Applied Math with MATLAB

CME 192 LECTURE 3

01/22/2026

Stanford University

Outline

Numerical Linear Algebra

- Sparse matrices
- Matrix decomposition
- Linear system solvers

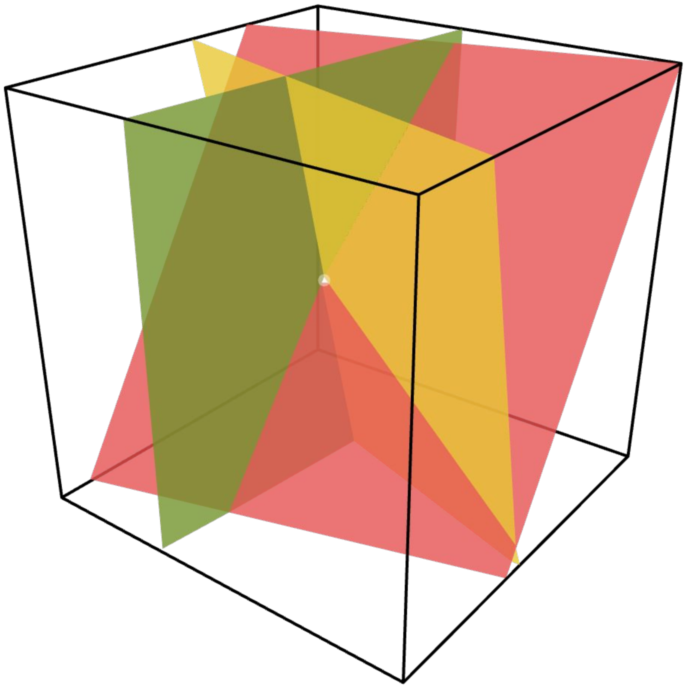
ODE and PDE

- Classification of ODEs, PDEs
- Numerical Methods for Solving
- Geometry Definitions + Workflow

Symbolic Math

Numerical Linear Algebra

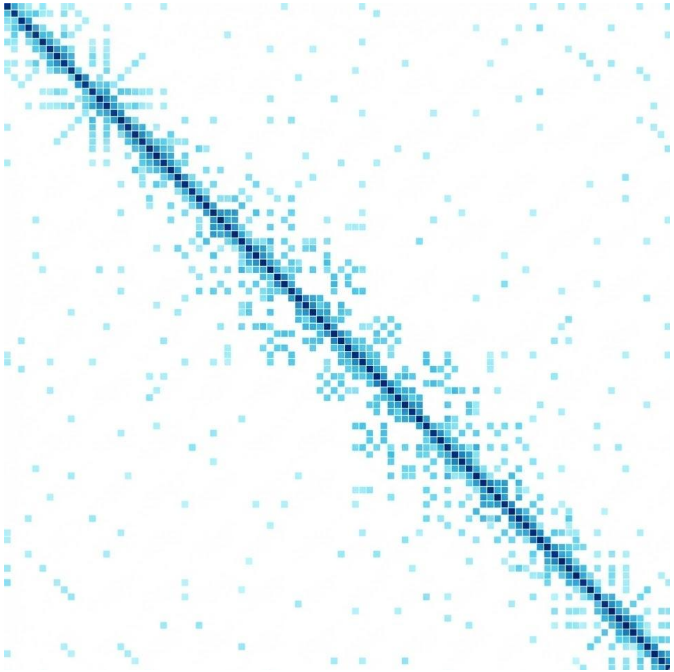
Applied Math with MATLAB



Stanford University

Dense vs. Sparse Matrices

This is a **sparse matrix**, a matrix with relatively small number of nonzero entries, compared to its size.



Let $A \in R^{m \times n}$ be a sparse matrix with v nonzeros.

Dense storage requires mn entries.

Sparse format saves storage.

Dense Matrix

1.2	0.5	-0.1	1.1	1.8	-4.3
3.7	-1.1	5.0	2.6	-3.4	4.8
1.6	-2.2	0.3	5.5	1.9	-0.9
-0.8	1.2	3.2	0.5	-4.3	1.7
1.9	-0.8	3.2	-5.1	2.4	6.5
1.3	-4.7	0.2	-3.8	4.1	3.8

Sparse Matrix

3.7	0	0	0	0	0
0	0	5.3	0	0	0
0	4.5	0	0	0	0
0	0	0	0	0	0
0	0	0	0	1.4	0
0	0	0	0	0	0

Sparse Matrix Storage Formats

1. Triplet format

Store nonzero values and corresponding row/column

Storage required = 3v (2v ints and v doubles)

General in that no assumptions are made about sparsity structure

Accepted as input for sparse matrix constructions by MATLAB

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Example: Dense vs. sparse storage (tridiagonal 1000×1000)

Dense (double): 1000×1000 = 1,000,000 entries

≈ 8,000,000 bytes (assuming 8 bytes per double)

Triplet storage: v = 3n − 2 = 2998 ⟶ 3v = 8994 numbers

≈ 71,952 bytes if each stored number is 8 bytes

Note: actual sparse storage depends on data types (indices vs values).

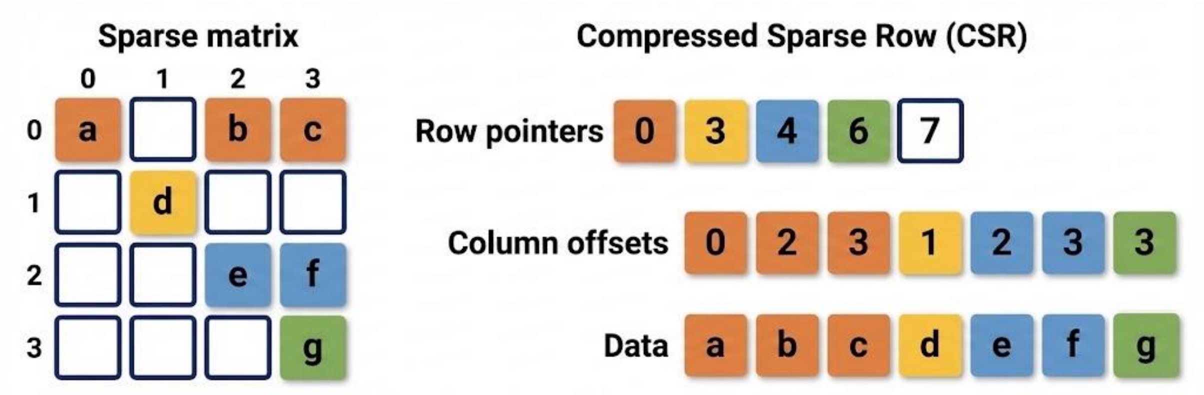
```
A = zeros(n,n);
A(1:n+1:end) = 2;
A(2:n+1:end) = -1;
A(n+1:n+1:end) = -1;

I = [ (1:n)';      (1:n-1)';    (2:n)'    ];
J = [ (1:n)';      (2:n)';      (1:n-1)' ];
V = [ 2*ones(n,1); -ones(n-1,1);
-ones(n-1,1) ];

S = sparse(I,J,V,n,n);
whos A S I J V
```

Name	Size	Bytes	Class	Attributes
A	1000x1000	8000000	double	
I	2998x1	23984	double	
J	2998x1	23984	double	
S	1000x1000	55976	double	sparse
V	2998x1	23984	double	

2. Compressed Sparse Row (CSR) format

Store nonzero values, corresponding column, and pointer into value array corresponding to first nonzero in each row

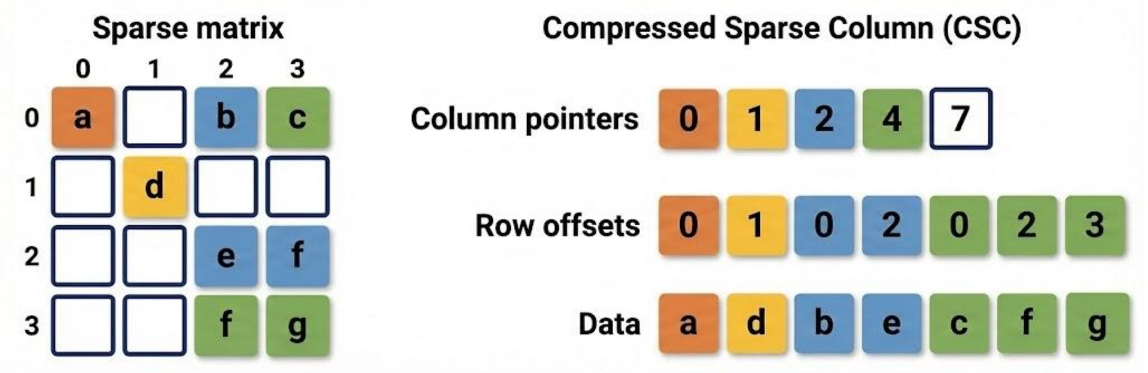
Storage required = 2v + m



3. Compressed Sparse Column (CSC) format

Store nonzero values, corresponding row, and pointer into value array corresponding to first nonzero in each column

Storage required = 2v + n



4. Others

Diagonal Storage format (useful for banded matrices)

Skyline Storage format

Block Compressed Sparse Row (BSR) format

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 0 \\ 8 & 0 & 9 & 10 & 11 & 0 \\ 0 & 13 & 0 & 0 & 14 & 15 \\ 0 & 0 & 16 & 0 & 17 & 18 \end{pmatrix}$$

(a) A sparse matrix

VAL

$$\begin{array}{|cccc|} \hline - & - & 1 & 2 \\ - & 3 & 4 & 5 \\ - & 6 & 7 & 0 \\ 8 & 9 & 10 & 11 \\ 13 & 0 & 14 & 15 \\ 16 & 17 & 18 & - \\ \hline \end{array}$$

OFFSET

$$\begin{array}{|cccc|} \hline -3 & -1 & 0 & 1 \\ \hline \end{array}$$

(b) Storage in diagonal format

Diagonal Storage Format

$$\begin{pmatrix} \boxed{\begin{matrix} 1 & 0 \\ 2 & 1 \end{matrix}} & \boxed{\begin{matrix} 6 & 7 \\ 8 & 2 \end{matrix}} & \begin{matrix} * & * \\ * & * \end{matrix} \\ \begin{matrix} * & * \\ * & * \end{matrix} & \boxed{\begin{matrix} 1 & 4 \\ 5 & 1 \end{matrix}} & \begin{matrix} * & * \\ * & * \end{matrix} \\ \begin{matrix} * & * \\ * & * \end{matrix} & \boxed{\begin{matrix} 4 & 3 \\ 0 & 0 \end{matrix}} & \boxed{\begin{matrix} 7 & 2 \\ 0 & 0 \end{matrix}} \end{pmatrix}$$

BSR Format

Break-Even Point for Sparse Storage

For $A \in R^{m \times n}$ with v nonzeros, there is a value of v where sparse vs dense storage is more efficient.

- For the triplet format, the cross-over point is defined by

$$3v = mn,$$

- If $v <= mn/3$ use sparse storage, otherwise use dense format.
- Cross-over point depends not only on *m, n, v*, but also on the data types of row, col, val.
- Besides storage efficiency, data access for linear algebra applications and ability to exploit symmetry in storage is also important.

Quiz

Suppose you have a tridiagonal 1000x1000 matrix. All entries in those three diagonals are non-zero. Calculate how many numbers can we omit if we store it as a triplet instead of a full matrix?

It sounds that a tridiagonal matrix is better stored as a triplet. For a tridiagonal $n \times n$ matrix, find the greatest n in which the triplet is not helpful
in saving memory for a tridiagonal matrix where all entries in those diagonals are non-zero.

Quiz

Suppose you have a tridiagonal 1000x1000 matrix. All entries in those three diagonals are non-zero. Calculate how many numbers can we omit if we store it as a triplet instead of a full matrix?

It sounds that a tridiagonal matrix is better stored as a triplet. For a tridiagonal n×n matrix, find the greatest n in which the triplet is not helpful in saving memory for a tridiagonal matrix where all entries in those diagonals are non-zero.

Solution

full matrix: 1000×1000 = 10^6

triplet: $v = 2998$, $3v = 8994$

For a tridiagonal matrix, $v = 3n - 2$. Solving $3v = n^2$ gives n ≈ 8.275.

Timing example (MATLAB): dense vs. sparse solve

```
n = 5000; e = ones(n,1);
A_dense  = diag(-2*e) + diag(e(1:end-1),1) +
diag(e(1:end-1),-1);
A_sparse = spdiags([e -2*e e], -1:1, n, n);
b = ones(n,1);

t_dense  = timeit(@() A_dense\b);
t_sparse = timeit(@() A_sparse\b);
[t_dense, t_sparse]
```

```
ans = 1x2
      0.0122    0.0001
```

Compare solve times on your machine (timeit reports seconds).

Create Sparse Matrices

Allocate space for m×n sparse matrix with v nonzeros:

```
S = spalloc(m; n; v)
```

Convert full matrix A to sparse matrix S:

```
S = sparse(A)
```

Create m×n sparse matrix with spare for v nonzeros from triplet (row,col,val):

```
S = sparse(row,col,val,m,n,v)
```

Create Sparse Matrices

Create matrix of 1s with sparsity structure defined by sparse matrix S:

```
R = spones(S)
```

Sparse identity matrix of size m×n :

```
I = speye(m,n)
```

Create sparse uniformly distributed random matrix from sparsity structure of sparse matrix S:

```
R = sprand(S)
```

Create Sparse Matrices

Create sparse uniformly distributed random matrix of size m×n with approximately mnρ nonzeros and condition number roughly κ (sum of rank 1 matrices):

```
R = sprand(m,n,rho,1/kappa)
```

Create sparse normally distributed random matrix:

```
R = sprandn(S), R = sprandn(m,n,rho,1/kappa)
```

Create sparse symmetric uniformly distributed random matrix:

```
R = sprandsym(S), R = sprandsym(n,rho,1/kappa)
```

Import from sparse matrix external format:

```
spconvert
```

Create sparse matrices from diagonals:

```
spdiags
```

Sparse storage information

Determine if matrix is stored in sparse format:

```
issparse(S)
```

Number of nonzero matrix elements:

```
nz = nnz(S)
```

Amount of nonzeros allocated for nonzero matrix elements:

```
nzmax(S)
```

Extract nonzero matrix elements: If (row, col, val) is sparse triplet of S:

```
val = nonzeros(S), [row,col,val] = find(S)
```

Sparse and dense matrix functions

Convert sparse matrix to dense matrix:

```
A = full(S)
```

Apply function (described by function handle func) to nonzero elements of sparse matrix:

```
F = spfun(func, S)
```

Plot sparsity structure of matrix:

```
spy(S)
```

Sparse Matrix Reordering

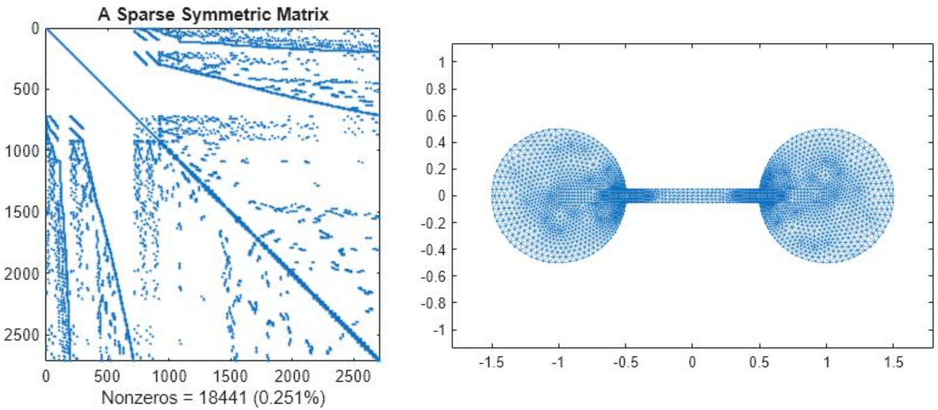
By reordering the rows and columns of a matrix, it is possible to reduce the amount of fill-in that factorization creates, thereby reducing the time and storage cost of subsequent calculations.

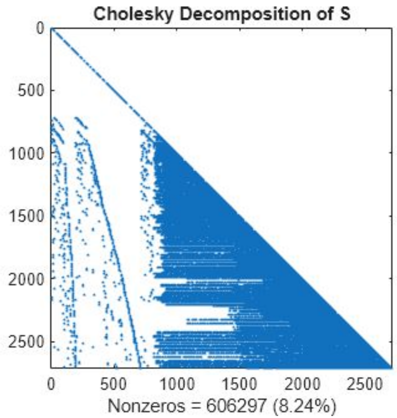
Reordering Functions

`amd` Approximate minimum degree permutation
`colamd` Column approximate minimum degree permutation
`colperm` Sparse column permutation based on nonzero count
`dmperm` Dulmage-Mendelsohn permutation/decomposition
`randperm` Random permutation
`symamd` Symmetric approximate minimum degreepermutation
`symrcm` Sparse reverse Cuthill-McKee ordering

A Sparse Symmetric Matrix

https://www.mathworks.com/help/matlab/math/sparse-matrix-reordering.html





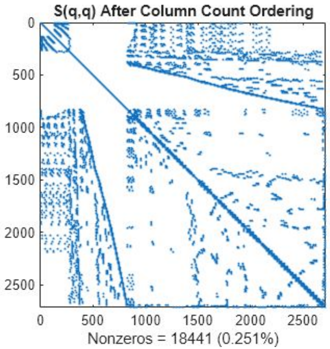
This `spy` plot shows a sparse symmetric positive definite matrix derived from a portion of the barbell matrix. This matrix describes connections in a graph that resembles a barbell.

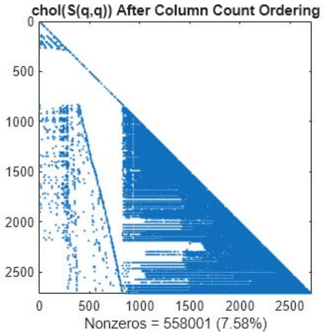
This is the Cholesky factor `L`, where `S = L*L'`. Notice that `L` contains *many* more nonzero elements than the unfactored `S`, because the computation of the Cholesky factorization creates fill-in nonzeros. These fill-in values slow down the algorithm and increase storage cost.

A Sparse Symmetric Matrix

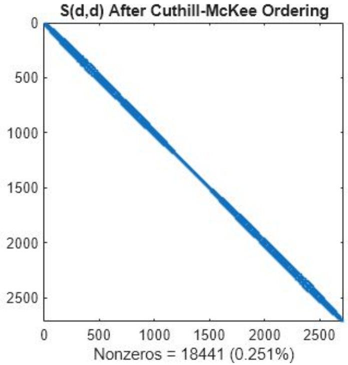
https://www.mathworks.com/help/matlab/math/sparse-matrix-reordering.html

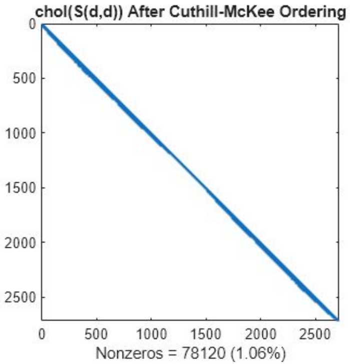
By reordering the rows and columns of a matrix, it is possible to reduce the amount of fill-in that factorization creates, thereby reducing the time and storage cost of subsequent calculations.





The `colperm` command uses the column count reordering algorithm to move rows and columns with higher nonzero count towards the end of the matrix.



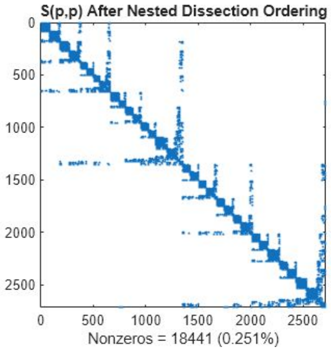


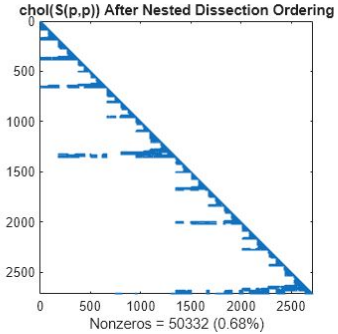
The `symrcm` command uses the reverse Cuthill-McKee reordering algorithm to move all nonzero elements closer to the diagonal, reducing the bandwidth of the original matrix.

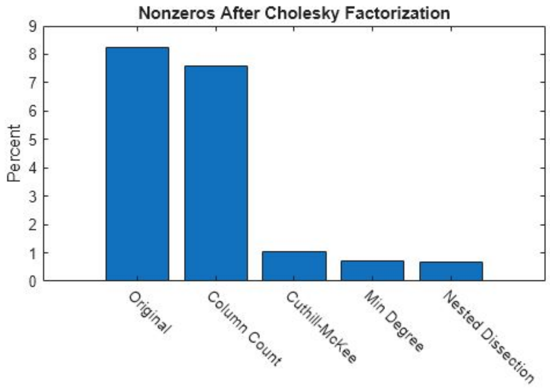
A Sparse Symmetric Matrix

https://www.mathworks.com/help/matlab/math/sparse-matrix-reordering.html

By reordering the rows and columns of a matrix, it is possible to reduce the amount of fill-in that factorization creates, thereby reducing the time and storage cost of subsequent calculations.







The `dissect` function uses graph-theoretic techniques to produce fill-reducing orderings. The algorithm treats the matrix as the adjacency matrix of a graph, coarsens the graph by collapsing vertices and edges, reorders the smaller graph, and then uses refinement steps to uncoarsen the small graph and produce a reordering of the original graph. The result is a powerful algorithm that frequently produces the least amount of fill-in compared to the other reordering algorithms.

This bar chart summarizes the effects of reordering the matrix before performing the Cholesky factorization. While the Cholesky factorization of the original matrix had about 8% of its elements as nonzeros, using `dissect` or `symamd` reduces that density to less than 1%.

Concrete example: reordering before factorization

```
S = sprandsym(2500, 0.002, 1e-2);
p1 = colperm(S);
p2 = amd(S);

R0 = chol(S);
R1 = chol(S(p1,p1));
R2 = chol(S(p2,p2));

[nnz(R0), nnz(R1), nnz(R2)]
```

```
ans = 1x3
      19658        7312        7280
```

Sparse Matrix Tips

Don't change sparsity structure (pre-allocate)

- Do not want to dynamically grows triplet
- Each component of triplet must be stored contiguously

Accessing values may be slow in sparse storage as location of row/columns is not predictable.

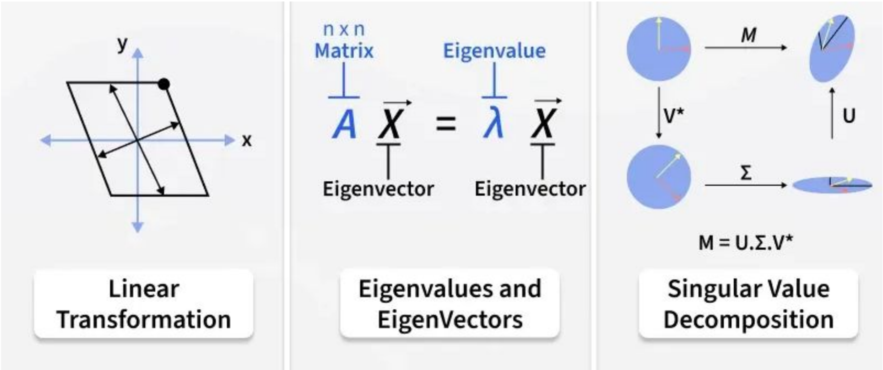
- If S (i , j) is requested, must search through row, col to find i, j

Component-wise indexing to assign values is expensive

- Requires accessing into an array
- If S (i , j) is previously zero, then S (i , j)= c changes sparsity structure

Solving Linear Systems

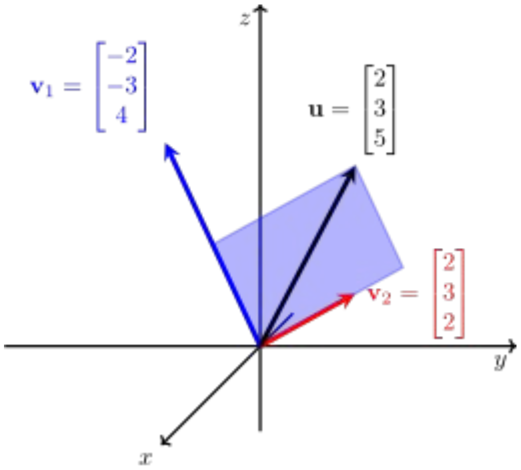
Applied Math with MATLAB



Some basic concepts in linear algebra:

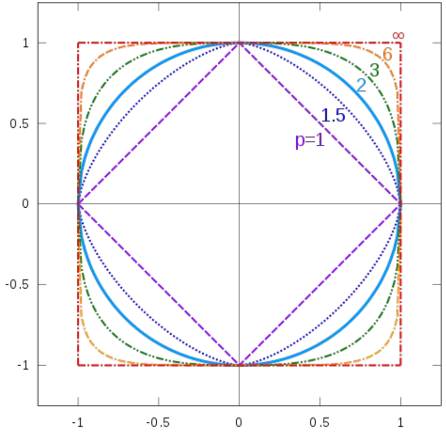
Rank

- the dimension of the vector space generated (or spanned) by its columns
- the maximal number of linearly independent columns
- the dimension of the vector space spanned by its rows



Norm

- 1-norm
- 2-norm
- Infinity-norm
- P-norms
- Frobenius norm



Linear system $Ax = b$ can be solved by factorizing the matrix A.

- Decompose A as $A = BC$, where B and C are matrices such that these two systems are easy to solve.
- Reduce the problem to solving $By = b$ and $Cx = y$.
- Examples of easy-to-solve matrices: diagonal, triangular, orthogonal
- For overdetermined system of equations, solve the linear least squares problem min *1/2 ||Ax − b||_2^2*.

Hence, matrix decomposition!

LU Decomposition

$$A = LU$$

where A is non-singular, L is lower-triangular, and U is upper triangular.

Pivoting

- Gaussian elimination is unstable without pivoting.
- Partial pivoting: PA = LU
  - Permute the rows of A using P, such that the largest entry of the first column is at the top of that first column.
  - Apply Gaussian elimination without pivoting to PA.
- Complete pivoting: PAQ = LU
- Rook pivoting

Cholesky Factorization

$$A = R^*R = LL^*$$

where R is upper triangular and L is lower triangular (*: conjugate transpose).

A needs to be a Hermitian, positive-definite matrix.

- Cholesky Factorization is a variant of Gaussian elimination (LU) that operations on both left and right of the matrix simultaneously.
- Cholesky decomposition uses symmetric Gaussian elimination.
- Every Hermitian positive definite A has a unique Cholesky factorization.

Hermitian matrix $A = A^*$

Symmetric, positive definite (SPD) matrix

- A symmetric matrix A is SPD iff all its eigenvalues are positive $\longrightarrow$ check by eigenvalue decomposition (expensive/difficult for large matrices)
- If a Cholesky decomposition can be successfully computed, the matrix is SPD $\longrightarrow$ check by Cholesky factorization (best option)

QR Factorization

$$A = QR, \qquad AE = QR$$

where Q is orthogonal ($QQ^T = I$), and R is upper triangular.

When is this useful?

- Pseudo-inverse
- Solution of least squares
- Solution of linear system of equations
- Extraction of orthogonal basis for column space of A

Full QR factorization

- Q: m x m, R: m x n

Economy QR (skinny QR) factorization

- Q: m x n, R: n x n

- **Least-Squares**: $min\|Ax - b\|_2 = min\|Rx - Q^T b\|_2$

Example: least-squares via QR / backslash

- Goal: solve min $||Ax - b||_2$ for an overdetermined system ($m > n$).
- Backslash uses a QR-based least-squares path for rectangular A.

```
% Fit y ≈ c1*x + c0
x = (0:0.1:1)';
A = [x ones(size(x))];
b = sin(2*pi*x);

c = A\b;
c

[Q,R] = qr(A,0);
c_qr = R\(Q'*b);
c_qr
```

```
c = 2x1
    -1.3989
     0.6995
```

```
c_qr = 2x1
    -1.3989
     0.6995
```

Eigenvalue Decomposition (EVD)

$$A = XDX^{-1}$$

where D is a diagonal matrix with the eigenvalues of A on the diagonal and the columns of X contain the eigenvectors of A.

- Diagonalizable (EVD exists) vs. defective (EVD does not exist)
- All EVD algorithms must be iterative
- Eigenvalue Decomposition algorithm: reduce to upper Hessenberg form and iteratively transform upper Hessenberg to upper triangular

Quiz

```
A = gallery('lehmer',4);
```

- Find the largest eigenvalue of the given matrix.
- Compute $\|A^3\|_\infty$ using EVD.
- Compute $\|e^A\|_1$ using EVD.

Solution

Quiz

```
A = gallery('lehmer',4);
```

- Find the largest eigenvalue of the given matrix.
- Compute $\|A^3\|_\infty$ using EVD.
- Compute $\|e^\wedge A\|_1$ using EVD.

Solution

```
[V,D] = eig(A); max(D)
norm(V*D^3/V,'inf')
norm(V*exp(1)^D/V,1)
```

Singular Value Decomposition (the most important matrix decomposition in numerical linear algebra)

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

where U and V are orthogonal and Σ is diagonal with real, positive entries.

Why do we care?

- Works for any matrix (square or rectangular, rank-deficient or not)
- Reveals the intrinsic rank and dimensionality of data
- Provides the best low-rank approximation (Eckart–Young theorem)
- Foundation of:
  - PCA and dimensionality reduction
  - Least-squares and ill-posed problems
  - Data compression, denoising, and regularization

Singular Value Decomposition (the most important matrix decomposition in numerical linear algebra)

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

where U and V are orthogonal and Σ is diagonal with real, positive entries.

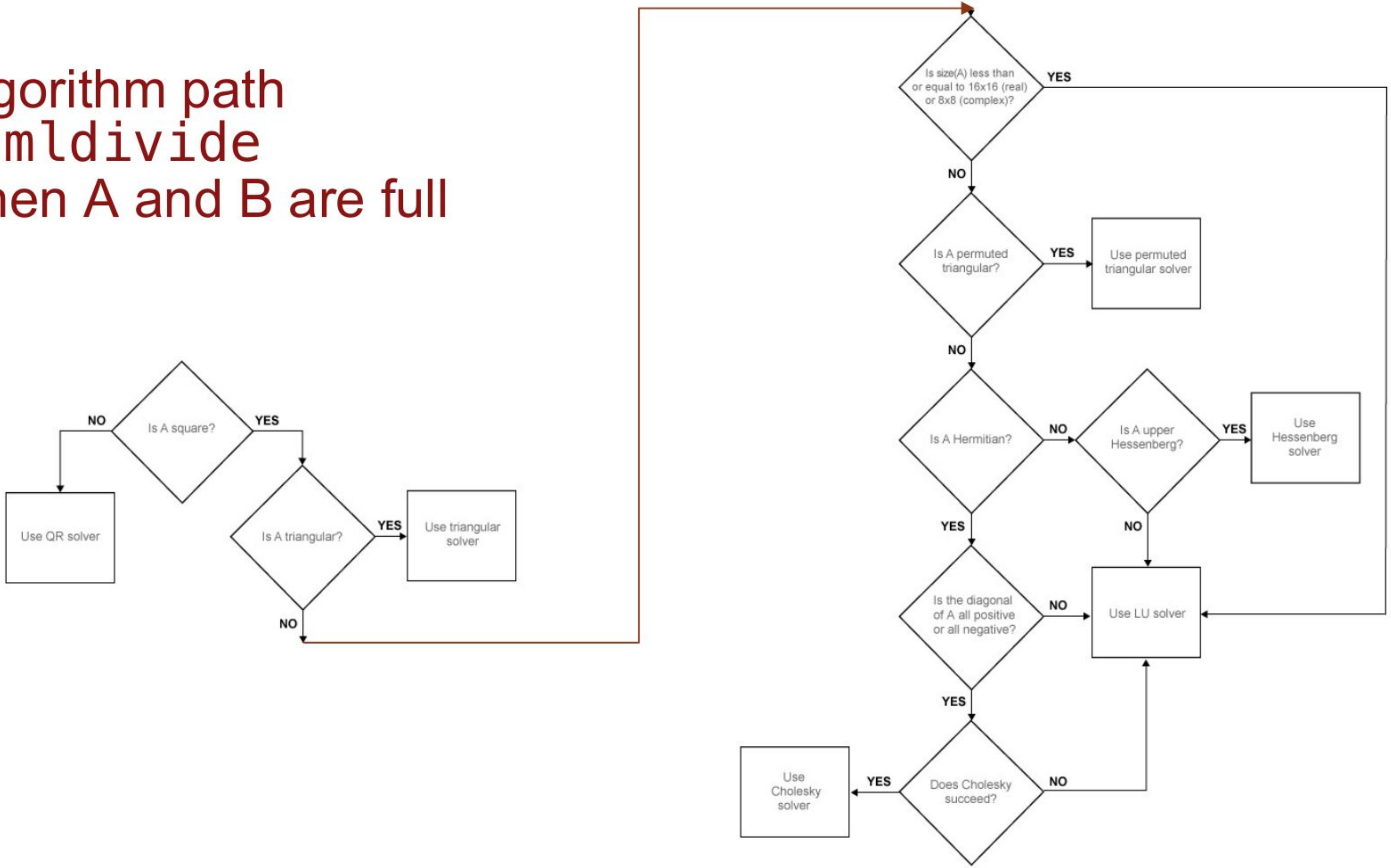
- SVD algorithm: Reduce A to bi-diagonal form and iteratively transform bi-diagonal to diagonal
- Full SVD: A_{mxn} = U_{mxm} Sigma_{mxn} V^T_{nxn}
    - Use when you need complete orthonormal bases
    - Useful for theoretical analysis and proofs
    - More expensive in memory and computation
- Reduced SVD: A_{mxn} = U_{mxr} Sigma_{rxr} V^T_{rxn}
    - R = rank(A)
    - Use for data analysis, PCA, compression
    - Faster, smaller, and almost always what you want in practice

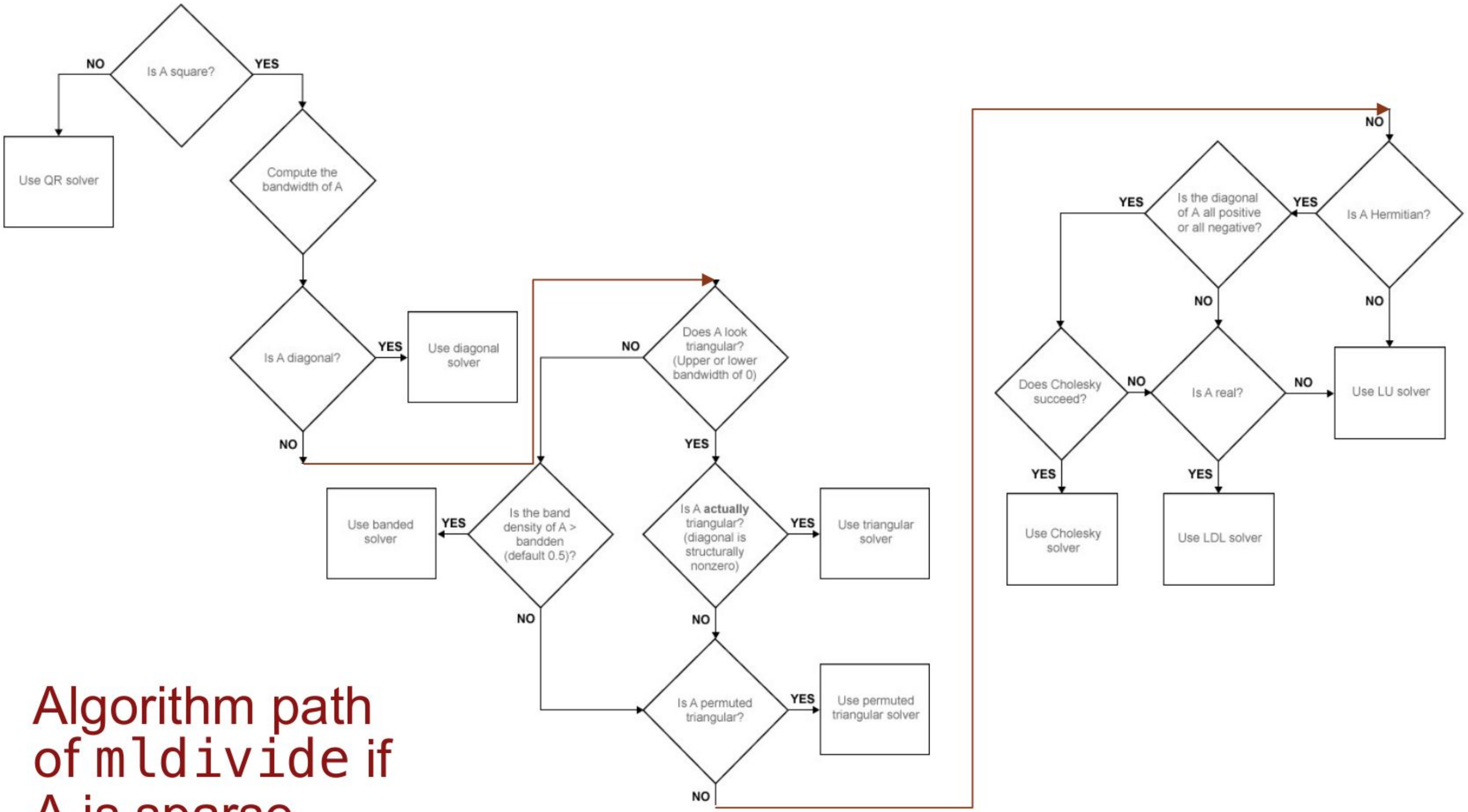
Direct Solvers for *Ax = b*: the Backslash

```
x = A\B
x = mldivide(A,B)
```

- This solves the system of linear equations A*x = B.
- The matrices A and B must have the same number of rows.
    - If A is a scalar, then A\B is equivalent to A.\B.
    - If A is a square n-by-n matrix and B is a matrix with n rows, then x = A\B is a solution to the equation A*x = B, if it exists.
    - If A is a rectangular m-by-n matrix with m ~= n, and B is a matrix with m rows, then A\B returns a least-squares solution to the system of equations A*x= B.
- **Use backslash rather than x = inv(A)* b**

Algorithm path of `mldivide` when A and B are full





Algorithm path of `mldivide` if A is sparse

Condition Number

A matrix is well-conditioned for condition number k close to 1;
ill-conditioned for condition number k large.

- `cond`: returns 2-norm condition number
- `condest`: lower bound for 1-norm condition number
- `rcond`: LAPACK estimate of inverse of 1-norm condition number

In Ax = b, if the condition number is large, even a small error in b may cause a large error in x.

Example: condition number sensitivity

- Small relative perturbation in b can cause larger relative change in x when cond(A) is large.

```
A = hilb(10
b = ones(10,1)
x = A\b
condA = cond(A)
db = 1e-8 * randn(size(b)
x2 = A\(b + db)
rel_b = norm(db)/norm(b)
rel_x = norm(x2 - x)/norm(x)
fprintf('Condition number of A: %.2e\n',
condA); fprintf('Relative perturbation in
b: %.2e\n', rel_b)
fprintf('Relative change in solution:
%.2e\n', rel_x)
fprintf('Amplification factor (~cond):
%.2e\n', rel_x / rel_b)
```

```
Condition number of A: 1.60e+13
Relative perturbation in b: 8.55e-09
Relative change in solution: 4.55e-03
Amplification factor (~cond): 5.33e+05
```

Iterative Solvers

- **Goal:** Solve large linear systems Ax=b approximately by iteratively improving the solution, without forming or factorizing A.
- **Who uses them:** Scientists and engineers in numerical PDEs, machine learning, optimization, physics simulations, and large-scale data analysis.
- **Why they matter:** They scale to massive sparse problems, use far less memory than direct solvers, and often converge quickly when combined with good preconditioners.

Iterative Solvers

Preconditioning

- Preconditioning replaces the original problem ($Ax = b$) with a different problems with the same (or similar) solution.
    - Left preconditioning: $L^{-1}Ax = L^{-1}b$
    - Right preconditioning: $y = Rx$, $AR^{-1}y = b$
    - Left and right preconditioning: $L^{-1}AR^{-1}y = L^{-1}b$
- Preconditioner M for A ideally provides a cheap approximation to A^{-1}, intended to drive condition number toward 1.
- Typical preconditioners:
    - Jacobi: $M = \mathrm{diag}(\mathrm{diag}(A))$
    - Incomplete factorizations: LU, Cholesky (control for level of fill-in)

Common Iterative Solvers

Linear system of equations $Ax = b$

- Symmetric Positive Definite matrix: Conjugate Gradients (CG)
- Symmetric matrix: Symmetric LQ Method (SYMMLQ), Minimum-Residual (MINRES)
- General, Unsymmetric matrix: Biconjugate Gradients (BiCG), Biconjugate Gradients Stabilized (BiCGstab), Conjugate Gradients Squared (CGS), Generalized Minimum-Residual (GMRES)

Linear least-squares $min\|Ax = b\|_2$

- Least-Squares Minimum-Residual (LSMR)
- Least-Squares QR (LSQR)

Example: preconditioned CG (pcg) in MATLAB

- Constructs a sparse SPD matrix from a 2D Laplacian, then solves $A\,x = b$ Ax=b using the preconditioned conjugate gradient method. The incomplete Cholesky factor serves as a preconditioner to significantly accelerate convergence compared to unpreconditioned CG.

```
n = 1000
m = round(sqrt(n)) + 2
A = delsq(numgrid('S', m))
b = ones(size(A,1),1)
L = ichol(A, struct('diagcomp',1e-2))
[x,flag,relres,iter,resvec] = pcg(A,b,1e-8,200,L,L')
fprintf('size(A)=%d, condest(A)=%.2e\n', size(A,1),
condest(A))
fprintf('pcg: flag=%d, iter=%d, relres=%.2e\n', flag, iter,
relres)
```

```
size(A)=1024, condest(A)=6.40e+02
pcg: flag=0, iter=30, relres=3.82e-09
```

MATLAB's built-in iterative solvers for $Ax = b$, $A \in R^{m\times m}$

```
[x,flag,relres,iter,resvec] =
solver(A,b,restart,tol,maxit,M1,M2,x0)
```

Inputs (only A, b required):

- `A` – full or sparse (recommended) square matrix or function handle returning Av for $v \in R^m$
- `b` – m vector
- `restart` – restart frequency (GMRES)
- `tol` – relative convergence tolerance
- `maxit` – maximum number of iterations
- `M1`, `M2` – full or sparse (recommended) preconditioner matrix or function handle returning $M2^{-1}M1^{-1}v$ for any $v \in R^m$
- `x0` – initial guess of solution

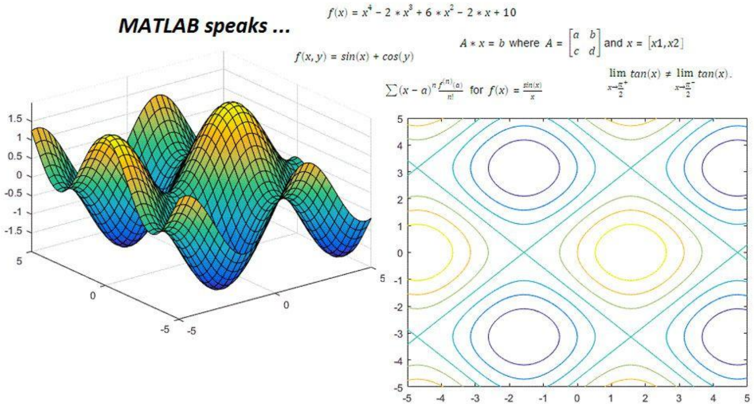
Outputs:

```
[x,flag,relres,iter,resvec] =
solver(A,b,restart,tol,maxit,M1,M2,x0)
```

- `x` – attempted solution to Ax = b
- `flag` – convergence flag
- `relres` – relative residual $\|b-Ax\|/\|b\|$ at convergence
- `iter` – number of iterations (inner and outer iterations for certain algorithms)
- `resvec` – vector of residual norms at each iteration $\|b - Ax\|$, including preconditioners if used ($\|M^{-1}(b - Ax)\|$).

Symbolic Math

Applied Math with MATLAB



Symbolic Math Toolbox provides functions for solving, plotting, and manipulating symbolic math equations.

- analytically perform differentiation, integration, simplification,
- transforms, and equation solving
- perform dimensional computations and convert between units
- display computation results in mathematical typeset
- convert work to HTML, Word, LaTex, or PDF documents

Operations and Commands

Symbolic arithmetic operations

- ceil, cong, cumprod, cumsum, fix, floor, frac, imag,
- minus, mod, plus, quorem, real, round

Symbolic relational operations

- eq, ge, gt, le, lt, ne, isequaln

Symbolic logical operations

- and, not, or, xor, all, any, isequaln, isfinite, isinf,
- isnan, logical

Equation Solving

- `finverse` Functional inverse
- `linsolve` Solve linear system of equations
- `poles` Poles of expression/function
- `solve` Equation/System of equations solver
- `dsolve` ODE solver

Formula Manipulation and Simplification

- `simplify` Algebraic simplification
- `simplifyFraction` Symbolic simplification of fractions
- `subexpr` Rewrite symbolic expression in terms of common subexpression
- `subs` Symbolic substitution

Calculus

- `diff` Differentiate symbolic
- `int` Deffnite and indefinite integrals
- `rsums` Riemann sums
- `curl` Curl of vector field
- `divergence` Divergence of vector field
- `gradient` Gradient vector of scalar function
- `hessian` Hessian matrix of scalar function
- `jacobian` Jacobian matrix
- `laplacian` Laplacian of scalar function
  potential Potential of vector field
- `vectorPotential` Vector potential of vector field
- `taylor` Taylor series expansion
- `limit` Compute limit of symbolic
- expression
- `fourier` Fourier transform
- `ifourier` Inverse Fourier transform
- `ilaplace` Inverse Laplace transform
- `iztrans` Inverse Z-transform
- `laplace` Laplace transform
- `ztrans` Z-transform

$$\int_0^\infty e^{-ax^2}\cos(bx)\,dx = \frac{\sqrt{\pi}}{2\sqrt{a}}\,e^{-b^2/(4a)}, \quad a > 0.$$

```
syms x a b positive real
I = int(exp(-a*x^2)*cos(b*x), x, 0, inf);
I_simplified = simplify(I)
disp('Integral result:')
pretty(I_simplified)

a_val = 2
b_val = 3;
I_num_sym = double(subs(I_simplified, [a b],
[a_val b_val]))
I_num_quad = integral(@(t)
exp(-a_val*t.^2).*cos(b_val*t), 0, Inf)

fprintf('Symbolic: %.15f\n', I_num_sym)
fprintf('Numeric : %.15f\n', I_num_quad)
fprintf('Abs diff: %.3e\n', abs(I_num_sym -
I_num_quad))
```

```
Integral result:
               /     2 \
               |    b  |
sqrt(pi) exp| - --- |
               \   4 a /
---------------------
      2 sqrt(a)
Symbolic: 0.203445763527289
Numeric : 0.203445763527288
Abs diff: 1.027e-15
```

Linear Algebra

Most matrix operations available for numeric arrays also available for symbolic matrices.

- `adjoint` Adjoint of symbolic square matrix
- `expm` Matrix exponential
- `sqrtm` Matrix square root
- `cond` Condition number of symbolic matrix
- `det` Compute determinant of symbolic matrix
- `norm` Norm of matrix or vector
- `colspace` Column space of matrix
- `null` Form basis for null space of matrix

Linear Algebra

- `rank` Compute rank of symbolic matrix
- `rref` Compute reduced row echelon form
- `eig` Symbolic eigenvalue decomposition
- `jordan` Jordan form of symbolic matrix
- `chol` Symbolic Cholesky decomposition
- `lu` Symbolic LU decomposition
- `qr` Symbolic QR decomposition
- `svd` Symbolic singular value decomposition
- `inv` Compute symbolic matrix inverse
- `linsolve` Solve linear system of equations

Assumptions

- `assume` Set assumption on symbolic object
- `assumeAlso` Add assumption on symbolic object
- `assumptions` Show assumptions set on symbolic variable

Polynomials

- `charpoly` Characteristic polynomial of matrix
- `coeffs` Coefficients of polynomial
- `minpoly` Minimal polynomial of matrix
- `poly2sm` Symbolic polynomial from coefficients
- `sym2poly` Symbolic polynomial to numeric

$$\int_0^\infty \frac{x^{s-1}}{1+x}\,dx = \pi\csc(\pi s), \quad 0 < s < 1.$$

```
syms x s real
assumeAlso(s > 0 & s < 1)

I = int(x^(s-1)/(1+x), x, 0, inf);
I_simplified = simplify(I);

disp('Integral result:')
pretty(I_simplified)

s_val = 0.37;
I_num_sym = double(subs(I_simplified, s, s_val));
I_num_quad = integral(@(t) t.^(s_val-1)./(1+t),
0, Inf);

fprintf('Symbolic: %.15f\n', I_num_sym);
fprintf('Numeric : %.15f\n', I_num_quad);
fprintf('Abs diff: %.3e\n', abs(I_num_sym -
I_num_quad));
```

```
Integral result:
    pi
---------
sin(pi s)
Symbolic: 3.423129195615263
Numeric : 3.423128249625757
Abs diff: 9.460e-07
```

Mathematical Functions

- `log`, `log10`, `log2` Logarithmic functions
- `sin`, `cos`, `tan`, etc Trigonometric functions
- `sinh`, `cosh` `tanh`, etc Hyperbolic functions

Precision Control

- `digits` Variable-precision accuracy
- `double` Convert symbolic expression to MATLAB double
- `vpa` Variable precision arithmetic

Code generation

- `ccode` C code representation of symbolic expression
- `fortran` Fortran representation of symbolic expression
- `latex` LATEX representation of symbolic expression
- `matlabFunction` Convert symbolic expression to function handle or file
- `texlabel` TeX representation of symbolic expression