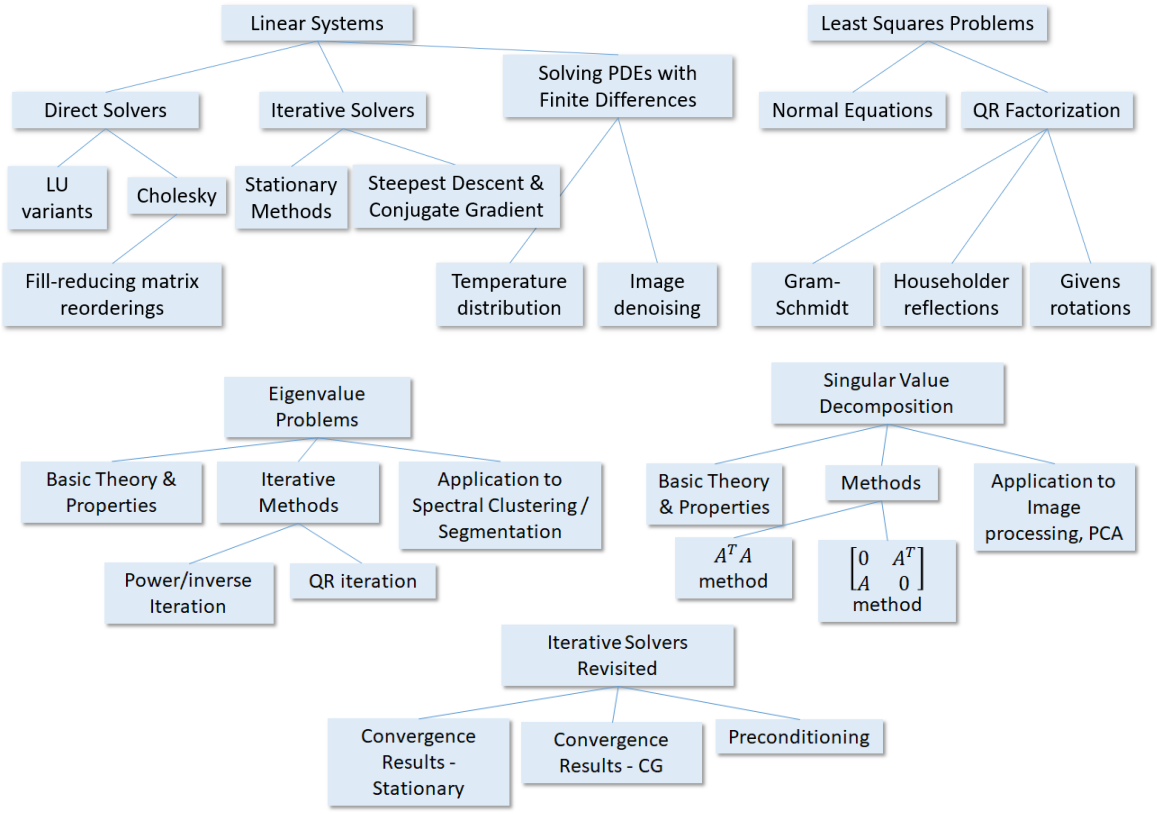
The details will be posted at https://www.student.cs.uwaterloo.ca/~CS475/schedule.shtml

24.2 Course Review

We finish this final lecture with a review of the central themes, which were:

- Direct vs. Iterative methods,
- Matrix properties, sparsity, and structure; their relationship to algorithms (and algorithm design),
- Matrix factorizations (LU/Cholesky, eigendecomposition, QR, SVD) and their uses (linear systems, least squares, eigenvalue problems),
- Orthogonality of vectors and matrices.

The topic map below gives an overview of all the material in this course.



24.3 Course Wrap-up

What questions do you have?

24.4 Student Perception Surveys

https://perceptions.uwaterloo.ca/

Index