AMS526: Numerical Analysis I (Numerical Linear Algebra for Computational and Data Sciences)

Lecture 19: Computing the SVD;
Sparse Storage Formats

Xiangmin Jiao

Stony Brook University

Outline

SVD of A and Eigenvalues of A^*A

- Intuitive idea for computing SVD of $A \in \mathbb{C}^{m\times n}$:
  - Form A^*A and compute its eigenvalue decomposition $A^*A = V\Lambda V^*$
  - Let $\Sigma = \sqrt{\Lambda}$, i.e., $\text{diag}(\sqrt{\lambda_1}, \sqrt{\lambda_2}, \dots, \sqrt{\lambda_n})$
  - Solve system $U\Sigma = AV$ to obtain U
- This method is efficient if $m \gg n$.
- However, it may not be stable, especially for smaller singular values because of the squaring of the condition number
  - For SVD of A, $|\tilde{\sigma}_k - \sigma_k| = O(\epsilon_{\text{machine}}\|A\|)$, where $\tilde{\sigma}_k$ and σ_k denote the computed and exact kth singular value
  - If computed from eigenvalue decomposition of A^*A, $|\tilde{\sigma}_k - \sigma_k| = O(\epsilon_{\text{machine}}\|A\|^2/\sigma_k)$, which is problematic if $\sigma_k \ll \|A\|$
- If one is interested in only relatively large singular values, then computing eigenvalues of A^*A is not a problem. For general situations, a more stable algorithm is desired.

A Different Reduction to Eigenvalue Problem

- Typical algorithm for computing SVD are similar to computation of eigenvalues
- Consider $A \in \mathbb{C}^{m \times n}$, then Hermitian matrix $H = \begin{bmatrix} 0 & A^* \\ A & 0 \end{bmatrix}$ has eigenvalue decomposition

$$H \begin{bmatrix} V & V \\ U & -U \end{bmatrix} = \begin{bmatrix} V & V \\ U & -U \end{bmatrix} \begin{bmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{bmatrix},$$

  where $A = U\Sigma V^*$ gives the SVD. This approach is stable.
- In practice, such a reduction is done implicitly without forming the large matrix
- Typically done in two phases

Two-Phase Method

- In the first phase, reduce to bidiagonal form by applying different orthogonal transformations on left and right, which involves $O(mn^2)$ operations
- In the second phase, reduce to diagonal form using a variant of QR algorithm or divide-and-conquer algorithm, which involves $O(n^2)$ operations for fixed precision

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}} \xrightarrow{\text{Phase 1}} \underset{B}{\begin{bmatrix} \times & \times & & \\ & \times & \times & \\ & & \times & \times \\ & & & \times \\ & & & \\ & & & \end{bmatrix}} \xrightarrow{\text{Phase 2}} \underset{\Sigma}{\begin{bmatrix} \times & & & \\ & \times & & \\ & & \times & \\ & & & \times \\ & & & \\ & & & \end{bmatrix}}.$$

We hereafter focus on the first phase

Golub-Kahan Bidiagonalization

- Apply Householder reflectors on both left and right sides

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}} \xrightarrow{U_1^*\cdot} \underset{U_1^*A}{\begin{bmatrix} \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}} \xrightarrow{\cdot V_1} \underset{U_1^*AV_1}{\begin{bmatrix} \times & \boldsymbol{\times} & \mathbf{0} & \mathbf{0} \\ & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}}$$

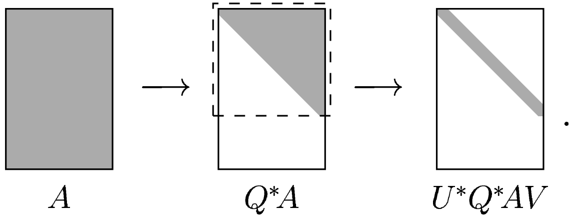
$$\xrightarrow{U_2^*\cdot} \underset{U_2^*U_1^*AV_1}{\begin{bmatrix} \times & \times & & \\ & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}} \xrightarrow{\cdot V_2} \underset{U_2^*U_1^*AV_1V_2}{\begin{bmatrix} \times & \times & & \\ & \times & \boldsymbol{\times} & \mathbf{0} \\ & & \boldsymbol{\times} & \boldsymbol{\times} \\ & & \boldsymbol{\times} & \boldsymbol{\times} \\ & & \boldsymbol{\times} & \boldsymbol{\times} \\ & & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}}.$$

- Work for Golub-Kahan bidiagonalization $\sim 4mn^2 - \frac{4}{3}n^3$ flops

Lawson-Hanson-Chan Bidiagonalization

- Speed up by first performing QR factorization on A

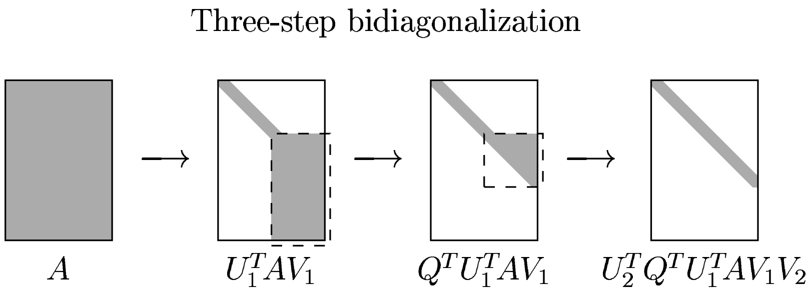
Lawson–Hanson–Chan bidiagonalization



- Work for LHC bidiagonalization $\sim 2mn^2 + 2n^3$ flops, which is advantageous if $m \geq \frac{5}{3}n$

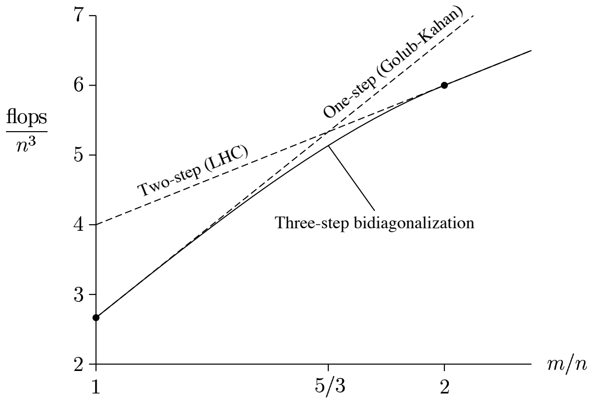
Three-Step Bidiagonalization

- Hybrid approach: Apply QR at suitable time on submatrix with 2 : 1 aspect ratio



- Work for three-step bidiagonalization is $\sim 4mn^2 - \frac{4}{3}n^3 - \frac{2}{3}(m-n)^3$

Comparison of Performance



Outline

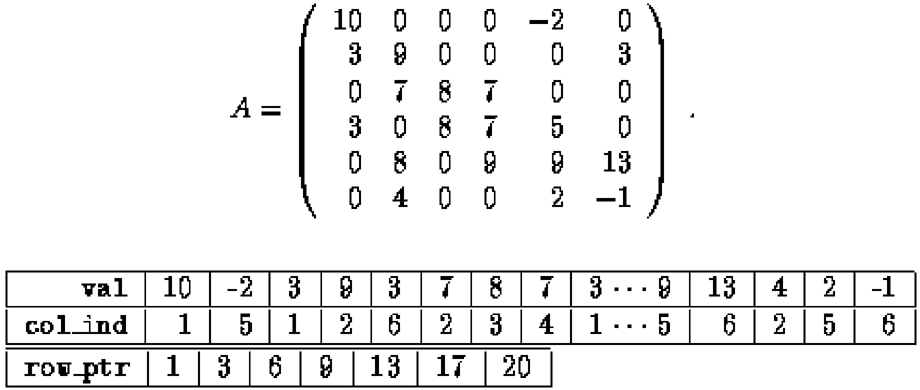
Sparse Linear System

- Boundary value problems and implicit methods for time-dependent PDEs yield systems of linear algebraic equations to solve
- A matrix is *sparse* if it has relatively few nonzeros in its entries
- Sparsity can be exploited to use far less than $O(n^2)$ storage and $O(n^3)$ work required in standard approach to solving system with dense matrix, assuming matrix is $n \times n$

Storage Format of Sparse Matrices

- Sparse-matrices are typically stored in special formats that store only nonzero entries, along with indices to identify their locations in matrix, such as
  - compressed-row storage (CRS)
  - compressed-column storage (CCS)
  - block compressed row storage (BCRS)
- Banded matrices have their own special storage formats (such as Compressed Diagonal Storage (CDS))
- See survey at http://netlib.org/linalg/html_templates/node90.html
- Explicitly storing indices incurs additional storage overhead and makes arithmetic operations on nonzeros less efficient due to indirect addressing to access operands, so they are beneficial only for very sparse matrices
- Storage format can have big impact the effectiveness of different versions of same algorithm (with different ordering of loops)
- Besides direct methods, these storage formats are also important in implementing iterative and multigrid solvers

Example of Compressed-Row Storage (CRS)



$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix} .$$

val	10	-2	3	9	3	7	8	7	3 ⋯ 9	13	4	2	-1
col_ind	1	5	1	2	6	2	3	4	1 ⋯ 5	6	2	5	6

row_ptr	1	3	6	9	13	17	20