Linear Algebra

Matrices in the MATLAB Environment

Some Function to Get Matrices

`pascal`, `magic`, `ones`, `zeros`, `rand`, `randn`

Vector Products and Transpose

```
u = [3; 1; 4]; % column vector
v = [2, 0, -1]; % row vector
x = v * u; % a scalar, the inner product
y = u * v; % a matrix, the outer product
A = magic(3);
B = A'; % a complex conjugate transpose
C = A.'; % transpose without conjugation
```

Multiplying Matrices

```
A = pascal(3);
B = magic(3);
m = 3; n = 3;
for i = 1 : m
  for j = 1 : n
    C(i, j) = A(i, :) * B(:, j);
  end
end
```

Identity Matrix

```
eye(m, n); % an m-by-n rectangular matrix
```

Kronecker Tensor product

```
% If X is a m-by-n matrix and Y is a p-by-q
% matrix, kron(X, Y) is mp-by nq.
% [X(1, 1)Y, ..., X(1, n)Y,
%         .........
%  X(m, 1)Y, ..., X(m, n) Y]
X = [1, 2; 3, 4];
kron(X, I);
```

Vector and Matrix Norms

The *p-norm* of a vector x,

$$||x||_p = \left(\sum |x_i|^p\right)^{1/p}$$

```
v = [1 0 -1];
[norm(v, 1), norm(v), norm(v, inf)]; % the default value is p = 2
```

The *p-norm* of a matrix A,

$$||A||_p = \max_a \frac{||Ax||_p}{||x||_p},$$

can be computed for $p = 1, 2$, and ∞ by `norm(A, p)`.

```
C = fix(10*rand(3,2));
[norm(C,1) norm(C) norm(C,inf)];
```

Using Multithreaded Computation with Linear Algebra Functions

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.
2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains than several thousand elements or more.
3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complex functions speed up more than simple functions.

System of Linear Equations

Computational Considerations

`(b / A)'; = (A'; \ b';)`

```
x = b / A; % Denotes the solution to the matrix equation xA = b,
           % obtained using mrdivide.
x = A \ b; % Denotes the solution to the matrix equation Ax = b,

           % obtained using mldivide
```

The coefficient matrix A need not be square. If A has size m-by-n, then there are three cases:

m = n	**Square system. Seek an exact solution.**
m > n	**Overdetermined system, with more equations than unknowns. Find a least-squares solution.**
m < n	**Undetermined system, with fewer equations than unknowns. Find a basic solution with at most m nonzero components.**

General Solution

The general solution to a system of linear equations Ax= b describes all possible solutions. You can find the general solution by:

1. Solving the corresponding homogeneous system `Ax = 0`. Do this using the null command, by typing `null(A)`. This returns a basis for the solution space to `Ax = 0`. Any solution is a linear combination of basis vectors.
2. Finding a particular solution to the nonhomogeneous system `Ax =b`.

You can then write any solution to `Ax= b` as the sum of the particular solution to `Ax =b`, from step 2, plus a linear combination of the basis vectors from step 1.

Square Systems

Nonsingular Coefficient Matrix

If the Matrix A is nonsingular, then the solution, $x = A\backslash b$, is the same size as b. It can be confirmed that $A * x$ is exactly equal to b.

```
A = pascal(3);
u = [3; 1; 4];
x = A\u;
% x =
%         10
%        -12
%          5

b = magic(3); % A and b are square and the same size
X = A\b
% X =
%         19    -3    -1
%        -17     4    13
%          6     0    -6
```

Singular Coefficient Matrix

A square matrix A is singular if it does not have linearly independent columns. If A is singular, then the operation `A \ b` issues a warning. If A is singular and $Ax = b$ has a solution, you can find a particular solution that is not unique, by typing `p = pinv(A) * b`. `pinv(!A)` is a pseudoinverse of A, a least-squares solution.

Exact Solutions

Verify that `pinv(A) * b` is an exact solution by typing

```
b = [5; 2; 12];
A = [ 1 3 7
    -1 4 4
    1 10 18 ];
x = pinv(A) * b;
A * pinv(A) * b == b; % may have some problem of precision
```

Least-Squares Solution

If $A * x = b$ has no exact solution, `pinv(A)` returns a least-squares solution.

```
A = [ 1 3 7
    -1 4 4
    1 10 18 ];
b = [3; 6; 0];
b_solved = A * pinv(A) * b; % do not get back the original vector b
```

finding the row reduced echelon form of the augmented matrix `[A, b]` can see whether $Ax = b$ has an exact solution.

```
rref([A, b]);
```

Overdetermined Systems

This example shows how overdetermined systems are often encountered in various kinds of curve fitting to experimental data

```
t = [0 .3 .8 1.1 1.6 2.3]';
y = [.82 .72 .63 .60 .55 .50]';
B = table(t, y);
% y = c1 + c2 * exp(-t)
E = [ones(size(t)) exp(-t)];
c = E \ y;
% y(t) = 0.4760 + 0.3413e-t
T = (0:0.1:2.5)';
Y = [ones(size(T)) exp(-T)]*c;
plot(T,Y,'-',t,y,'o');
```

If A is rank deficient, then the least-squares solution to $AX = B$ is not unique. $A\backslash B$ issues a warning if A is rank deficient and produces a least-squares solution. You can use `lsqminnorm` to find the solution X that has the minimum norm among all solutions.

Underdetermined Systems

The solution to underdetermined systems is **not unique**. Underdetermined linear systems involve more unknowns than equations. The matrix left division operation in MATLAB finds a basic least-squares solution, which has at most m nonzero components for an m-by-n coefficient matrix.

```
R = [6, 8, 7, 3;
     3, 5, 4, 1];
rng(0);
b = randi(8, 2, 1);
format rat;
p = R \ b; % p is the basic least-squares solution, R * p = b
z = null(R, 'r'); % R * z = 0
q = [-2, 1]; % q = [u, w], u and w can be any numbers
x = p + z * q; % the complete general solution
```

It can be confirmed that $R * z$ is zero and that the residual $R * x - b$ is small for any vector x, where

$$x = p + z * q$$

Since the columns of z are the null space vectors, the product $z * q$ is a linear combination of those vectors:

$$Z * q = (\bar{x}_1 \quad \bar{x}_2) \begin{pmatrix} u \\ w \end{pmatrix} = u\bar{x}_1 + w\bar{x}_2.$$

Use the `lsqminnorm` to compute the minimum-norm least-square solution.

```
p = lsqminnorm(R, b); % has the smallest possible value for norm(p)
```

Solving for Several Right-Hand Sides

Solving the linear systems that have the same coefficient matrix A, but different right-hand sides b.

A redundant computation is:

```
X = A \ [b1, b2, b3];
```

Another way is precompute the decomposition of A, but you need to know

which decomposition to compute as well as how to multiply the factors to solve the problem:

```
[L,U] = lu(A);
x = U \ (L \ b);
```

recommended method:

```
dA = decomposition(A, 'lu');
x = dA \ b;
```

An example:

```
n = 1e3;
A = sprand(n,n,0.2) + speye(n);
b = ones(n,1);
% Backslash solution
tic
for k = 1:100
    x = A\b;
end
toc
% Elapsed time is 9.006156 seconds.

% decomposition solution
tic
dA = decomposition(A);
for k = 1:100
    x = dA\b;
end
toc
% Elapsed time is 0.374347 seconds.
```

Iterative Methods

These method is for matrix A which is large and sparse.

Function	Description
pcg	Preconditioned conjugate gradients method. This method is appropriate for Hermitian positive definite coefficient matrix A.
bicg	BiConjugate Gradients Method
bicgstab	BiConjugate Gradients Stabilized Method
bicgstabl	BiCGStab(l) Method
cgs	Conjugate Gradients Squared Method
gmres	Generalized Minimum Residual Method
lsqr	LSQR Method
minres	Minimum Residual Method. This Method is appropriate for Hermitian coefficient matrix A.
qmr	Quasi-Minimal Residual Method
symmlq	Symmetric LQ Method
tfqmr	Transpose-Free QMR Method

Inverses and Determinants

function: `inv`, `det`

`inv` is only for square and nonsingular matrix.

`det` is useful in theoretical considerations and some types of symbolic computation, but is scaling and round-off error properties make it far less satisfactory for numeric computation.

pseudoinverses

Rectangular matrices do not have inverses or determinants. At least one of the equations $AX = I$ and $XA = I$ does not have a solution.

`pinv` function compute the *Moore-Penrose pseudoinverse*:

```
format short;
C = [9, 4;
     2, 8;
     6, 7];
X = pinv(C);
Q = X * C; % Q is a 2-by-2 identity matrix
P = C * X; % P is a 3-by-3 matrix
P * C; % P * C = C
X * P; % X * P = X
```

Solving a Rank-Deficient System

If A is m-by-n matrix and full rank n, each of the three statements

```
x = A\b; % faster
x = pinv(A) * b;
x = inv(A' * A) * A' * b;
```

theoretically computes the same least-squares solution x.

However, if A does not have full rank, the solution to the least-squares problem is not unique.There are many vectors x that minimize `norm(A * x - b)`

`x = A \ b` is a basic solution; it has at most r nonzero components, where r is the rank of A.

`x = pinv(A)` is the minimal norm solution because it minimizes `norm(X)`.

`x = inv(A'; * A) * A'; * b` will be fail because `A'; * A` is singular.

Factorizations

Cholesky Factorization

The Cholesky factorization expresses a **symmetric matrix** (not all, for positive definite) as the product of a triangular matrix and its transpose

$$A = R'R,$$

where R is an upper triangular matrix.

```
A = pascal(6);
R = chol(A);
```

The Cholesky factorization also applies to complex matrices. Any complex matrix that has a Cholesky factorization satisfies $A' = A$ and is said to ***Hermitian positive definite***.

$Ax = b$ can be replaced by $R'Rx = b$.

The backslash operator recognizes triangular systems, it can be solved with `x = R \ (R'; \ b)`

Computational complexity of `chol(A)` is $O(n^3)$, but the complexity of the subsequent backslash solutions is only $O(n^2)$.

LU Factorization (Gaussian elimination)

Express any square matrix A as the product of a permutation of a lower triangular matrix and an upper triangular matrix.

$$A = LU$$

where L is a permutation of a lower triangular matrix with ones on its diagonal and U is an upper triangular matrix.

```
[L, U] = lu(B);
X = U \ (L \ b);
% det(A) == det(L) * det(U);
% inv(A) == inv(U) * inv(L);
% det(A) == prod(diag(U)); % the sign of the determinants might be reversed.
```

QR Factorization

An *orthogonal* matrix, or a matrix with orthonormal columns, is a real matrix whose columns all have unit length and are perpendicular to each other. If Q is orthogonal, then

$$Q'Q = 1$$

For complex matrices, the corresponding term is **unitary**. **Orthogonal and unitary matrices** are desirable for numerical computation because they **preserve length**, **preserve angles**, and **do not magnify errors**.

The orthogonal, or QR, factorization expresses any rectangular matrix as the product of an orthogonal or unitary matrix and an upper triangular matrix. A column permutation might also be involved:

$$A = QR$$

or

$$AP = QR$$

where Q is orthogonal or unitary, R is upper triangular, and P is a permutation.

```
C = gallery('uniformdata', [5, 4], 0);
[Q, R] = qr(C);
[Q, R] = qr(C, 0); % the economy-size QR factorization saves time and memory
[Q, R, P] = qr(C); % largest norm column first
[Q, R, P] = qr(C, 0);
```

Power and Exponentials

Positive Integer Powers

```
A = [1, 1, 1; 1, 2, 3; 1, 3, 6];
p = 2; % p is a positive number
X = A ^ p; % effectively multiplies A by itself p - 1 times
```

Inverse and Fractional Powers

Requires A to be square and nonsingular, A^{-p} effectively multiplies `inv(A)` by itself $p-1$ times

```
Y = A ^ (-3);
```

Fractional powers, like $A^{2/3}$, are also permitted; the results depend upon the distribution of the eigenvalues of the matrix.

Element-by-Element Powers

use the `.^`

Exponentials

`sqrtm(A)` computes `A^(1/2)` by a more accurate algorithm.

`sqrt(A)` computes element-by-element.

Eigenvalues

Eigenvalue Decomposition

An eigenvalue and eigenvector of a square matrix A are, respectively, a scalar λ and a nonzero vector v that satisfy

$$Av = \lambda v$$

With the eigenvalues on the diagonal of a diagonal matrix Λ and the corresponding eigenvectors forming the columns of a matrix V, you have

$$AV = VA$$

If V is nonsingular, this becomes the eigenvalue decomposition

$$A = V\Lambda V^{-1}$$

```
A = [
  0, -6, -1;
  6, 2, -16;
  -5, 20, -10;
]; % A is nonsigular
lambda = eig(A); % a column vector containing the eigenvalues, some of them may be comlex
[V, D] = eig(A); % V: the eigenvectors with Eucildean length, D: eigenvalues in a diagonal
matrix
% A = V * D * inv(V); with round-off error
% D = inv(V) * A * V; with round-off error
```

Multiple Eigenvalues

Some matrices do not have an eigenvector decomposition. These matrices are not diagonalizable.

```
A = [
  1, -2, 1;
  0, 1, 4;
  0, 0, 3
];
[V, D] = eig(A); % V has two same vectors, D have two same eigenvalues
```

A full set of linearly independent eigenvectors does not exist.

Schur Decomposition

The Schur Decomposition:

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

where U is an orthogonal matrix and S is a block upper triangular matrix with 1-by-1 and 2-by2 blocks on the diagonal.

```
A = [
  1, -2, 1;
  0, 1, 4;
  0, 0, 3
];
[U, S] = schur(A);
```

Singular Values

A singular value and corresponding singular vectors of a rectangular matrix A are, respectively, a scalar σ and a pair of vectors u and v that satisfy

$$Av = \sigma u$$
$$A^H u = \sigma v,$$

where A^H is the Hermitian transpose of A. The singular vectors u and v are typically

scaled to have a norm of 1. Also, if u and v are singular vectors of A, then $-u$ and $-v$ are singular vectors of A as well.

The singular values σ are always real and nonnegative, even if A is complex. With the singular values on the diagonal of a diagonal matrix Σ and the corresponding singular vectors forming the columns of two orthogonal matrices U and V, you obtain the equations

$$AV = U\Sigma$$
$$A^H U = V\Sigma$$

Since U and V are unitary matrices, multiplying the first equation by

on the right yields the singular value decomposition equation

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

```
A = [
  9, 4;
  6, 8;
  2, 7;
];
[U, S, V] = svd(A); % A = U * s * V' with round-off error
[U, S, V] = svd(A, 0); % economy-size decomposition
```

Only want some largest singular values in a large and sparse matrix A, use the function `svds`

```
n = 1000;
A = sprand(n, n, 0.3);
S = svds(A);
S = svds(A, 6, 'smallest'); % the six smallest singular values
svd(full(A));
```

Operation on Nonlinear Functions

Function Handles

```
fhandle = @sin;

function plot_fhandle(fhandle, data)
plot(data, fandle(data))

plot(data, fhandle(data));
```

Function Functions

A class of functions called “function functions” works with nonlinear functions of a scalar variable. That is, one function works on another function. The function functions include

- Zero finding

- Optimization
- Quadrature
- Ordinary differential equations

MATLAB represents the nonlinear function by the file that defines it.

```
% my_function.m
function y = humps(x)
y = 1.0 / ((x - 0.3)) .^ 2 + 0.01) + 1.0 / ((x - 0.9) .^ 2 + 0.04) -
```

```
% use_my_function.m
x = 0:0.002:1;
y = my_function(x);
plot(x, y, '-');
p = fminsearch(@my_function, .5);
Q = quadl(@my_function, 0, 1);
z = fzero(@my_function, 5); % cannot work on functions having no sign change
```

`quadl` see also Lobatto Method.

`quad` see also Simpsom's rule.

Multivariate Data

In **MATLAB**, `column`: *observation*, `row`: *variable*.

```
D = [
  72, 134, 3.2;
  81, 201, 3.5;
  69, 156, 7.1;
  82, 148, 2.4;
  75, 170, 1.2
];
mu = mean(D);
sigma = std(D);
```

see also `datafun` and `stats`

Data Analysis

Introduction

standard components:

- `Preprocessing` Consider outliers and missing values, and smooth data to identify possible models.
- `summarizing` Compute basic statistics to describe the overall location, scale, and shape of the data.
- `Visualizing` Plot data to identify patterns and trends.

- `Modeling` Give data trends fuller descriptions, suitable for predicting new values.

Two basic goals:

1. Describe the patterns in the data with simple models that lead to accurate predictions.
2. Understand the relationships among variables that lead to the model.

Preprocessing Data

Loading the Data

```
load count.dat
```

Missing Data

`NaN` is normally used to represent missing data.

```
col3 = count(:, 3); % data at intersection 3
zeros_in_col3 = sum(isnan(col3)); % isnan returns a logical vector the same size as c3
```

Outliers

```
bin_counts = hist(c3); % Histogram bin counts
N = max(bin_counts); % Maximum bin count
mu3 = mean(c3); % Data mean
sigma3 = std(c3); % Data standard deviation

hist(c3) % Plot histogram
hold on
plot([mu3 mu3], [0 N], 'r', 'LineWidth', 2) % Mean
X = repmat(mu3 + (1:2) * sigma3, 2, 1);
Y = repmat([0; N], 1, 2);
plot(X, Y, 'g', 'LineWidth', 2)
legend('Data', 'Mean', 'Stds')
hold off
outliers = (c3 - mu3) > 2 * sigma3;
c3m = c3; % copy c3 to c3m
c3m(outliers) = NaN; % add NaN values
```

Smoothing and Filtering

```
plot(c3m, 'o-')
hold on

span = 3;
window = ones(span, 1) / span;
smoothed_c3m = convn(c3m, window, 'same');

h = plot(smoothed_c3m, 'ro-');
legend('Data', 'Smoothed Data')

smoothed2_c3m = filter(window, 1, c3m);

delete(h)
plot(smoothed2_c3m, 'ro-', 'DisplayName', 'Smoothed Data')
```

Summarizing Data

Measures of Location

`mean` `median` `mode`

Measures of Scale

`max` `min` `std` `var`

Shape of a Distribution

```
figure
hist(count)
legend('Intersection 1', 'Intersection 2', 'Intersection 3')

c1 = count(:, 1);
[bin_counts, bin_locations] = hist(c1);
bin_width = bin_locations(2) - bin_location(1);
hist_area = (bin_width) * (sum(bin_counts));

figure
hist(c1)
hold on

mu1 = mean(c1);
exp_pdf = @(t)(1 / mu1) * exp(-t / mul);

t = 0:150;
y = exp_pdf(t);
plot(t, (hist_area) * y, 'r', 'LineWidth', 2);
legend('Distribution', 'Exponetatial Fit');
```

Visualizing Data

2-D Scatter Plots

`scatter`

`cov` convariance

`corrcoef`

3-D Scatter Plots

```
c1 = count(:, 1);
c2 = count(:, 2);
c3 = count(:, 3);
scatter3(c1, c2, c3, 'filled')
vars = eig(cov([c1, c2, c3]));
explained = max(vars) / sum(vars);
```

Scatter Plot Arrays

```
figure
plotmatrix(count)
```

Modeling Data

Polynomial Regression

```
load count.dat
c3 = count(:, 3);
tdata = (1:24)';
p_coeffs = plotfit(tdata, c3, 6);

figure
plot(c3, 'o-')
hold on
tfit = (1:0.01:24)';
yfit = ployval(p_coeffs, tfit);
plot(tfit, yfit, 'r-', 'LineWidth', 2)
legend('Data', 'Polynomial Fit', 'Location', 'NW');
```

General Linear Regression

```
load count.dat
c3 = count(:, 3);
tdata = (1:24)';
X = [ones(size(tdata)), cos(2 * pi / 12) * (tdata - 7)];
s_coeffs = X \ c3;

figure
plot(c3, 'o-')
hold on

tfit = (1:0.01:24)';
yfit = [ones(size(tfit)), cos(2 * pi / 12) * (tfit - 7)] * s_coeffs;
plot(tfit, yfit, 'r-', 'LineWidth', 2)
legend('Data', 'Sinusoidal Fit', 'Location', 'NW')

[s_coeffs, stdx, mse] = lscov(X, c3) % mean squared error
```