Sparse Matrix Library

Xiaobai Jiang
Yuwei An
Shuting Zhang

Matrix API

Base Matrix(Virtual class)

- print() // print the matrix data structure
- show_matrix // print the matrix as general format
- multiply // multiply with base matrix pointer or std::vector
- add // add with base matrix pointer
- subtract //subtract with base matrix pointer
- transpose // return a new base matrix pointer with transpose data
- getRows // get row number
- getCols // get column number
- set // set value at (i, j)
- get // get value at (i, j)

Matrix Data Structure(derived from the Virtual class)

- COO(Coordinate)
- CSR(Compressed Sparse Row)
- CSC: Compressed Sparse Column

Matrix Generator

- Generate a sparse matrix as give data structure
  - `BaseMatrix* generate_matrix(const std::string& format, int m, int n, int density)`
  - `BaseMatrix* generate_spd_matrix(const std::string& format, int n);`

    Generate a random sparse matrix
    or Symmetric Positive Definite Matrix(SPD)

Matrix Decomposition

- LU Decomposition
- QR Decomposition
- Cholesky Decomposition

Static Public Member Functions

static void	**LU** (const **BaseMatrix** &A, **BaseMatrix** &L, **BaseMatrix** &U) Performs LU decomposition on matrix A such that A = L * U. More...
static void	**QR** (const **BaseMatrix** &A, **BaseMatrix** &Q, **BaseMatrix** &R) Performs QR decomposition using the Gram-Schmidt process. More...
static void	**Cholesky** (const **BaseMatrix** &A, **BaseMatrix** &L) Performs Cholesky decomposition on matrix A such that A = L * L^T. More...
static std::vector< double >	**solveLU** (const **BaseMatrix** &L, const **BaseMatrix** &U, const std::vector< double > &b) Solves Ax = b using LU decomposition (A = L * U). More...

What is Matrix Decomposition?

- **Definition**: the process of breaking a matrix into a product of simpler matrices, which makes certain matrix computations more efficient
- **Applications**
  - ✓ Solving linear systems
  - ✓ Eigenvalue problems
  - ✓ …

LU Decomposition: A = LU

- **Goal**: decomposes a matrix A into two matrices
  - L (*lower* triangular matrix)
  - U (*upper* triangular matrix)
- **Assumption:** A is square and non-singular
- **Steps**
  - Upper triangular matrix: $U_{ik} = A_{ik} - \sum_{j=0}^{i-1} L_{ij}U_{jk}$
  - Lower triangular matrix: $L_{ki} = \frac{1}{U_{ii}}(A_{ki} - \sum_{j=0}^{i-1} L_{kj}U_{ij})$
  - Diagonal of L : Set $L_{ii} = 1$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

LU Decomposition

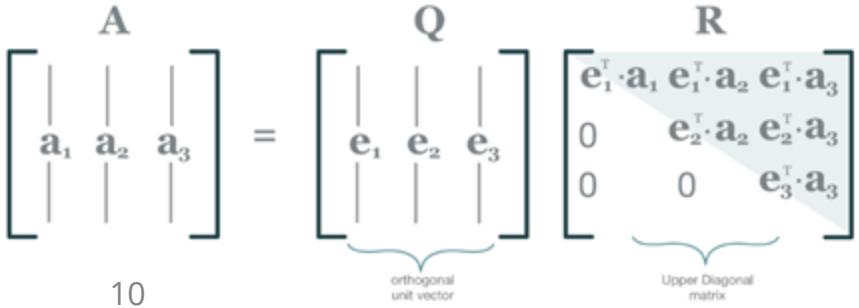
- **Pros**: Simple; easy to use in $Ax = b$ solving in $O(n^2)$ time
  - Convert into $LUx = b$
  - Forward substitution: solve $Ly = b$
  - Backward substitution: solve $Ux = y$
- **Cons**: Requires pivoting for numerical stability
- **Testcase**

```
matrix_size = 5;
BaseMatrix* lu_matrix = mg.generate_spd_matrix("COO", matrix_size);

BaseMatrix* L = mg.generate_matrix("COO", matrix_size, matrix_size, 0);
BaseMatrix* U = mg.generate_matrix("COO", matrix_size, matrix_size, 0);

Decomposition::LU(*lu_matrix, *L, *U);
```

```
[Generated Random Sparse Matrix in COO Format for LU Decomposition]
Row Indices: 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4
Column Indices: 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
Values: 2.47198 1.7375 1.27811 2.2062 1.64348 1.7375 1.78453 0.951362 1.832
74 1.63353 1.27811 0.951362 0.934324 1.12523 1.04285 2.2062 1.83274 1.12523
 2.40383 1.96611 1.64348 1.63353 1.04285 1.96611 1.87271
[L Matrix]
Matrix:
1 0 0 0 0
0.702878 1 0 0 0
0.517041 0.0940992 1 0 0
0.892484 0.500725 -0.156454 1 0
0.664842 0.849255 0.551538 0.985871 1
[U Matrix]
Matrix:
2.47198 1.7375 1.27811 2.2062 1.64348
0 0.563277 0.0530039 0.282047 0.478366
0 0 0.268499 -0.0420078 0.148088
0 0 0 0.287032 0.282977
0 0 0 0 0.0131537
[Frobenius Norm of Difference (L * U - Original Matrix)]
Frobenius Norm: 2.22045e-16
LU Decomposition Time: 5139 ns
```

QR Decomposition: A = QR

- **Goal**: decomposes a matrix A into two matrices
  - *Orthogonal* matrix Q $(QQ^T = Q^TQ = I)$
  - *upper triangular* matrix R
- **Steps (using Gram-Schmidt process)**
  - Orthogonalize
  - Construct $R = Q^TA$



$$q_1 = \frac{a_1}{\|a_1\|}$$

1. First vector

$$r_{ij} = q_i^\top a_j \quad \text{for } i < j$$

$$\tilde{a}_j = a_j - \sum_{i=1}^{j-1} r_{ij} q_i$$

2. Remove projections

$$q_j = \frac{\tilde{a}_j}{\|\tilde{a}_j\|}$$

3. Normalize

QR Decomposition

- **Pros**: Numerically more stable than LU for least squares
- **Cons**: Produces dense matrices even from sparse input
- **Testcase**

```
BaseMatrix* qr_matrix = mg.generate_spd_matrix("CSR", matrix_size);

BaseMatrix* Q_mat = mg.generate_matrix("CSR", matrix_size, matrix_size, 0);
BaseMatrix* R = mg.generate_matrix("CSR", matrix_size, matrix_size, 0);  //

Decomposition::QR(*qr_matrix, *Q_mat, *R);
```

```
[Generated Random Sparse Matrix in CSR Format for QR Decomposition]
Row Pointers: 0 5 10 15 20 25
Column Indices: 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
Values: 2.47198 1.7375 1.27811 2.2062 1.64348 1.7375 1.78453 0.95136
2 1.83274 1.63353 1.27811 0.951362 0.934324 1.12523 1.04285 2.2062 1
.83274 1.12523 2.40383 1.96611 1.64348 1.63353 1.04285 1.96611 1.872
71
[Q Matrix]
Matrix:
0.577359 -0.604661 -0.20485 -0.255721 0.440097
0.405813 0.603861 -0.1904 -0.638756 -0.162499
0.298518 -0.215819 0.82557 -0.159569 -0.396586
0.515284 -0.0244722 -0.352484 0.550231 -0.553971
0.383852 0.471763 0.340548 0.445399 0.561911
[R Matrix]
Matrix:
4.28153 3.58276 2.38303 4.34677 3.65504
0 0.547475 0.0644601 0.398583 0.602976
0 0 0.286903 -0.0496943 0.157978
0 0 0 0.283972 0.285818
0 0 0 0 0.00739119
Frobenius Norm of Difference (Q * R - Original Matrix)
Frobenius Norm: 3.14018e-16
QR Decomposition Time: 4707 ns
```

Cholesky Decomposition: $A = LL^T$

- **Goal**: decomposes a matrix A into the product $A = LL^T$
  - where L is a lower triangular matrix
- **Assumption**: A is a symmetric, positive-definite matrix
- **Steps** $x^T Ax > 0$ for all nonzero $x \in \mathbb{R}^n$
  - Diagonal entries (for each row)

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

  - Off-diagonal entries (compute elements below L_{ii})

$$L_{ij} = \frac{1}{L_{jj}}(A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}), \text{ for } i > j$$

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{00} & L_{10} & L_{20} \\ 0 & L_{11} & L_{21} \\ 0 & 0 & L_{22} \end{bmatrix}$$

Cholesky Decomposition

- **Pros**: Fast and memory efficient, numerically stable without pivoting
- **Cons**: Only applies to symmetric, positive-definite matrices
- **Testcase**

```
BaseMatrix* cholesky_matrix = mg.generate_spd_matrix("CSC", matrix_size);
BaseMatrix* chol_L = mg.generate_matrix("CSC", matrix_size, matrix_size, 0);
Decomposition::Cholesky(*cholesky_matrix, *chol_L);
```

```
[Generated Random Sparse Matrix in CSC Format for Cholesky Decomposition]
Column Pointers: 0 5 10 15 20 25
Row Indices: 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
Values: 2.47198 1.7375 1.27811 2.2062 1.64348 1.7375 1.78453 0.951362 1.8327
4 1.63353 1.27811 0.951362 0.934324 1.12523 1.04285 2.2062 1.83274 1.12523 2
.40383 1.96611 1.64348 1.63353 1.04285 1.96611 1.87271
[Cholesky L Matrix]
Matrix:
1.57225 0 0 0 0
1.1051 0.750518 0 0 0
0.812919 0.0706231 0.518169 0 0
1.40321 0.375803 -0.0810695 0.535754 0
1.0453 0.637381 0.28579 0.528184 0.114689
[Frobenius Norm of Difference (L * L^T - Original Matrix)]
Frobenius Norm: 0
Cholesky Decomposition Time: 2399 ns
```

EigenSolver

- **Power Iteration**
- **Inverse Iteration**
- **QR Iteration**
- **Lanczos Iteration**
- **Arnoldi Iteration**

Power Iteration

- - Goal: Compute the largest magnitude eigenvalue and its eigenvector.
- - Start with a random vector b_0.
  - Iterate: $b_{k+1} = A\, b_k$, then normalize.
  - Stop when $||b_{k+1} - b_k|| <$ tolerance.
  - Estimate $\lambda \approx |Ab_k| / |b_k|$ using Rayleigh quotient.
  - Converges to the dominant eigenvalue (largest in magnitude).

Inverse Iteration Method

- **- Goal: Compute an eigenvalue close to a guess u (default: 0).**
- **- Start with a random vector b_0.**
  - LU Decomposition: A = LU (precomputed).
  - Iterate: Solve LU $b_{k+1} = b_k \rightarrow b_{k+1} \approx A^{-1} b_k$.
  if shift: $b_{k+1} \approx (A-u\ I)^{-1} b_k$
  - Normalize x and check $|| b_{k+1} - b_k|| <$ tolerance.
  - Estimate $\lambda \approx |Ab_k| / |b_k|$ using Rayleigh quotient.
  - Finds eigenvalue closest to initial guess (or 0 if no shift).

QR Iteration Method

- **Goal: compute all eigenvalues of a square matrix based on repeated QR decompositions and similarity transforms**
- **Given a square matrix $A_0 = A$:**
  1. Compute QR decomposition: $A_k = Q_k R_k$
  2. Construct next iterate: $A_{k+1} = R_k Q_k$
  3. Repeat until A_k becomes nearly diagonal or upper-triangular.

Why Does It Work?

- - $A_{k+1} = Q_k^T A_k Q_k$ is a similarity transform.
- - Similar matrices share eigenvalues.
- - As $k \to \infty$, $A_k \to$ diagonal form.
- - Diagonal elements ≈ eigenvalues.

Lanczos Iteration Method

- - Goal: approximate eigenvalues of symmetric matrix $A \in \mathbb{R}^{nxn}$
  - Builds orthonormal basis Q_k for Krylov subspace:
    $K_k(A, q_0) = \text{span}\{q_0, Aq_0, A^2q_0, ..., Ak\text{-}1q_0\}$
  - Projects A to low-dim tridiagonal matrix:
    $T_k = Q_k T A Q_k \in \mathbb{R}^{kxk}$
  - $A \approx Q_k T_k Q_k T$
- -Eig(T_k) ≈ partial eig(A)
- Efficient for large sparse symmetric matrices.

Lanczos Iteration Method

- $Q_k = [q_0, q_1, ..., q_{k-1}]$ ($n \times k$ matrix with orthonormal cols)
  T_k = tridiagonal matrix with α_j on diag, β_j on off-diag
  Then:
  $A \approx Q_k T_k Q_k T \Rightarrow A Q_k = Q_k T_k$
- Main iteration:
  $A q_j = \beta_{j-1} q_{j-1} + \alpha_j q_j + \beta_j q_{j+1}$
  Implementation:
  $w = A q_j - \beta_{j-1} q_{j-1}$
  $\alpha_j = q_j T w$
  $w \leftarrow w - \alpha_j q_j$
  $\beta_j = ||w||$, then $q_{j+1} = w / \beta_j$
  Yields: tridiagonal T with α on diag, β on off-diagonals
- Solve eigenvalue by QR Iteration on T

$$T_k = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \cdots \\ \beta_1 & \alpha_2 & \beta_2 & \cdots \\ 0 & \beta_2 & \alpha_3 & \cdots \\ \vdots & \vdots & \ddots & \ddots \end{bmatrix}$$

Arnoldi Iteration Method

- **- Goal: approximate eigenvalues of a general (non-symmetric) matrix $A \in \mathbb{R}^{n \times n}$**
- **- Builds orthonormal basis Q_k for Krylov subspace:**
  $K_k(A, q_0) = \text{span}\{q_0, Aq_0, A^2q_0, ..., Ak\text{-}1q_0\}$
  - Projects A to low-dim upper Hessenberg matrix:
  $H_k = Q_k T A Q_k \in \mathbb{R}^{k \times k}$
  - $A \approx Q_k H_k Q_k T$
  - Eig(H_k) ≈ partial eig(A)

Arnoldi Iteration Method

- - $Q_k = [q_0, q_1, ..., q_{k-1}]$ (n × k matrix with orthonormal cols)
  - H_k = upper Hessenberg matrix (zero below subdiagonal)
  - Then: $A \approx Q_k H_k Q_k T \Rightarrow A Q_k = Q_k H_k$
- Main iteration:

  $w = A q_j$

  for i = 0 to j:

  $h_{ij} = q_i T w$

  $w \leftarrow w - h_{ij} q_i$

  $h_{i+1,j} = ||w||$, $q_{i+1} = w / h_{i+1,j}$
- Solve eigenvalue by QR Iteration on H

Eigen Solver Summary

Method	Requirement	Return
Power Iteration	-	max magnitude eigen value
Inverse Iteration	invertible	eigen value close to a given value
QR Iteration	-	all eigen values
Lanczos Iteration	real symmetric	k eigen values
Arnoldi Iteration	-	k eigen values

Test case on Symmetric Positive Definite Matrix

Eigen Solver Test Case

- Generate a spd matrix of 50, max_iter = 50, num_eigenvalues=10
- [Testing Power Iteration]
  Finish iteration in iter 6
  Estimated dominant eigenvalue (Power Iteration): 792.356
  Time taken: 0 ms
- [Testing Inverse Iteration]
  Finish iteration in iter 3
  Estimated smallest eigenvalue (Inverse Iteration): 5.25056e-08
  Time taken: 145 ms
- [Testing QR Iteration]
  [QR finished]
  Time taken: 19478 ms
  [QR eigenvalues (first 10)]
  792.356 12.8577 11.9656 10.7768 9.66794 8.73318 8.41568 7.88438 7.15266 7.14658
- [Testing Lanczos Iteration]
  [Lanczos finished]
  Time taken: 13 ms
  [Lanczos eigenvalues (first 10)]
  792.356 792.355 12.778 11.4522 9.91066 7.30989 4.48347 2.7544 1.04209 0.133806
- [Testing Arnoldi Iteration]
  [Arnoldi finished]
  Time taken: 13 ms
  [Arnoldi eigenvalues (first 10)]
  792.356 12.8503 11.6539 10.3237 8.81115 6.48019 3.72511 2.11372 0.924672 0.123859

SVD(Singular Value Decomposition

Any real m×n matrix A can be factored as
$A=U\Sigma V^T$
Singular values $\sigma_i = \sqrt{}$(eigenvalues of $A A^T$)
Orthonormal bases: columns of U span the column space; columns of V span the row space

Largest SVD Singular Value

Application:

1. In PCA, the top singular vector (associated with σ_1) defines the first principal component and the variance σ_1 stands the error distance

2. The adjacency matrix's largest singular (or eigen) value relates to connectivity measures, community detection, and thresholds for diffusion processes.

Lanczos Bidiagonalization

For sparse matrix, how to make full use of sparse matrix properties so to speed up the process of getting largest svd singular value

Algorithm 1 Lanczos Bidiagonalization for the Largest Singular Value

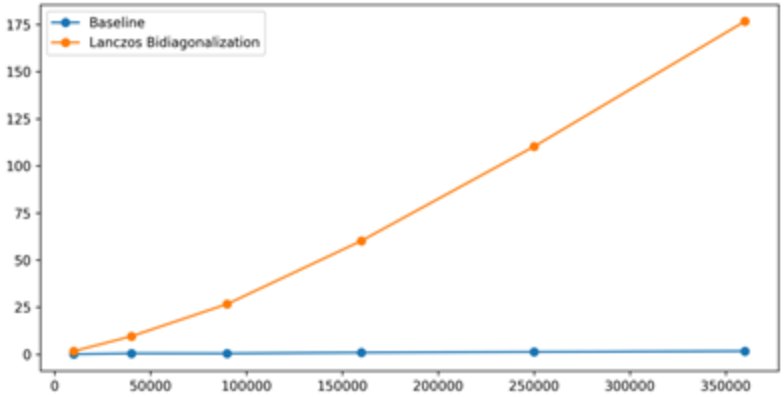
$A \in \mathbb{R}^{m \times n}$, number of steps k Approximation to the largest singular value σ_1 and vectors u_1, v_1 Choose a random unit vector $u_1 \in \mathbb{R}^m$, set $\beta_0 \leftarrow 0$ and $v_0 \leftarrow 0$ $j = 1$ to k $r \leftarrow A^T u_j - \beta_{j-1} v_{j-1}$ $\alpha_j \leftarrow \|r\|_2$ $v_j \leftarrow r/\alpha_j$ $p \leftarrow A v_j - \alpha_j u_j$ $\beta_j \leftarrow \|p\|_2$ $u_{j+1} \leftarrow p/\beta_j$ Construct the bidiagonal matrix

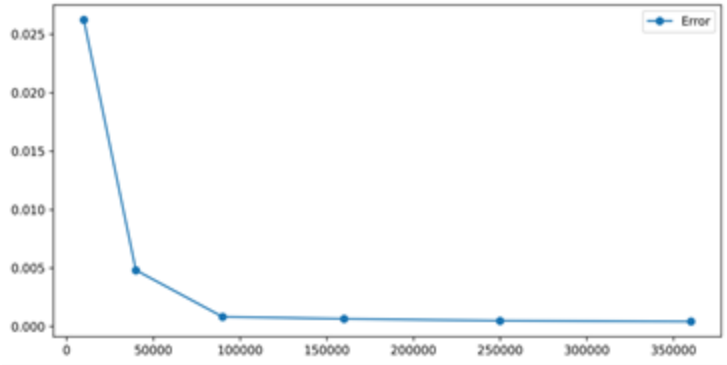
$$B_k = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ & \alpha_2 & \beta_2 & \\ & & \ddots & \beta_{k-1} \\ & & & \alpha_k \end{pmatrix} \in \mathbb{R}^{k \times k}$$

Compute the (small) SVD: $B_k = \widehat{U} \widehat{\Sigma} \widehat{V}^T$ $\hat{\sigma}_1 = \widehat{\Sigma}_{1,1}$ and $\hat{u}_1 = \sum_{j=1}^{k} \widehat{U}_{j,1} u_j$, $\hat{v}_1 = \sum_{j=1}^{k} \widehat{V}_{j,1} v_j$

In application we apply the svd lib on the bidiagonalization matrix B

Performance






```
(base) yuweia@ans-macbook-pro 18-847A-Project % ./bin/test_svd 600 600 76
============================
Sparse Matrix SVD Testing
============================
============================
Lanczos Bidiagonalization
============================

Estimated Largest singular value: 228.093
Time taken: 1.386 ms
Time taken: 184.053 ms
Largest singular value: 228.004
```

Faster and more acceptable with larger entries