Welcome back to **CME 292**

Advanced MATLAB for Scientific Computing

WINTER 2023

Stanford University

Applied Math with MATLAB

CME 292 Lecture 5

1/24/2023

Stanford University

Outline

Numerical Linear Algebra

- Sparse matrices
- Matrix decomposition
- Linear system solvers

Numerical Optimization

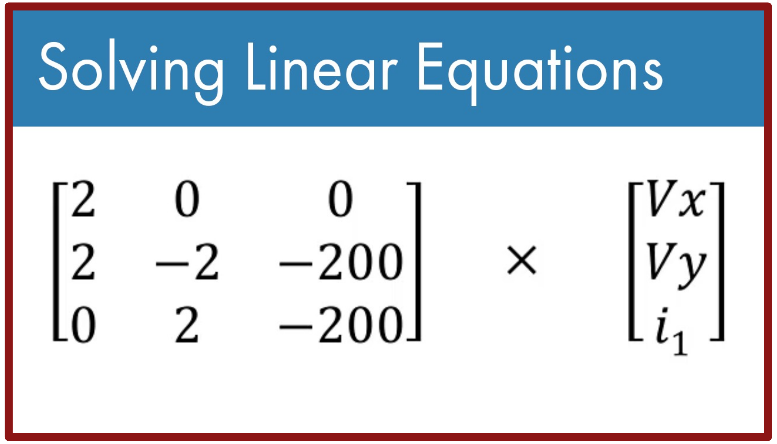
- Optimization problem
- Nonlinear system of equations
- Optimization solvers

Symbolic Math

ODE and PDE

Numerical Linear Algebra

Applied Math with MATLAB



Solving Linear Equations

$$\begin{bmatrix} 2 & 0 & 0 \\ 2 & -2 & -200 \\ 0 & 2 & -200 \end{bmatrix} \times \begin{bmatrix} Vx \\ Vy \\ i_1 \end{bmatrix}$$

Image from MathWorks

Stanford University

Dense vs. Sparse Matrices

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

$$A = \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}$$

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

- Dense storage requires mn entries.
- Sparse format saves storage.

Sparse Matrix Storage Formats

1. Triplet format
    - Store nonzero values and corresponding row/column
    - Storage required = $3n_z$ ($2n_z$ ints and n_z doubles)
    - General in that no assumptions are made about sparsity structure
    - Used by MATLAB (column-wise)

$$\begin{bmatrix} 1 & 9 & 0 & 0 & 1 \\ 8 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 5 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{bmatrix}$$

$$\texttt{row} = [1 \quad 2 \quad 1 \quad 2 \quad 5 \quad 3 \quad 3 \quad 4 \quad 1 \quad 5]$$

$$\texttt{col} = [1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 4 \quad 4 \quad 5 \quad 5]$$

$$\texttt{val} = [1 \quad 8 \quad 9 \quad 2 \quad 4 \quad 3 \quad 5 \quad 7 \quad 1 \quad 1]$$

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 = $2n_z + m$
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 = $2n_z + n$
4. Others
    - Diagonal Storage format (useful for banded matrices)
    - Skyline Storage format
    - Block Compressed Sparse Row (BSR) format

Break-Even Point for Sparse Storage

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

- For the triplet format, the cross-over point is defined by
$$3n_z = mn,$$
- If $n_z \leq \frac{mn}{3}$ use sparse storage, otherwise use dense format.
- Cross-over point depends not only on m, n, n_z, 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 $n \times n$ 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.

Solution

full matrix: $1000 \times 1000 = 10^6$

triplet: $n_z = 2998, \ 3n_z = 8994$

For a tridiagonal matrix, $n_z = 3n - 2$. Solving $3n_z = n^2$ gives $n \approx 8.275$.

Create Sparse Matrices

- Allocate space for $m{\times}n$ sparse matrix with n_z nonzeros:
  `S = spalloc(m; n; nz)`
- Convert full matrix A to sparse matrix S:
  `S = sparse(A)`
- Create $m{\times}n$ sparse matrix with spare for n_z nonzeros from triplet `(row,col,val)`:
  `S = sparse(row,col,val,m,n,nz)`
- Create matrix of 1s with sparsity structure defined by sparse matrix S:
  `R = spones(S)`
- Sparse identity matrix of size $m{\times}n$:
  `I = speye(m,n)`
- Create sparse uniformly distributed random matrix from sparsity structure of sparse matrix S:
  `R = sprand(S)`

Stanford University

- Create sparse uniformly distributed random matrix of size $m \times n$ with approximately $mn\rho$ 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

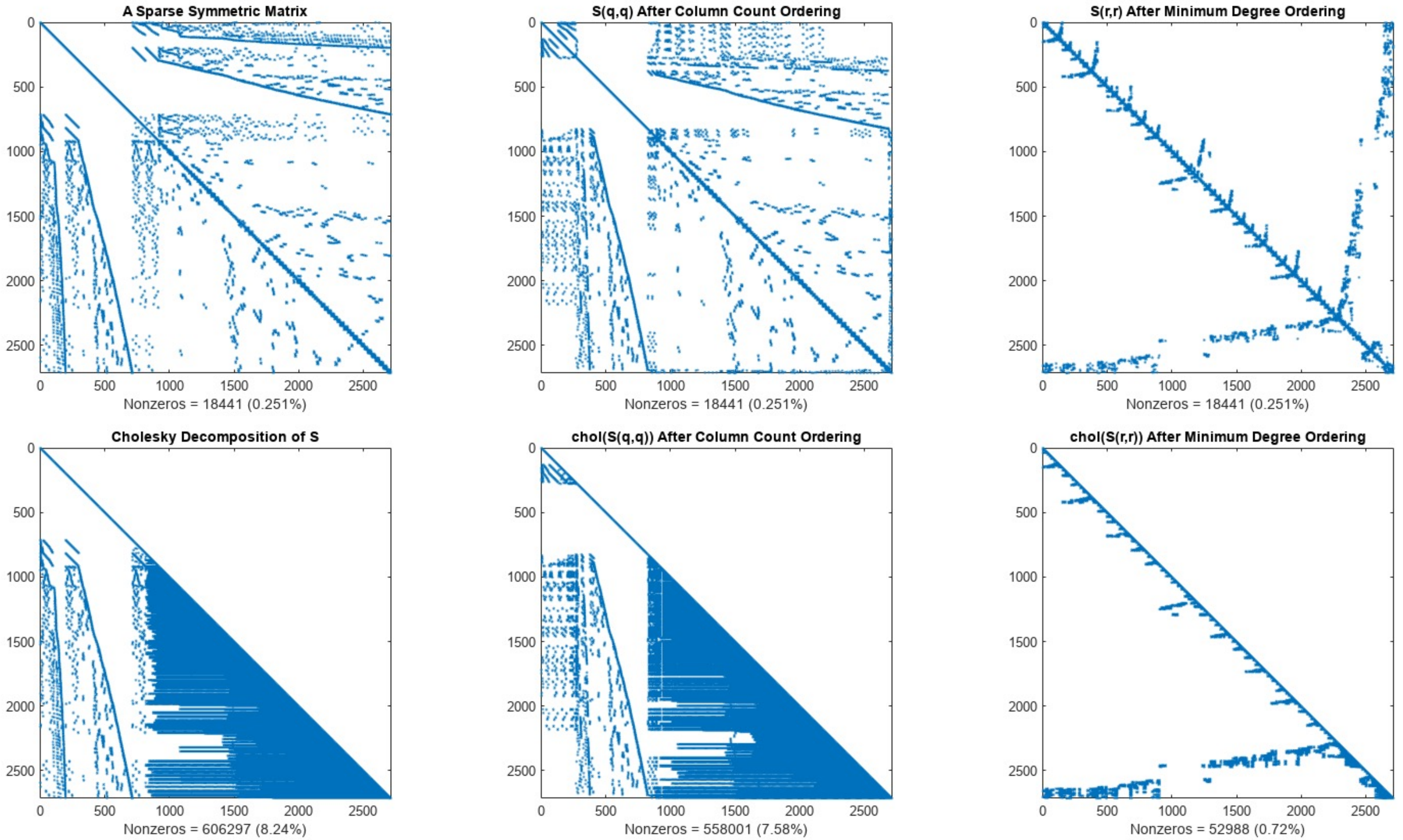
- Convert sparse matrix to dense matrix: `A = full(S)`
- Apply function (described by function handle func) to nonzero elements of sparse matrix: `F = spfun(func, S)`
- Plot sparsity structure of matrix: `spy(S)`

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
Nonzeros = 18441 (0.251%)
S(q,q) After Column Count Ordering
Nonzeros = 18441 (0.251%)
S(r,r) After Minimum Degree Ordering
Nonzeros = 18441 (0.251%)
Cholesky Decomposition of S
Nonzeros = 606297 (8.24%)
chol(S(q,q)) After Column Count Ordering
Nonzeros = 558001 (7.58%)
chol(S(r,r)) After Minimum Degree Ordering
Nonzeros = 52988 (0.72%)

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

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 $\min \frac{1}{2} ||Ax - b||_2^2$.

→ 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 → check by eigenvalue decomposition (expensive/difficult for large matrices)
- If a Cholesky decomposition can be successfully computed, the matrix is SPD → 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.

- useful in:
  - 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: $\min\|Ax - b\|_2 = \min\|Rx - Q^T b\|_2$

Eigenvalue Decomposition (EVD)

$$A = X\mathrm{D}X^{-1}$$

where A is 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.

Stanford University

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

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

Singular Value Decomposition

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

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

- SVD algorithm: bi-diagonalize A and iteratively transform bi-diagonal to diagonal

- Full SVD
- Reduced SVD

Direct Solvers for $Ax = b$: the Backslash

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

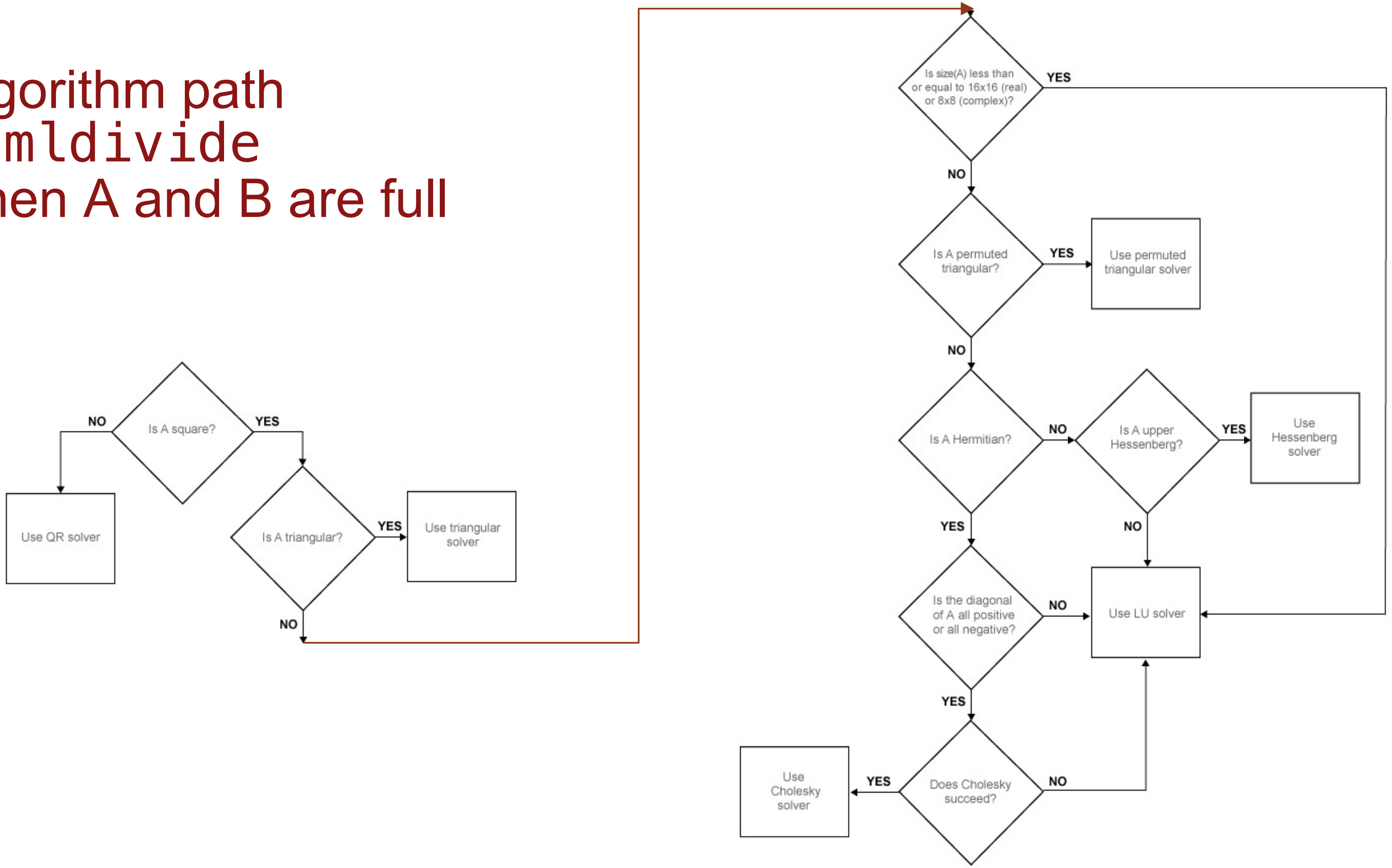
- This solves the system of linear equations A*x = B.
- The matrices A and B must have the same number of rows.

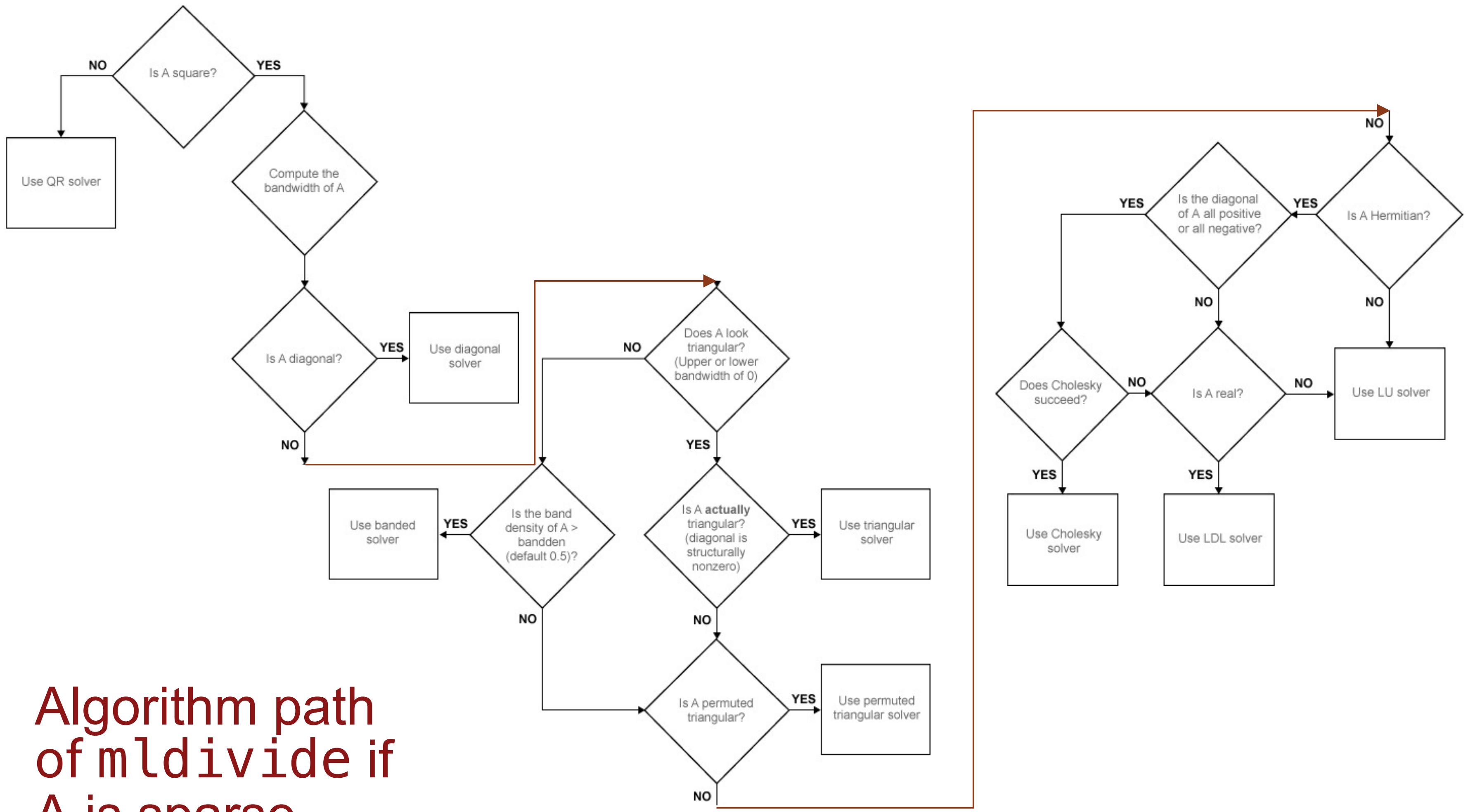
  - If A is a scalar, then A\B is equivalent to A.\B.
  - If A is a square n-by-n matrix and B is a matrix with n rows, then x = A\B is a solution to the equation A*x = B, if it exists.
  - If A is a rectangular m-by-n matrix with m ~= n, and B is a matrix with m rows, then A\B returns a least-squares solution to the system of equations A*x= B.

- **Use backslash rather than `x = inv(A)* b`**

Stanford University

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




NO
Is A square?
YES
Use QR solver
Compute the bandwidth of A
Is A diagonal?
YES
Use diagonal solver
NO
NO
Does A look triangular? (Upper or lower bandwidth of 0)
YES
Use banded solver
YES
Is the band density of A > bandden (default 0.5)?
NO
Is A **actually** triangular? (diagonal is structurally nonzero)
YES
Use triangular solver
NO
Is A permuted triangular?
YES
Use permuted triangular solver
NO
NO
Is A Hermitian?
YES
Is the diagonal of A all positive or all negative?
YES
NO
NO
Does Cholesky succeed?
NO
Is A real?
NO
Use LU solver
YES
YES
Use Cholesky solver
Use LDL solver

Algorithm path of `mldivide` if A is sparse

Condition Number κ

A matrix is well-conditioned for κ close to 1; ill-conditioned for κ 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*.

→☹

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 = diag(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)

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 $M_2^{-1} M_1^{-1} v$ for any $v \in R^m$
- `x0` – initial guess of solution

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

Outputs:

- `x` – attempted solution to $Ax = b$
- `flag` – convergence flag
- `relres` – relative residual $\frac{||b-Ax||}{||b||}$ 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 ($||M^{-1}(b - Ax)||$).

Numerical Optimization

Applied Math with MATLAB

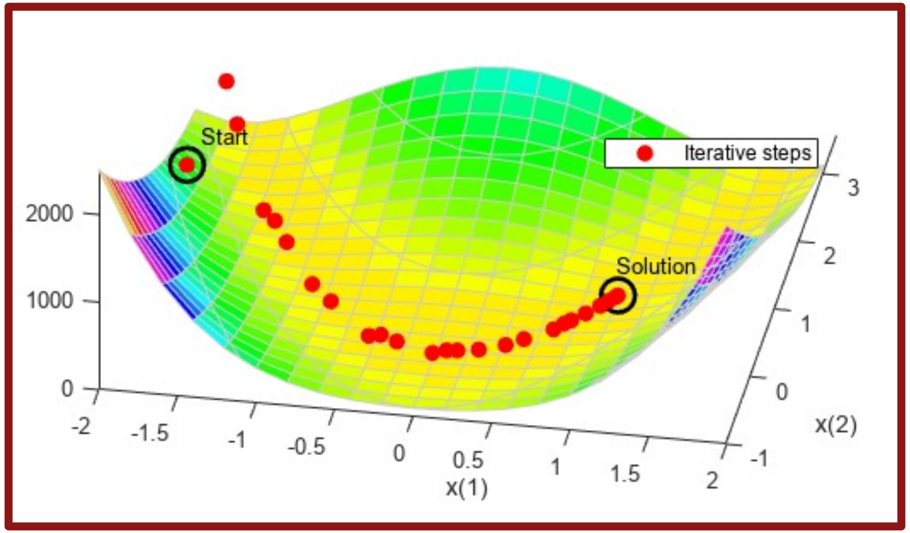


Image from MathWorks

Stanford University

Function Derivatives

	Jacobian	Gradient	Hessian
Scalar-valued function	$\frac{\partial}{\partial x} f(x)$	$\nabla f(x) = \frac{\partial}{\partial x} f(x)^T$	$[\nabla f^2(x)]_{ij} = \frac{\partial f^2}{\partial x_i \partial x_j}(x)$
Vector-valued function	$\left[\frac{\partial}{\partial x} F(x)\right]_{\text{ij}} = \frac{\partial}{\partial x_j} F_i(x)$	$\nabla F = \left(\frac{\partial}{\partial x} F(x)\right)^T$	$[\nabla^2 F(x)]_{ijk} = \frac{\partial F_i^2}{\partial x_j \partial x_k}(x)$

General Optimization Problem

Objective
$$\boxed{\underset{x \in R^{n_v}}{\text{mininize}}\, f(x)}$$

Constraints

$$\begin{aligned} \text{subject to} \quad Ax &\le b && \rightarrow \text{Linear inequality constraints} \\ A_{eq}x &= b_{eq} && \rightarrow \text{Linear equality constraints} \\ c(x) &\le 0 && \rightarrow \text{Nonlinear inequality constraints} \\ c_{eq}(x) &= 0 && \rightarrow \text{Nonlinear equality constraints} \\ l \le x &\le u && \rightarrow \text{box constraints} \end{aligned}$$

Lagrangian and Karush-Kuhn-Tucker (KKT) optimality conditions

$$\underset{x \in R^{n_v}}{\text{mininize}}\, f(x)$$

$$\text{subject to} \quad g(x) \leq 0$$
$$h(x) = 0$$

Lagrangian: $L(x, \lambda, \mu) = f(x) + \lambda^T g(x) + \mu^T h(x)$

$\nabla_x L(x^*, \lambda^*, \mu^*) = 0$ → Stationarity

$g(x^*) \leq 0$
$h(x^*) = 0$ → Primal feasibility

$\lambda^* \geq 0$ → Dual feasibility

${\lambda^*}^T g(x^*) = 0$ → Complementary slackness

Nonlinear System of Equations

Find $x \in R^n$ such that

$$F(x) = 0$$

where $F: R^n \to R^m$ is continuously differentiable, nonlinear function.

- Solution methods are iterative, in general, which require initial guess and convergence criteria.
- Solution is not guaranteed to exist.
- If the solution, it is not necessarily unique. The solution found depends heavily on the initial guess.

$n = m = 1$ case

Derivative-free methods

- Only require evaluations of f(x).
- Bisection, Fixed point iteration, etc.
- `fzero` (for scalar functions)
  - combining bisection, secant, and interpolation methods)
  - `[x,fval,exitflag,output] = fzero(fun,x0,options)`
  - a MATLAB builtin command for finding a root of a continuous, scalar-valued, univariate function

Gradient-based methods

- Requires function, f(x), and gradient, f'(x), evaluations.
- Newton's method: $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$

Secant method: use finite differences to approximate gradients instead of analytic gradients (only requires function evaluations)

Quiz

```
[x,fval,exitflag,output] = fzero(fun,x0,options)
```

Try solving $x^2 = 0$ and $x^2 + 1 = 0$ using `fzero`. What do you get? How would you explain the results?

Quiz

```
[x,fval,exitflag,output] = fzero(fun,x0,options)
```

Try solving $x^2 = 0$ and $x^2 + 1 = 0$ using `fzero`. What do you get? How would you explain the results?

Solution

```
[x2,fval2,exitflag2,output2] = fzero(@(x) x^2,4);
[x3,fval3,exitflag3,output3] = fzero(@(x) x^2+1,4);
```

`fzero` first finds an interval containing `X0` where the function values of the interval endpoints differ in sign, then searches that interval for a zero.

General case

Derivative-free methods

- Requires function, F(x), evaluations
- Fixed point iteration, Secant method, etc

Gradient-based methods

- Requires function and Jacobian evaluations
- Newton-Raphson method
- Gauss-Newton and Levenberg-Marquardt (nonlinear least squares)
- Can use finite difference approximations to gradients instead of analytic gradients (only requires function evaluations)

`fsolve` (for vector-valued functions)

- Gradient-based
- `[x,fval,exitflag,output,jac] = fsolve(fun,x0,options)`
- Algorithms: standard trust region (default), trust region reflexive, Gauss-Newton, Levenberg-Marquardt

Types of Optimization Solvers

Derivative-free (only function evaluations)

- Brute force
- Genetic algorithms
- Finite difference computations of gradients/Hessians
- Usually require very large number of function evaluations

Gradient-based (most popular, function and gradient evaluations)

- Finite difference computations of Hessians
- SPD updates to build Hessian approximations (BFGS)

Hessian-based (function, gradient, and Hessian evaluations)

- Hessians is usually difficult/expensive to compute, although often very sparse.
- Second-order optimality conditions

Interior point methods

- Iterates always strictly feasible
- Use barrier functions to keep iterates away from boundaries
- Sequence of optimization problems

Active set methods

- Active set refers to the inequality constraints active at the solution.
- There are possibly infeasible iterates when constraints are nonlinear.
- Minimize problem for given active set, then update active set based on Lagrange multipliers

Globalization:

- Techniques to make optimization solver globally convergent (convergent to some local minima from any starting point)
- Trust region methods, line search methods

MATLAB Solvers

- Linear Program (LP):
  `linprog`
- Binary Integer Linear Program:
  `bintprog`
- Integer Linear Program:
  `intlinprog`
- Quadratic Program (QP):
  `quadprog`

$$\begin{aligned} &\text{minimize}_{x \in R^{n_v}} \; f^T x \\ &\text{subject to } \mathrm{Ax} \le b \\ &A_{\text{eq}} x = b_{\text{eq}} \\ &l \le x \le u \end{aligned}$$

LP

$$\begin{aligned} \text{minimize}_{x \in R^{n_v}} \; & \frac{1}{2} x^T \mathrm{Hx} + f^T x \\ \text{subject to} \quad \mathrm{Ax} &\le b \\ A_{\text{eq}} x &= b_{\text{eq}} \\ l \le x &\le u \end{aligned}$$

QP

- Unconstrained optimization:
  `fminsearch, fminunc`

$$\text{minimize}_{x\in R^{n_v}} f(x)$$

- Linearly constrained optimization:
  `fminbnd, fmincon`

$$\begin{aligned}&\text{minimize}_{x\in R^{n_v}} f(x)\\ &\text{subject to } \mathrm{Ax} \le b\\ &\qquad A_{\mathrm{eq}}x=b_{\mathrm{eq}}\\ &\qquad l \le x \le u\end{aligned}$$

- Nonlinear constrained optimization:
  `fmincon, fseminf, fgoalattain, fminimax`

Call to Optimization Solver

```
[x,fval,exitflag,out,lam,grad,hess] =
solver(f,x0,A,b,Aeq,beq,lb,ub,nlcon,opt)
[x,fval,exitflag,out,lam,grad,hess] = solver(problem)
```

Instead input a `problem` **structure** with fields

Inputs

- `f` – function handle (or vector for LP) defining objective function (and gradient)
- `x0` – vector defining initial guess
- `A,b` – matrix, RHS defining linear inequality constraints
- `Aeq, beq` – matrix, RHS defining linear equality constraints
- `lb,ub` – vectors defining lower, upper bounds
- `nlcon` – function handle defining nonlinear contraints (and Jacobians)
- `opt` – optimization options structure
- `problem` – structure containing above information

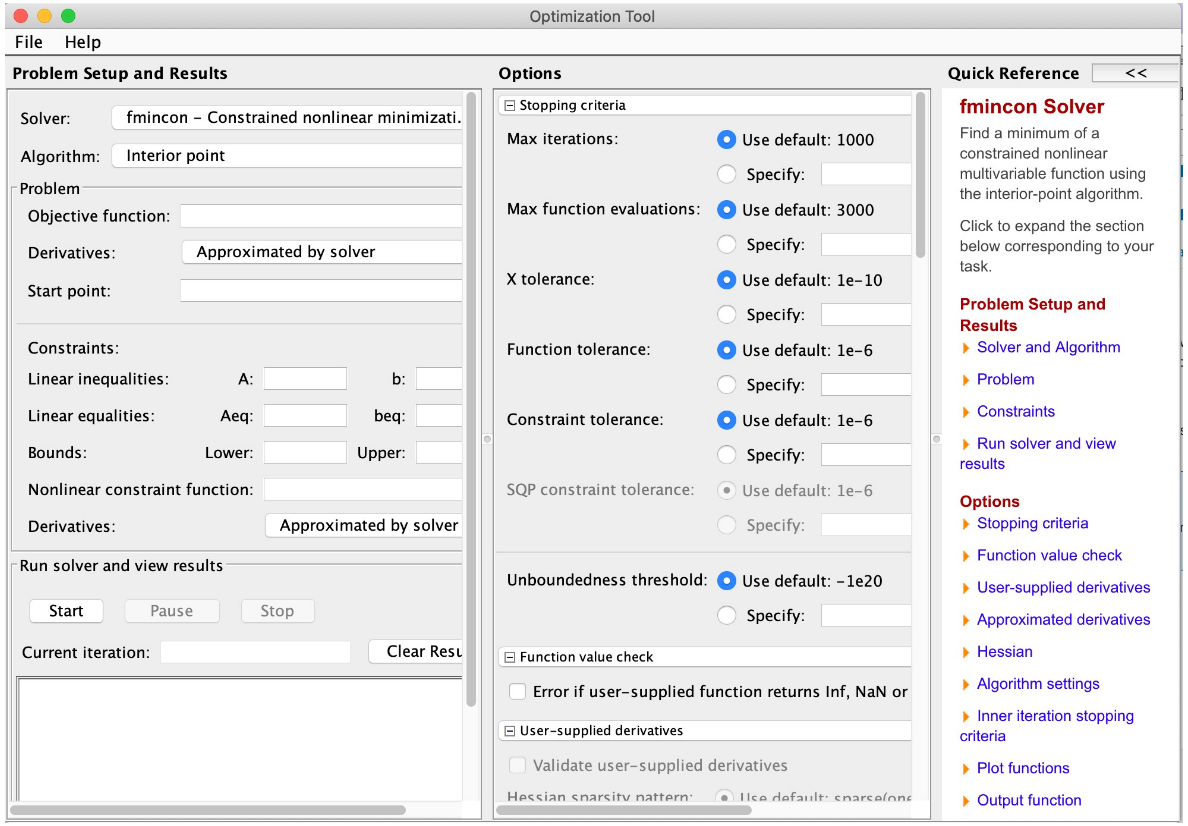
```
[x,fval,exitflag,out,lam,grad,hess] =
solver(f,x0,A,b,Aeq,beq,lb,ub,nlcon,opt)
```

Outputs

- `x` – minimum found
- `fval` – value of objective function at x
- `exitflag` – describes exit condition of solver
- `out` – structure containing output information
- `lam` – structure containing Lagrange multipliers at x
- `grad` – gradient of objective function at x
- `hess` – Hessian of objective function at x

Optimization Toolbox GUI

`optimtool`


Optimization Tool

File Help

Problem Setup and Results

Solver: fmincon - Constrained nonlinear minimizati.

Algorithm: Interior point

Problem

Objective function:

Derivatives: Approximated by solver

Start point:

Constraints:

Linear inequalities: A: b:

Linear equalities: Aeq: beq:

Bounds: Lower: Upper:

Nonlinear constraint function:

Derivatives: Approximated by solver

Run solver and view results

Start Pause Stop

Current iteration: Clear Resu

Options

Stopping criteria

Max iterations:	[✓ Use default: 1000] Specify:
Max function evaluations:	[✓ Use default: 3000] Specify:
X tolerance:	[✓ Use default: 1e-10] Specify:
Function tolerance:	[✓ Use default: 1e-6] Specify:
Constraint tolerance:	[✓ Use default: 1e-6] Specify:
SQP constraint tolerance:	[✓ Use default: 1e-6] Specify:
Unboundedness threshold:	[✓ Use default: -1e20] Specify:

Function value check

Error if user-supplied function returns Inf, NaN or

User-supplied derivatives

Validate user-supplied derivatives

Hessian sparsity pattern: Use default: sparse(one

Quick Reference <<

fmincon Solver

Find a minimum of a constrained nonlinear multivariable function using the interior-point algorithm.

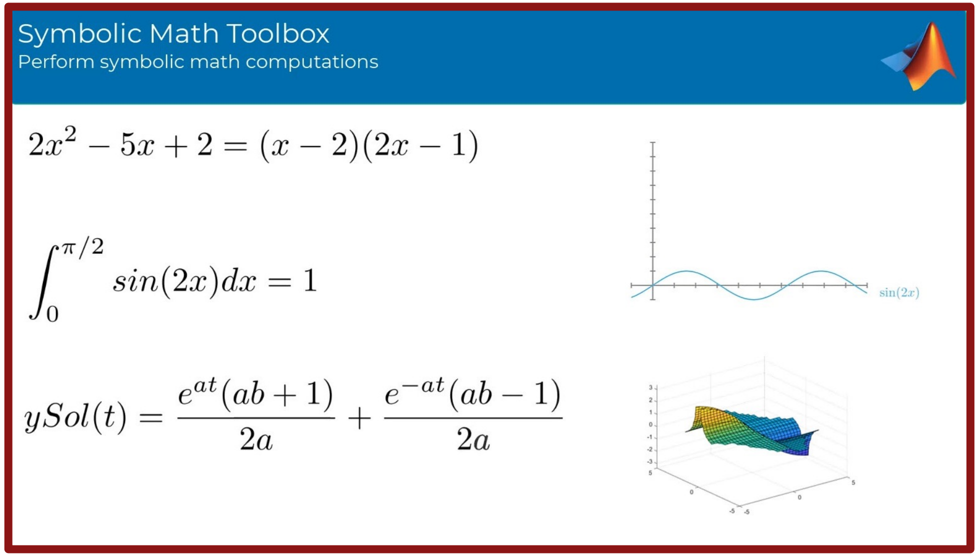
Click to expand the section below corresponding to your task.

Problem Setup and Results
- Solver and Algorithm
- Problem
- Constraints
- Run solver and view results

Options
- Stopping criteria
- Function value check
- User-supplied derivatives
- Approximated derivatives
- Hessian
- Algorithm settings
- Inner iteration stopping criteria
- Plot functions
- Output function

Symbolic Math

Applied Math with MATLAB



Symbolic Math Toolbox

Perform symbolic math computations

$$2x^2 - 5x + 2 = (x-2)(2x-1)$$

$$\int_0^{\pi/2} sin(2x)dx = 1$$

$$ySol(t) = \frac{e^{at}(ab+1)}{2a} + \frac{e^{-at}(ab-1)}{2a}$$

Image from MathWorks

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

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
- `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` Coeffcients of polynomial
- `minpoly` Minimal polynomial of matrix
- `poly2sm` Symbolic polynomial from coeffcients
- `sym2poly` Symbolic polynomial to numeric

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

ODE and PDE

Applied Math with MATLAB

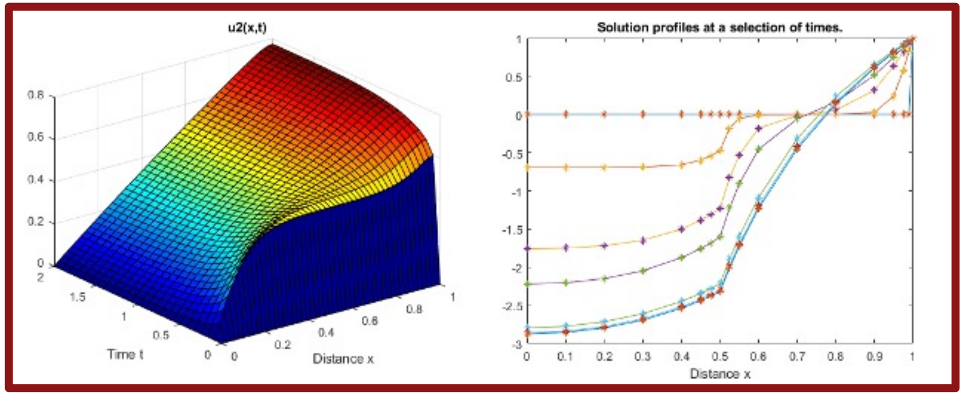


Image from MathWorks

Stanford University

Ordinary Differential Equation (ODE)

A system of ODEs can be written in the form

$$\frac{dy}{dt}(t) = F(t, y)$$
$$y(0) = y_0$$

- An ODE problem is stiff
  - If the solution being sought is varying slowly, but there are nearby solutions that vary rapidly.
  - A numerical method must take small steps to obtain satisfactory results.

- Various types of ODE solvers
  - Multi- vs single-stage
  - Multi- vs single-step (Number of time steps used approximate time derivative)
  - Implicit vs. Explicit (Trade-off between ease of advancing a single step versus number of steps required. Implicit schemes usually require solving a system of equations.)
  - Serial vs. Parallel

MATLAB ODE Solvers

```
[TOUT,YOUT] = ode_solver(ODEFUN,TSPAN,Y0)
```

- `ODEFUN` is a function handle.
  - For a scalar T and a vector Y, `ODEFUN(T,Y)` must return a column vector corresponding to f(t,y)
- `TSPAN = [T0 TFINAL]`
- `Y0` is the initial conditions
- Each row in the solution array `YOUT` corresponds to a time returned in the column vector `TOUT`.

MATLAB ODE Solvers

Options for `ode_solver`, its stiffness, and accuracy:

- `ode45` Non-stiff, Medium
- `ode23` Non-stiff, Low
- `ode113` Non-stiff, Low - High
- `ode15s` Stiff, Low - Medium
- `ode23s` Stiff, Low
- `ode23t` Moderately stiff, Low
- `ode23tb` Stiff, Low

Fourth-Order Explicit Runge-Kutta (ERK4)

A multi-stage, single-step, explicit, serial ODE solver.

- Discretize of the time domain into N+1 intervals.
- Fourth-order accuracy: error $\mathcal{O}(\Delta t^4)$

$$
\begin{aligned}
\mathbf{k}_1 &= \mathbf{F}(t_n, \mathbf{y}_n) \\
\mathbf{k}_2 &= \mathbf{F}(t_n + 0.5\Delta t, \mathbf{y}_n + 0.5\Delta t \mathbf{k}_1) \\
\mathbf{k}_3 &= \mathbf{F}(t_n + 0.5\Delta t, \mathbf{y}_n + 0.5\Delta t \mathbf{k}_2) \\
\mathbf{k}_4 &= \mathbf{F}(t_n + \Delta t, \mathbf{y}_n + \Delta t \mathbf{k}_3) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{\Delta t}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)
\end{aligned}
$$

- ode45 is based on an explicit Runge-Kutta (4,5) formula.

Partial Differential Equation (PDE)

The function being solved for depends on several variables, and the differential equation can include partial derivatives taken with respect to each of the variables.

Useful for modelling waves, heat flow, fluid dispersion, and other phenomena with spatial behavior that changes over time.

Examples:

- Fluid Mechanics: Euler equations, Navier-Stokes equations
- Solid Mechanics: Structural dynamics
- Electrodynamics: Maxwell equations
- Quantum Mechanics: Schrodinger equation

Analytical solutions over arbitrary domains are mostly unavailable.

Classification of PDEs

hyperbolic

- Ex: wave equation $$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0$$
- Hyperbolic equations model the transport of some physical quantity, such as fluids or waves.

parabolic

- Ex: heat equation $$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$
- Parabolic problems describe evolutionary phenomena that lead to a steady state described by an elliptic equation.

elliptic

- Equations without a time derivative. $$\frac{\partial^2 u}{\partial x^2} = 0$$
- Ex: Laplace equation
- Elliptic equations are associated to a special state of a system, in principle corresponding to the minimum of the energy.

Numerical Methods for Solving PDEs

The (major) steps required to compute the numerical solution of to a system of PDEs are

1. Derive discretization of governing equations (Semi-discretization; Space-time discretization; Boundary conditions)
2. Construct spatial mesh (or space-time mesh)
3. If semi-discretized, define temporal mesh
4. Implement and solve
5. Postprocess

1-D PDE Solver in MATLAB

$u(x,t)$ that depends on time t and one spatial variable x

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
```

solves 1-D parabolic and elliptic PDEs

- `m` is the symmetry constant.
- `pdefun` defines the equations being solved.
- `icfun` defines the initial conditions.
- `bcfun` defines the boundary conditions.
- `xmesh` is a vector of spatial values for *x*.
- `tspan` is a vector of time values for *t*.
- `sol` is a 3-D solution array. `ui` = `sol(:,:,i)` is an approximation to the `i`th component of the solution vector u. The element `ui(j,k)` = `sol(j,k,i)` approximates u_i at (*t*,*x*) = (`tspan(j)`,`xmesh(k)`).

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
```

The standard form that pdepe expects is

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x - m\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right)$$

The initial condition function should have signature `u0 = pde_ic(x)`.

The standard form for the boundary conditions expected by the solver is

$$p\left(x,t,u\right) + q\left(x,t\right) f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0$$

and use the function signature `[pl,ql,pr,qr] = pde_bc(xl,ul,xr,ur,t)`.

Partial Differential Equation Toolbox for General PDEs

A typical workflow:

- Convert PDEs to the form required by the toolbox.
- Create a PDE model container specifying the number of equations.
- Define 2-D or 3-D geometry and mesh it using triangular and tetrahedral elements with linear or quadratic basis functions.
- Specify the coefficients, boundary and initial conditions.
- Solve and plot the results at nodal locations or interpolate them to custom locations.

Geometry Definition

- Construct mesh interactively using pdetool
  - Unions and intersections of basic shapes: Rectangles, ellipses, circles, etc.
- Use `pdegeom` to create geometry programmatically
  - Build parametrized, oriented boundary edges
  - Label left and right regions of edges
  - Geometry built from union of regions with similar labels
- Mesh Generation
  - `initmesh` Create initial triangular mesh
  - `adaptmesh` Adaptive mesh generation and PDE solution
  - `jigglemesh` Jiggle internal points of triangular mesh
  - `reinemesh` Refine triangular mesh
  - `tri2grid` Interpolate from PDE triangular mesh to rectangular grid
  - `pdemesh` Plot PDE triangular mesh
  - `pdetriq` Triangle quality measure

Problem Definition

(Here we focus on scalar PDEs.)

Elliptic

$$-\nabla \cdot (c\nabla u) + au = f$$

Parabolic

$$d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f$$

Hyperbolic

$$d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f$$

Eigenvalue

$$-\nabla \cdot (c\nabla u) + au = \lambda du$$

PDE coefficients a, c, d, f can vary with space and time (can also depend on the solution u or the edge segment index)

Boundary conditions

Dirichlet (essential) boundary conditions

$$hu = r \text{ on } \partial\Omega$$

Generalized Neumann (natural) boundary conditions

$$\mathbf{n} \cdot (\nabla u) + qu = g \text{ on } \partial\Omega$$

Boundary coefficients h, r, q, g can vary with space and time (can also depend on the solution u or the edge segment index)

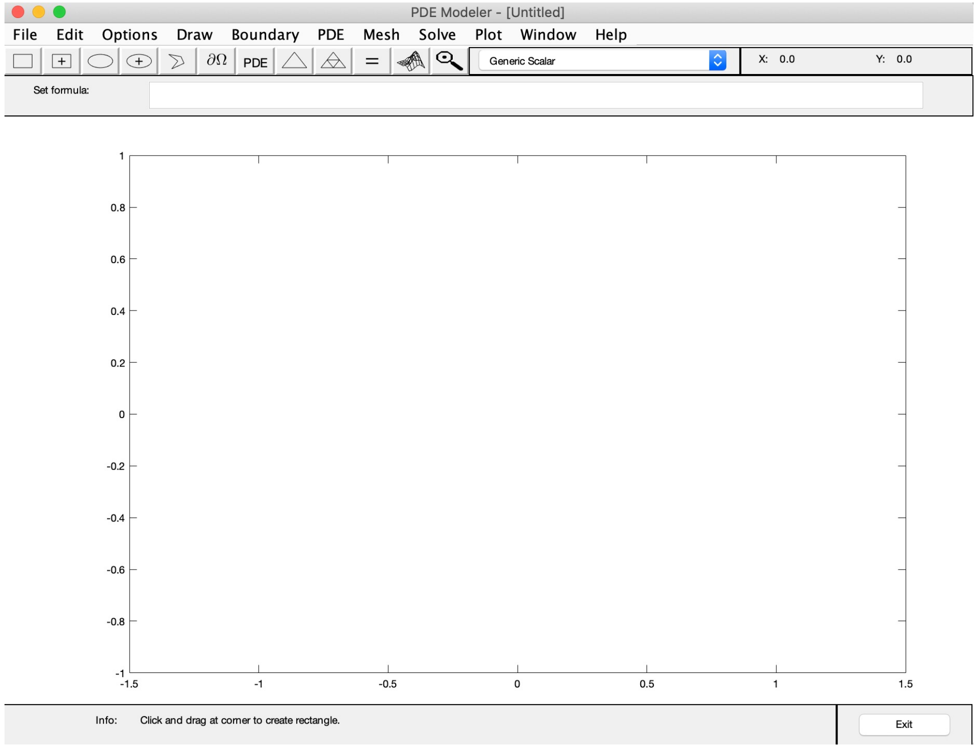
Specify Boundary Conditions

- Graphically using `pdetool`
- Programmatically using `pdebound`:
  `[q,g,h,r] = pdebound(p,e,u,time)`

Specify PDE Coeffcients

- Graphically using `pdetool`
- Programmatically via constants, strings, functions:
  `u = parabolic(u0,tlist,b,p,e,t,c,a,f,d);`

pdetool



Set formula:

Info: Click and drag at corner to create rectangle.

Stanford University

PDE solvers

- Elliptic: `[u,res]=pdenonlin(b,p,e,t,c,a,f);`
- Parabolic: `u=parabolic(u0,tlist,b,p,e,t,c,a,f,d);`
- Hyperbolic: `u=hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d);`

```
result = solvepde(model)
```

solves the stationary PDE represented in `model`, which is a `PDEModel` object that contains the geometry, mesh, and problem coefficients.

Next Lecture

Object Oriented Programming

- OOP fundamentals and features
- Class components
- Implementing operators
- Value vs handle classes
- Event and listener

Efficient Code Writing

- Publishing coding
- Code debugger and analyzer
- Profiler and benchmark
- Code performance and memory management
- Parallel processing

Fun with MATLAB

Type `spy` in Command Window.

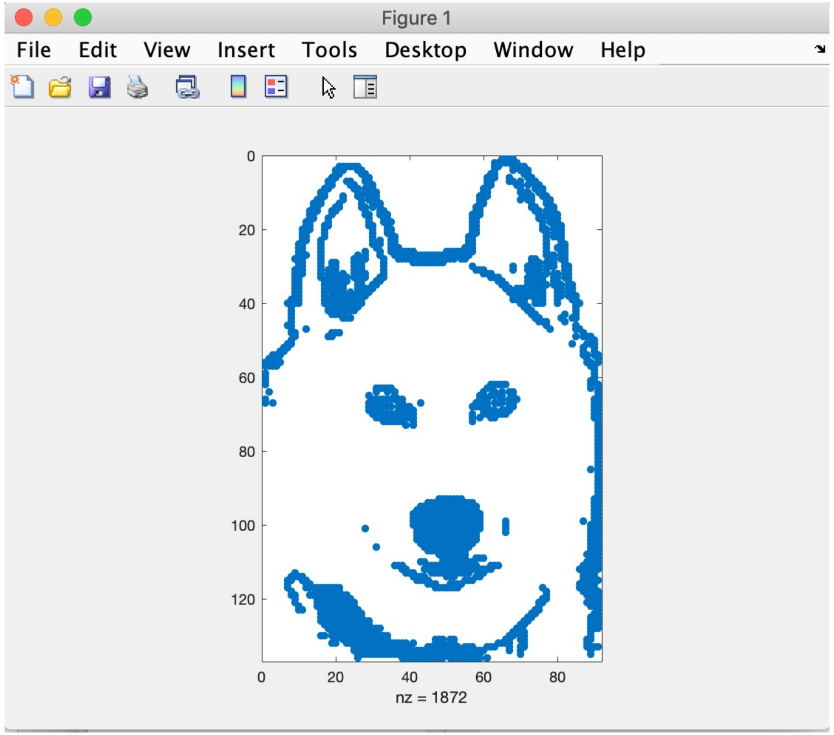


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
0
20
40
60
80
100
120
0
20
40
60
80
nz = 1872