# 5 LINEAR ALGEBRA

## Contents

## Initialization

```
% reduce needless whitespace
format compact
% reduce irritations
more off
% start a diary
%diary lectureN.txt
```

## SOLVING LINEAR SYSTEMS OF EQUATIONS

In the next subsection, we will solve a system of 3 equations in 3 unknowns. That is just a small example of much larger systems of maybe billions of equations in billions of unknowns used to solve flow fields by modern engineers.

```
disp('SOLVING LINEAR SYSTEMS OF EQUATIONS:')
```

```
 SOLVING LINEAR SYSTEMS OF EQUATIONS:
```

### The problem we want to solve

As an example, we want to solve the system of equations

```
   x1 + 2 x2 + 3 x3 = 3
        5 x2 + 6 x3 = 2
 7 x1 + 8 x2 + 9 x3 = 9
```

for the unknowns x1, x2, and x3.
*Note that we have taken the terms involving unknowns to the left and terms without unknowns to the right. We have also ordered the unknowns*

### Put the problem in vector matrix form

To find the solution, first put the coefficients of the unknowns in a "matrix" A. Also put the right hand sides in a column vector b. Note that we use uppercase for matrices and lowercase for vectors.

```
    A = [1 2 3;                         b = [3;
         0 5 6;         and                  2;
         7 8 9]                              9]
```

Note that the unknowns must be ordered and you must enter a zero for missing unknowns. Also note that inside a matrix, a semi-colon starts a new line. A comma does not do anything special. The Newlines inside the brackets are for readability; the next would also work:

```
  A=[0 2 3; 2 3 4; 5 6 7]
  b=[3;2;9]
```

but would lose credit because it is a mess.

```
% create the matrix
disp('Create the system matrix A:')
A = [1 2 3;
     0 5 6;
     7 8 9]

% Put the right hand sides in a column vector b:
disp('Create the right hand side vector b')
b = [3;
     2;
     9]
```

```
 Create the system matrix A:
A =
    1    2    3
    0    5    6
    7    8    9
 Create the right hand side vector b
b =
    3
    2
    9
```

## Solve the system the correct way

The system can now compactly be written as

```
 A x = b
```

with A and b as above and x the column vector of the three unknowns. So, if A and b were simple numbers, the solution would be x = b/A. But A and b are **not** simple numbers.
The *wrong* way to solve would now be to first check that the determinant of A is nonzero and then find the solution vector x as the inverse of A times b:

```
 % Wrong, zero credit:
 det(A)
 disp('det A is nonzero, so OK.')
 x = inv(A)*b
```

Determinants work in *exact* mathematics, but *not* in numerical methods, where there are numerical errors, overflow, and underflow. And computing an inverse matrix is very to extremely inefficient, and tends to increase round-off errors. The correct way is to check that the condition number of A is not too large. If it is not then solve the system using "left division"

```
% OK, credit
condA=cond(A)
disp('The condition number is not too large, so OK.')
x = A \ b      (left division: A\b instead of b/A)
```

The meaning of the condition number is as follows:

```
The condition number determines by what factor
    the matrix magnifies relative errors.
```

So if the condition number is large, even very small errors in the coefficients of matrix A and vector b can produce large errors in the solution vector x.

```
% the bad way
disp('Solve the system the zero-credit way:')
baddetA=det(A)
badx=inv(A)*b

% the good way: check the condition number
disp('Solve the system the good way:')
disp('Check whether the condition number is OK:')
condA=cond(A)
disp('The condition number is not too large, so OK.')
disp('Now solve the correct way: use "Left division",')
disp('i.e. not b/A but A\b:')
x = A \ b
disp('This is the correct solution to the system of')
disp('equations as  given.  But note that if the values')
disp('of A and b have measurement errors of just 0.1%,')
disp('then the computed x values may have relative')
disp('errors as high as 4%:')
relErrMeasurement=0.001
xRelErrDueToMeasurement=relErrMeasurement*condA
disp('Without checking the condition number, we would')
disp('have no clue of that!')
```

```
Solve the system the zero-credit way:
baddetA = -24
badx =
    1.0000
   -2.0000
```

```
    2.0000
 Solve the system the good way:
 Check whether the condition number is OK:
 condA =   37.939
 The condition number is not too large , so OK.
 Now solve the correct way: use "Left division",
 i.e. not b/A but A\b:
 x =
    1
   -2
    2
 This is the correct solution to the system of
 equations as  given.  But note that if the values
 of A and b have measurement errors of just 0.1%,
 then the computed x values may have relative
 errors as high as 4%:
 relErrMeasurement =   0.0010000
 xRelErrDueToMeasurement =   0.037939
 Without checking the condition number, we would
 have no clue of that!
```

## Problematic matrices

Consider now the modified system of equations

```
   x1 + 2 x2 + 3 x3 = 3
 4 x1 + 5 x2 + 6 x3 = 2
 7 x1 + 8 x2 + 9 x3 = 9
```

The only change is the additional 4 x1 in the second equation. But the matrix is now singular, i.e. it has a zero determinant. In that case there is normally no solution at all. (If there is a solution, there are infinitely many other ones that are just as good).
To form the new matrix, we want to take the old matrix and just change the zero in row 2, column 1 into a 4. We can do that with "indices". Always remember:

```
 For matrices, the proper order is row-column
```

In particular, the element in row 2 and column 1 of A is A(2,1). The numbers 2 and 1 are called the "indices" of the element. Note that the row number 2 goes before the column number 1.

```
% Change the element of A in row 2 and column 1 into a 4.
disp('Let''s make A singular now:')
A(2,1)=4

% check the condition number
```

```
condA=cond(A)
disp('The condition number is excessive.')
disp('Even with its 10^-16 relative error, Matlab can')
disp('not find the solution to an acceptable error:')
xRelErrorDueToMatlab=condA*eps(1)
disp('The 10^-16 relative error in A and b will produce')
disp('a relative error in x of about 1,300%!')

% Try solving again
x = A \ b
disp('Nice numbers, but they are all wrong:')
disp('the correct solution is infinite!')
```

```
Let's make A singular now:
A =
    1   2   3
    4   5   6
    7   8   9
condA =    6.0262e+16
The condition number is excessive.
Even with its 10^-16 relative error, Matlab can
not find the solution to an acceptable error:
xRelErrorDueToMatlab =  13.381
The 10^-16 relative error in A and b will produce
a relative error in x of about 1,300%!
warning: matrix singular to machine precision, rcond =
    2.20304e-18
x =
   0.50000
   0.33333
   0.16667
Nice numbers, but they are all wrong:
the correct solution is infinite!
```

## MATRIX MANIPULATIONS

For advanced applications in linear algebra you must know how to do certain tasks.

### Transposes

The transpose of a matrix has rows and columns swapped:

```
Transposing swaps rows and columns
```

As we already saw

```
 Transposing can be done with a single-quote
```

The conventional symbol for transpose is a superscript T.

```
disp('Create "transposes" (indicated by T) using'':')

% try it for vector b
b
bT=b'
bTT=bT'

% try it for matrix A
A
AT=A'
ATT=AT'
```

```
 Create "transposes" (indicated by T) using':
b =
    3
    2
    9
bT =
    3    2    9
bTT =
    3
    2
    9
A =
    1    2    3
    4    5    6
    7    8    9
AT =
    1    4    7
    2    5    8
    3    6    9
ATT =
    1    2    3
    4    5    6
    7    8    9
```

## Matrix multiplication

We never checked whether A x is really b. Now multiplying matrix A and column vector x together is an example of matrix multiplication, because a three- dimensional column vector is also a matrix with 3 rows and 1 column.

The key thing to remember is:

```
Matrix multiplication is always row-column.
```

In particular, if we multiply matrices

```
      A = [1  2  3;                  x = [x1;
           0  5  6;      and              x2;
           7  8  9]                       x3]
```

together, we get

```
 [x1 + 2 x2 + 3 x3;                                   b = [3;
  5 x2 + 6 x3;             which *should* equal           2;
  7 x1 + 8 x2 + 9 x3]                                     9]
```

after substituting the values of x. Note that the number 3 in row 1, column 1 of b is found as a dot product between row 1 of A and the single column 1 of vector x. Similarly the number 2 in row 2, column 1 of b is a dot product between row 2 of A and the single column 1 of vector x, and similarly for the number 9 in row 3, column 1 of b.

```
The row-column multiplications are dot products.
```

From the above, it follows

```
The rows and columns involved in matrix
multiplications must have the same number of
elements.
```

```
disp('Let''s try some matrix multiplications:')

% restore the nonsingular matrix and its x for now
disp('First back to the nonsingular A and x:')
A(2,1)=0
x = A \ b

% do NOT put . before the * for matrix multiplication
disp('Check what A*x is (do NOT use .* here):')
valueAStarx=A*x
errorAStarx=A*x-b
disp('All OK as expected.')

% repeat for the singular case
disp('let''s try the singular matrix too:')
A(2,1)=4
x = A \ b
valueAStarx=A*x
errorAStarx=A*x-b
```

```
% reset A and x
disp('Back to the nonsingular A and x:')
A(2,1)=0
x = A \ b
```

```
Let's try some matrix multiplications:
First back to the nonsingular A and x:
A =
   1   2   3
   0   5   6
   7   8   9
x =
   1
  -2
   2
Check what A*x is (do NOT use .* here):
valueAStarx =
   3
   2
   9
errorAStarx =
   0
   0
   0
All OK as expected.
let's try the singular matrix too:
A =
   1   2   3
   4   5   6
   7   8   9
warning: matrix singular to machine precision, rcond =
    2.20304e-18
x =
   0.50000
   0.33333
   0.16667
valueAStarx =
   1.6667
   4.6667
   7.6667
errorAStarx =
  -1.3333
   2.6667
  -1.3333
Back to the nonsingular A and x:
```

```
A =
     1     2     3
     0     5     6
     7     8     9
x =
     1
    -2
     2
```

### More matrix multiplication

```
% let's play a bit with matrix multiplication
disp('How about some more multiplications?')
A
B = [x x b x]
disp('Each column in B is multiplied to A separately:')
AStarB=A*B
disp('Note that AB is not BA; BA does not exist:')
disp('the four element rows of B cannot be dotted with')
disp('the three element columns of A.')
disp('And even if they could be multiplied, normally')
disp('AB is not the same as BA (exceptions exists).')

% transpose
disp('If b is a column vector:')
b
disp('then the "transpose" of b is a row vector:')
bT=b'
% dot product of b with itself
disp('row vector bT * column vector b is a dot product:')
bTStarb=b'*b
disp('It equals the square length of either vector.')

% "outer" product of b with itself
disp('column vector b * row vector bT is a matrix:')
b
bT=b'
bStarbT=b*b'

% b^2 fails: row length 1 times column length 3 is bad
disp('Unlike bT*b and b*bT, b*b cannot be multiplied.')
disp('And so, neither can b^2 be evaluated.')
disp('The same for bT.')

% "elementwise" computation of x
```

```
disp('b .* b is multiplied elementwise:')
bPtStarb=b.*b
disp('and so is b.^2:')
bPtSquareb=b.^2
disp('The same for bT:')
bTPtStarbT=bT.*bT
bTPtSquarebT=bT.^2

% some more multiplications
disp('The same for bigger matrices:')
A
AStarA=A*A
disp('For square matrices, ^2 works:')
ASquareA=A^2
APtStarA=A.*A
APtSquareA=A.^2
```

```
How about some more multiplications?
A =
    1    2    3
    0    5    6
    7    8    9
B =
    1    1    3    1
   -2   -2    2   -2
    2    2    9    2
Each column in B is multiplied to A separately:
AStarB =
       3       3      34       3
       2       2      64       2
       9       9     118       9
Note that AB is not BA; BA does not exist:
the four element rows of B cannot be dotted with
the three element columns of A.
And even if they could be multiplied, normally
AB is not the same as BA (exceptions exists).
If b is a column vector:
b =
    3
    2
    9
then the "transpose" of b is a row vector:
bT =
    3    2    9
row vector bT * column vector b is a dot product:
bTStarb =  94
```

```
It equals the square length of either vector.
column vector b * row vector bT is a matrix:
b =
   3
   2
   9
bT =
   3   2   9
bStarbT =
    9    6   27
    6    4   18
   27   18   81
Unlike bT*b and b*bT, b*b cannot be multiplied.
And so, neither can b^2 be evaluated.
The same for bT.
b .* b is multiplied elementwise:
bPtStarb =
    9
    4
   81
and so is b.^2:
bPtSquareb =
    9
    4
   81
The same for bT:
bTPtStarbT =
    9    4   81
bTPtSquarebT =
    9    4   81
The same for bigger matrices:
A =
   1   2   3
   0   5   6
   7   8   9
AStarA =
    22    36    42
    42    73    84
    70   126   150
For square matrices, ^2 works:
ASquareA =
    22    36    42
    42    73    84
    70   126   150
APtStarA =
     1     4     9
    0    25    36
   49    64    81
APtSquareA =
    1     4     9
    0    25    36
   49    64    81
```

## Parts of matrices

Take parts out of matrices using START:END constructs.

```
disp('Try taking parts out of matrices:')

% make a bigger matrix to test
disp('First, let''s make a bigger matrix:')
Big=[A AT]
% the size of the matrix is again row-column
dims=size(Big)

% Taking part of a row out of a matrix (note row-column!)
row2part=Big(2,2:4)

% Taking an entire row out
row2all=Big(2,:)
% Bad, since less readable:
%row2=Big(2,1:end)
% Worse:
%row2=Big(2,1:6)

% Taking the columns out of a matrix (important)
col4=Big(:,4)

% Taking three columns out at the same time
col345=Big(:,3:5)

% deleting a column
disp('Let''s delete column 2 in AT:')
AT
AT(:,2)=[]
```

```
Try taking parts out of matrices:
First, let's make a bigger matrix:
Big =
     1     2     3     1     4     7
     0     5     6     2     5     8
     7     8     9     3     6     9
```

```
dims =
    3    6
row2part =
    5    6    2
row2all =
    0    5    6    2    5    8
col4 =
    1
    2
    3
col345 =
    3    1    4
    6    2    5
    9    3    6
Let's delete column 2 in AT:
AT =
    1    4    7
    2    5    8
    3    6    9
AT =
    1    7
    2    8
    3    9
```

## Special matrices

Two important types of matrices are zero matrices and unit matrices.
A zero matrix is the matrix equivalent of the number zero. Adding or subtracting a zero matrix A to something does not do anything. Multiplying by a zero matrix produces zero. A zero matrix contains all zeros.
A unit matrix is the matrix equivalent of the number 1; multiplying by a unit matric does not change anything. A unit matrix is square and contains zeros except on the "main diagonal" that goes from top left corner to bottom right corner.
A matrix is symmetric if it is the same as its transpose. Symmetric matrices occur in many engineering applications.

```
disp('Let''s look at some special matrices:')

% a zero matrix consists of all zeros
disp('Adding a "zero matrix" makes no difference:')
Z=zeros(3,6)
Z=zeros(size(Big))
Big
BigPlusZ=Big+Z
disp('Multiplying by a zero matrix produces zero:')
```

```
[m n]=size(Big)
Z=zeros(m)
Big
ZStarBig=Z*Big
Z=zeros(n)
Big
BigStarZ=Big*Z
Z=zeros(n,1)
Big
BigStarZ=Big*Z

% a unit matrix has ones on the main diagonal
disp('Multyplying by a unit matrix makes no difference:')
I=eye(3)
Big
IStarBig=I*Big
I=eye(6)
Big
BigStarI=Big*I
I=eye(3)
x
IStarx=I*x
xT=x'
xTStarI=xT*I

% look at a symmetric matrix
disp('An example symmetric matrix:')
S = [3  4  5;
     4  6  7;
     5  7  8]
disp('The transpose is the same:')
ST=S'
```

```
Let's look at some special matrices:
Adding a "zero matrix" makes no difference:
Z =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
Z =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
Big =
     1     2     3     1     4     7
     0     5     6     2     5     8
```

```
    7     8     9     3     6     9
BigPlusZ =
    1     2     3     1     4     7
    0     5     6     2     5     8
    7     8     9     3     6     9
Multiplying by a zero matrix produces zero:
m =   3
n =   6
Z =
    0     0     0
    0     0     0
    0     0     0
Big =
    1     2     3     1     4     7
    0     5     6     2     5     8
    7     8     9     3     6     9
ZStarBig =
    0     0     0     0     0     0
    0     0     0     0     0     0
    0     0     0     0     0     0
Z =
    0     0     0     0     0     0
    0     0     0     0     0     0
    0     0     0     0     0     0
    0     0     0     0     0     0
    0     0     0     0     0     0
    0     0     0     0     0     0
Big =
    1     2     3     1     4     7
    0     5     6     2     5     8
    7     8     9     3     6     9
BigStarZ =
    0     0     0     0     0     0
    0     0     0     0     0     0
    0     0     0     0     0     0
Z =
    0
    0
    0
    0
    0
    0
Big =
    1     2     3     1     4     7
    0     5     6     2     5     8
    7     8     9     3     6     9
```

```
BigStarZ =
    0
    0
    0
Multyplying by a unit matrix makes no difference:
I =
Diagonal Matrix
    1    0    0
    0    1    0
    0    0    1
Big =
    1    2    3    1    4    7
    0    5    6    2    5    8
    7    8    9    3    6    9
IStarBig =
    1    2    3    1    4    7
    0    5    6    2    5    8
    7    8    9    3    6    9
I =
Diagonal Matrix
    1    0    0    0    0    0
    0    1    0    0    0    0
    0    0    1    0    0    0
    0    0    0    1    0    0
    0    0    0    0    1    0
    0    0    0    0    0    1
Big =
    1    2    3    1    4    7
    0    5    6    2    5    8
    7    8    9    3    6    9
BigStarI =
    1    2    3    1    4    7
    0    5    6    2    5    8
    7    8    9    3    6    9
I =
Diagonal Matrix
    1    0    0
    0    1    0
    0    0    1
x =
    1
   -2
    2
IStarx =
    1
   -2
```

```
    2
xT =
    1   −2    2
xTStarI =
    1   −2    2
An example symmetric matrix:
S =
    3    4    5
    4    6    7
    5    7    8
The transpose is the same:
ST =
    3    4    5
    4    6    7
    5    7    8
```

## Eigenvalues and eigenvectors

A vector e is an eigenvector of a square matrix A if

```
        A e = lambda e
```

where lambda is a number called the eigenvalue.
Finding eigenvalues and eigenvectors is important for very many engineering problems. For example, the principal moments of inertia of a rotating body are eigenvalues. The corresponding eigenvectors are the unit vectors of the "principal coordinate system". Also, the eigenvalues of "stiffness matrices" of vibrating systems give the frequencies of vibration, and the eigenvectors give the mode shapes. Eigenvalues and eigenvectors are also critical in beam bending, in beam buckling, in the stresses and strains in materials under loads, and so on.
Here we want to explore how, given a matrix A, you can find its eigenvalues and eigenvectors.

```
% see what is available to do so
%lookfor eigenvalue
```

## A simple example

```
disp('Let''s find some eigenvalues and eigenvectors!')

% example symmetric matrix
disp('The "strain rate" matrix S in Couette flow:')
C=1
```

```
S = [0 C 0;
     C 0 0;
     0 0 0]
lambda=eig(S)
disp('Separate out the eigenvalues:')
lambda1=lambda(1)
lambda2=lambda(2)
lambda3=lambda(3)
[E Lambda]=eig(S)
disp('Separate out eigenvectors:')
e1=E(:,1)
e2=E(:,2)
e3=E(:,3)

% let's check that Matlab found the right vectors
Se1=S*e1
lambda1e1=lambda1*e1
errors1=S*e1-lambda1*e1
Se2=S*e2
lambda2e2=lambda2*e2
errors2=S*e2-lambda2*e2
Se3=S*e3
lambda3e3=lambda3*e3
errors3=S*e3-lambda3*e3
```

```
Let's find some eigenvalues and eigenvectors!
The "strain rate" matrix S in Couette flow:
C =  1
S =
    0    1    0
    1    0    0
    0    0    0
lambda =
   -1
    0
    1
Separate out the eigenvalues:
lambda1 = -1
lambda2 = 0
lambda3 =  1
E =
   -0.70711    0.00000    0.70711
    0.70711    0.00000    0.70711
    0.00000    1.00000    0.00000
Lambda =
Diagonal Matrix
```

```
  -1    0    0
   0    0    0
   0    0    1
Separate out eigenvectors:
e1 =
  -0.70711
   0.70711
   0.00000
e2 =
   0
   0
   1
e3 =
   0.70711
   0.70711
   0.00000
Se1 =
   0.70711
  -0.70711
   0.00000
lambda1e1 =
   0.70711
  -0.70711
  -0.00000
errors1 =
   0
   0
   0
Se2 =
   0
   0
   0
lambda2e2 =
   0
   0
   0
errors2 =
   0
   0
   0
Se3 =
   0.70711
   0.70711
   0.00000
lambda3e3 =
   0.70711
```

```
    0.70711
    0.00000
errors3 =
    0
    0
    0
```

## About symmetric matrices

As already noted, a matrix A is symmetric if it equals its transpose; A=A'. There are some special rules for the eigenvalues and eigenvectors of symmetric matrices:

1. The eigenvalues are always real, not complex.

1. The eigenvectors can be taken to be mutually orthogonal unit vectors. It is said that the matrix of eigenvectors is "orthonormal".

(For complex matrices, these things remain true if you replace "symmetric" by "Hermitian". A matrix A is Hermitian if it is equal to its *complex conjugate* transpose.)
Note: the inverse of an orthonormal matrix is the same as its (Hermitian) transpose.

```
% Since our example matrix was symmetric, let's check
% whether Matlab found the right eigenvalues and
% eigenvectors.  The eigenvalues, -1, 0, and 1, are
% indeed real, check.

% the length of the vectors can be computed using norm
e1Length=norm(e1)
e2Length=norm(e2)
e3Length=norm(e3)

% or dot the vector with itself and take square root
e1Length=sqrt(e1'*e1)
e2Length=sqrt(e2'*e2)
e3Length=sqrt(e3'*e3)

% vectors are orthogonal if their dot product is zero
e1e2=e1'*e2
e2e3=e2'*e3
e3e1=e3'*e1

% a quicker way is to check that ET E is the unit matrix
ET=E'
E
```

```
ET*E
```

```
e1Length =  1
e2Length =  1
e3Length =  1
e1Length =  1
e2Length =  1
e3Length =  1
e1e2 = 0
e2e3 = 0
e3e1 = 0
ET =
  -0.70711   0.70711   0.00000
   0.00000   0.00000   1.00000
   0.70711   0.70711   0.00000
E =
  -0.70711   0.00000   0.70711
   0.70711   0.00000   0.70711
   0.00000   1.00000   0.00000
ans =
   1.00000   0.00000   0.00000
   0.00000   1.00000   0.00000
   0.00000   0.00000   1.00000
```

## ADDITIONAL REMARKS

In left division, Matlab will examine the matrix and if the matrix has special properties that warrant a special solution procedure, select it. To save Matlab time or force it to use a given procedure, you can use *linsolve*, which allows you to specify options.
If the matrix is "sparse", i.e. it is a big matrix whose elements are almost all zeros, you should create it as a Matlab sparse matrix. This avoids wasting storage to store all these zeros, and wasting computational time to do trivial operations on all these zeros. You can create Matlab sparse matrices with the *sparse* function. If the matrix is a band matrix, i.e. the nonzero elements are along 45 degree downward diagonals, function *spdiags* may be a more suitable way to create the sparse matrix.

## End lesson 5