5 LINEAR ALGEBRA

Contents

Initialization

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

% Tell the students to save their work space
disp('Save your workspace before leaving!')
```

```
Save your workspace before leaving!
```

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, say, solve flow fields by modern engineers.

```
disp(' ')
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

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 3 \\ 5x_2 + 6x_3 &= 2 \\ 7x_1 + 8x_2 + 9x_3 &= 9 \end{aligned}$$

for the unknowns x_1, x_2, and x_3.
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 and the right hand sides of the equations into a "right hand side vector $\vec{b}$:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} 3 \\ 2 \\ 9 \end{pmatrix}$$

Note that the unknowns must be ordered. Also note that $x_1 = 1x_1$ and no x_1 is the same as $0x_1$. Also note that it is customary to use uppercase for matrices and lowercase for vectors.
In Matlab, a "matrix" is just a two-dimensional "array", (a table of with rows and columns). The most straightforward way to create such an array is by writing it out. For our matrix that is done as:

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

Note that inside an array, 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]
```

but would lose credit because it is a mess.
Also note that the use of the term "matrix" instead of simply "array" means that you are planning to use the array in various special ways. In this case, as a way to solve a system of equations.
In Matlab a column vector is just a one-dimensional column array (or a two-dimensional array with just a single column). The right hand side vector of our system can be created as:

```
b = [3;
     2;
     9]
```

```
% create the matrix
disp('Create the system matrix A:')
A = [1  2  3;
     0  5  6;
     7  8  9]
disp('The more concise A=[1  2  3; 0  5  6; 7  8  9],')
A=[1  2  3; 0  5  6; 7  8  9]
disp('also works.  However that reduces credit because')
disp('it is a mess.')

% 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
The more concise A=[1 2 3; 0 5 6; 7 8 9],
A =
   1   2   3
   0   5   6
   7   8   9
also works. However that reduces credit because
it is a mess.
Create the right hand side vector b
b =
   3
   2
   9
```

Check whether the system is solvable, the correct way

The system can now compactly be written as

$$A\vec{x} = \vec{b} \qquad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

where matrix A and right-hand side vector $\vec{b}$ are as described above, and $\vec{x}$ is the column vector of the three unknowns x_1, x_2, and x_3.
To solve the system correctly, first you *must* check that there is a numerically accurate solution to this system of equations in the first place. If there is not, then what is the point in solving it?
The *wrong* way to proceed would here be to check first that the determinant of A is nonzero:

```
% WRONG, zero credit:
det(A)
disp('det(A) is nonzero so the system is solvable.')
```

The above would be fine if you would do *exact* mathematics. But computers do not normally do exact mathematics. They have numerical errors. In the presence of numerical errors, whatever you compute for a determinant is *meaningless*. A numerically zero determinant does not mean the system is unsolvable. Instead it may very well be a system that can be solved to about the maximum accuracy of the computer (a relative error of about 10^{-16} in normal Matlab).

Conversely, an extremely large determinant does not mean the system can be solved accurately. It may very well be impossible to get a meaningful solution to such a system.
The *correct* way to check whether you are going to get a good solution is check the so-called "condition number" of the matrix A. In Matlab you can get a suitable condition number using `cond(A)`. This number may not be too large. In Matlab that means that the condition number `cond(A)` should be several orders of magnitude less than 10^{16}:

```
% Correct, credit
condA=cond(A)
disp('cond(A) is much less than 1E16, so OK.')
```

In general the condition number should be several orders of magnitude less than the inverse of the relative error in floating point numbers. Note that 10^{16}, is the inverse of the Matlab relative error `eps(1)`.
The meaning of the condition number is as follows:

```
Definition:
   The condition number determines by what factor
    solving the system magnifies relative errors.
```

Note that floating point numbers are stored in Matlab with a relative error of 10^{-16}. So if the condition number is 10^{+16} or larger, the error in the solution may be 100% or more, just due to storing the numbers of the system in storage locations. In that case the solution is obviously meaningless. For the most accurate solution you would really like the condition number to be relatively small. (The smallest it can be is 1.)

```
disp(' ')
disp('Check the system the zero-credit way:')
disp('find the determinant:')
badBadBadDetA=det(A)
disp('In numerical methods, the determinant means:')
disp('- nothing if it is zero;')
disp('- nothing if it is small;')
disp('- nothing if it is finite;')
disp('- nothing if it is large.')

% the good way: check the condition number
disp(' ')
disp('Check the system the full-credit way:')
condA=cond(A)
disp('cond(A) is a lot smaller that 1.e16, so OK.')
disp(' ')
disp('But note that if the values of A and b have')
disp('measurement errors of just 1%, then the')
disp('computed x values might have relative errors')
```

```
disp('as high as 40%:')
relErrData=0.01
xRelErr=relErrData*condA
disp('Without checking the condition number, we would')
disp('have no clue of that!')
disp(' ')
disp('To rub it in, the following matrix, diag(-2,3,4):')
Diag=diag([-2 3 4])
disp('also has')
detDiag=det(Diag)
disp('but there is no big increase in the error solving')
disp('equations with this matrix:')
condDiag=cond(Diag)
```

```
Check the system the zero-credit way:
find the determinant:
badBadBadDetA = -24
In numerical methods, the determinant means:
- nothing if it is zero;
- nothing if it is small;
- nothing if it is finite;
- nothing if it is large.

Check the system the full-credit way:
condA =  37.939
cond(A) is a lot smaller that 1.e16, so OK.

But note that if the values of A and b have
measurement errors of just 1%, then the
computed x values might have relative errors
as high as 40%:
relErrData =  0.010000
xRelErr =  0.37939
Without checking the condition number, we would
have no clue of that!

To rub it in, the following matrix, diag(-2,3,4):
Diag =
Diagonal Matrix
  -2    0    0
   0    3    0
   0    0    4
also has
detDiag = -24
but there is no big increase in the error solving
```

```
equations with this matrix:
condDiag =  2
```

Solve the system, the correct way

Next if the system is indeed solvable according to the test above, solve it. The *wrong* way to do so is

```
% WRONG, zero credit
x = inv(A)*b
```

Computing an inverse matrix is *very* to *extremely* inefficient. It also tends to increase round-off errors. (What do you think, doing all these needless computations?)
The correct way to solve a generic system of equations in Matlab is using "left division"

```
% Correct, credit
x = A \ b     (left division: A\b instead of b/A)
```

If this is not the best way to solve your system, then you must learn a lot about linear algebra and numerical linear algebra before you can solve it yourself.

```
disp(' ')
A
b
disp(' ')
disp('Solve the system the zero-credit way:')
badInvA_Star_b=inv(A)*b

% the good way: check the condition number
disp(' ')
disp('Solve the system the full-credit way:')
disp('use "left division" (not b/A but A\b):')
x = A \ b
disp('This is the correct solution to the system of')
disp('equations as given, to at least 14 digits or so.')
```

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

```
Solve the system the zero-credit way:
badInvA_Star_b =
    1.0000
   -2.0000
    2.0000

Solve the system the full-credit way:
use "left division" (not b/A but A\b):
x =
    1
   -2
    2
This is the correct solution to the system of
equations as given, to at least 14 digits or so.
```

Problematic matrices

Consider now the modified system of equations

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 3 \\ 4x_1 + 5x_2 + 6x_3 &= 2 \\ 7x_1 + 8x_2 + 9x_3 &= 9 \end{aligned}$$

The only change is the additional $44x_1$ 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). We want to see what happens when we try to solve this matrix correctly in Matlab.
To form the new matrix, which we will call `ASing` in Matlab, 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:

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

In particular, the element in row 2 and column 1 of `ASing` is `ASing(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.

```
disp(' ')
disp('Let''s try a singular matrix now:')
% copy A into ASing
disp('After ASing=A:')
ASing=A
```

```
% change the 0 element in row 2 and column 1 into a 4.
ASing(2,1)=4;
disp('After ASing(2,1)=4:')
ASing

% check the condition number
disp('Check the condition number:')
condASing=cond(ASing)
disp('The condition number is excessive.')
disp('Even with its 1E-16 relative error, Matlab can')
disp('not find the solution to an acceptable error:')
xRelErrorDueToMatlab=condASing*eps(1)
disp('The 1E-16 relative error in A and b would predict')
disp('a maximum relative error in x of about 1,300%!')
disp('But, like here, a condition number of about 1E16')
disp('or more may simply mean the matrix is singular.')
disp('Then any numerical solution is meaningless.')

% Try solving anyway?
disp('Try xSing=A\B anyway? (Stupid):')
xSing = A \ b
disp('Nice numbers, but they are all wrong;')
disp('the exact solution is infinite!')
disp('Do not solve singular systems unless told so!')
```

```
Let's try a singular matrix now:
After ASing=A:
ASing =
   1   2   3
   0   5   6
   7   8   9
After ASing(2,1)=4:
ASing =
   1   2   3
   4   5   6
   7   8   9
Check the condition number:
condASing =    6.0262e+16
The condition number is excessive.
Even with its 1E-16 relative error, Matlab can
not find the solution to an acceptable error:
xRelErrorDueToMatlab =  13.381
The 1E-16 relative error in A and b would predict
a maximum relative error in x of about 1,300%!
But, like here, a condition number of about 1E16
```

```
or more may simply mean the matrix is singular.
Then any numerical solution is meaningless.
Try xSing=A\B anyway? (Stupid):
xSing =
    1
   -2
    2
Nice numbers, but they are all wrong;
the exact solution is infinite!
Do not solve singular systems unless told so!
```

MATRIX MANIPULATIONS

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

```
disp(' ')
disp('MATRIX MANIPULATIONS:')
```

MATRIX MANIPULATIONS:

Transposes

The "transpose" of a matrix A is indicated by A^{T}. The columns in A becomes rows in A^{T} and vice-versa:

```
Definition:
  Transposing swaps rows and columns.
```

As we already saw

```
Important:
  To transpose in Matlab, append a quote.
```

```
disp(' ')
disp('Create a transpose T using a '' at the end:')

% try it for vector b
b=b
bT=b'
disp('Note that a second transpose undoes the first:')
bTT=bT'

% try it for matrix A
A=A
```

```
AT=A'
ATT=AT'
```

```
Create a transpose T using a ' at the end:
b =
    3
    2
    9
bT =
    3    2    9
Note that a second transpose undoes the first:
bTT =
    3
    2
    9
A =
    1    2    3
    0    5    6
    7    8    9
AT =
    1    0    7
    2    5    8
    3    6    9
ATT =
    1    2    3
    0    5    6
    7    8    9
```

Matrix multiplication

So far, we never actually checked that the solution $\vec{x}$ that we found for $A\vec{x} = \vec{b}$ was any good. To check it, we can tell Matlab to multiply matrix A with column vector $\vec{x}$ and compare that to $\vec{b}$. But to do so, we cannot multiply A and $\vec{x}$ using the usual "elementwise" array multiplication '.*'. Instead we must us a simple '*' without the point to multiply A and $\vec{x}$. Then Matlab will multiply the two in a special way that is called "matrix multiplication". (True, $\vec{x}$ is a column vector, but remember that any column vector is also a matrix with just one column.) The key thing to remember is:

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

In particular, if we multiply matrices

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad \text{and} \qquad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

together with $*$, we get

$$A\vec{x} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1x_1 + 2x_2 + 3x_3 \\ 0x_1 + 5x_2 + 6x_3 \\ 7x_1 + 8x_2 + 9x_3 \end{pmatrix}$$

Note that the first of the three components of the result vector is found as a dot product between row 1 of A and the single column 1 of vector x. This should be the same as the first component of $\vec{b}$, which is 3. Similarly the second component is a dot product between row 2 of A and the single column 1 of vector x, and should be 2. And similarly for the third component, which should be 9.

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

From the above, it is obvious that

```
Important:
  Rows and columns involved in matrix multiplication
      must have the same number of elements.
```

Also note the following

```
Important:
  If you multiply arrays together with '*', Matlab
  will try to do matrix multiplication.
```

To get Matlab to do elementwise multiplication, use '.*' instead of '*'. Similarly use '.^' instead of '^' and './' instead of '/' as operators on arrays to avoid matrix interpretation.

```
disp(' ')
disp('Let''s try some matrix multiplications:')
disp(' ')
disp('Let''s repeat our check that A x = b:')
% evaluate A x
disp('Find out what A*x is (do NOT use .* here):')
A_Star_x=A*x
% compare with b
disp('This should be b:')
b=b
% evaluate the error vector A x - b
disp('See what the difference (error vector) is:')
A_Star_x_Minus_b=A*x-b
disp('Seems good!')
```

```
% evaluate the maximum error
fprintf('Maximum difference between A x and b: %E.1',...
        max(abs(A*x-b)))

disp(' ')
% evaluate ASing xSing
disp('Find out what ASing*xSing is:')
ASing_Star_xSing=ASing*xSing
% compare with b
b=b
% evaluate the error vector A x - b
disp('See what the difference (error vector) is:')
ASing_Star_xSing_Minus_b=ASing*xSing-b
disp('This error may be small even if xSing is no good!')
% evaluate the maximum error
fprintf('Maximum difference: %.1E\n',...
        max(abs(ASing*xSing-b)))
```

```
Let's try some matrix multiplications:

Let's repeat our check that A x = b:
Find out what A*x is (do NOT use .* here):
A_Star_x =
    3
    2
    9
This should be b:
b =
    3
    2
    9
See what the difference (error vector) is:
A_Star_x_Minus_b =
    0
    0
    0
Seems good!
Maximum difference between A x and b: 0.000000E+00.1
Find out what ASing*xSing is:
ASing_Star_xSing =
    3
    6
    9
b =
    3
```

```
    2
    9
See what the difference (error vector) is:
ASing_Star_xSing_Minus_b =
    0
    4
    0
This error may be small even if xSing is no good!
Maximum difference: 4.0E+00
```

Play around with matrix multiplication

```
% let's play a bit with matrix multiplication
disp(' ')
disp('How about some more multiplications?')
A=A
x=x
b=b
B = [x b b x]
disp('Each column in B is multiplied to A separately:')
A_Star_B=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(' ')
disp('If b is a column vector:')
b=b
disp('then the "transpose" of b is a row vector:')
bT=b'

% matrix product of bT and x
disp(' ')
disp('"Dot product" bT*x produces b dot x:')
bT=b'
x=x
bT_Star_x=b'*x
disp('Can also be obtained using the dot function:')
bDotx=dot(b,x)
disp('"Dot product" bT*b is the square length of b:')
bT=b'
b=b
```

```
bT_Star_b=b'*b
disp('Can also be obtained using the norm function:')
bNormSquare=norm(b)^2

% "outer" product of b and x
disp(' ')
disp('"Outer product" b*bT produces a matrix:')
b=b
bT=b'
b_Star_bT=b*b'
disp('And so does b*xT:')
b=b
xT=x'
b_Star_xT=b*x'

% b^2 fails: row length 1 times column length 3 is bad
disp(' ')
disp('Unlike bT*x and b*xT, b*x and bT*xT are illegal.')
disp('And so is b^2.')
disp('None of these are row-column as they should be.')

% "elementwise" computation of x
disp(' ')
disp('b.*b is multiplied elementwise:')
b=b
b_PtStar_b=b.*b
disp('and so is b.^2:')
b_PtSquare=b.^2
disp('The same for bT:')
bT_PtStar_bT=b'.*b'
bT_PtSquare=b'.^2
disp('Still another way to get the dot product:')
b_PtStar_x=b.*x
sum_b_PtStar_x=sum(b.*x)

% some more multiplications
disp(' ')
disp('The same ideas apply to matrices:')
A=A
AT_Star_A=A'*A
disp('For a square matrix A, A*A and A^2 work:')
A_Star_A=A*A
A_Square=A^2
disp('Elementwise operations on matrices:')
A_PtStar_A=A.*A
A_PtSquare=A.^2
```

```
How about some more multiplications?
A =
    1    2    3
    0    5    6
    7    8    9
x =
    1
   -2
    2
b =
    3
    2
    9
B =
    1    3    3    1
   -2    2    2   -2
    2    9    9    2
Each column in B is multiplied to A separately:
A_Star_B =
     3    34    34     3
     2    64    64     2
     9   118   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

"Dot product" bT*x produces b dot x:
bT =
    3    2    9
x =
    1
   -2
```

```
    2
bT_Star_x =  17
Can also be obtained using the dot function:
bDotx =  17
"Dot product" bT*b is the square length of b:
bT =
    3    2    9
b =
    3
    2
    9
bT_Star_b =  94
Can also be obtained using the norm function:
bNormSquare =  94.000

"Outer product" b*bT produces a matrix:
b =
    3
    2
    9
bT =
    3    2    9
b_Star_bT =
     9     6    27
     6     4    18
    27    18    81
And so does b*xT:
b =
    3
    2
    9
xT =
    1   -2    2
b_Star_xT =
     3    -6     6
     2    -4     4
     9   -18    18

Unlike bT*x and b*xT, b*x and bT*xT are illegal.
And so is b^2.
None of these are row-column as they should be.

b.*b is multiplied elementwise:
b =
    3
    2
```

```
    9
b_PtStar_b =
     9
     4
    81
and so is b.^2:
b_PtSquare =
     9
     4
    81
The same for bT:
bT_PtStar_bT =
     9     4    81
bT_PtSquare =
     9     4    81
Still another way to get the dot product:
b_PtStar_x =
     3
    -4
    18
sum_b_PtStar_x =  17

The same ideas apply to matrices:
A =
    1    2    3
    0    5    6
    7    8    9
AT_Star_A =
     50     58     66
     58     93    108
     66    108    126
For a square matrix A, A*A and A^2 work:
A_Star_A =
     22     36     42
     42     73     84
     70    126    150
A_Square =
     22     36     42
     42     73     84
     70    126    150
Elementwise operations on matrices:
A_PtStar_A =
     1     4     9
     0    25    36
    49    64    81
A_PtSquare =
```

```
     1     4     9
     0    25    36
    49    64    81
```

Dot products

> Important:
> In matrix multiplication terms, any dot product between vectors must be row–column.

So, to take the dot product between two column vectors, you must put a quote on the first vector in the product.

```
% examples:
disp(' ')
disp('example dot products:')
v=[3;
   4]
w=[1;
   2]
disp('The dot product of vectors v and w is 11:')
vT_Star_w=v'*w
wT_Star_v=w'*v
disp('The square length of vector v is 25:')
vT_Star_v=v'*v
disp('The length of a vector v is sqrt(v dot v):')
sqrt_vT_Star_v=sqrt(v'*v)
```

```
example dot products:
v =
   3
   4
w =
   1
   2
The dot product of vectors v and w is 11:
vT_Star_w =  11
wT_Star_v =  11
The square length of vector v is 25:
vT_Star_v =  25
The length of a vector v is sqrt(v dot v):
sqrt_vT_Star_v =  5
```

Root-mean-square errors

One type of error you often want to find is not the maximum error, but the "root-mean-square" (RMS) one.

```
% the error in ASing xSing = b
disp(' ')
disp('The errors in ASing x = bSing were:')
errorVec=ASing*xSing-b
% evaluate the maximum error
fprintf('Maximum difference between ASing xSing and b:
    %.1E\n',...
        max(abs(errorVec)))
% evaluate the sum of the square errors
disp('A dot product gives the sum of the square errors:')
summedSquareErrors=errorVec'*errorVec
% get the number of errors
disp('The size function says how many errors there are:')
sizeErrorVec=size(errorVec)
totalSizeErrorVec=prod(size(errorVec))
% to get the average, divide by the number of errors
disp('Use that to get the average square error:')
meanSquareError=...
    errorVec'*errorVec/prod(size(errorVec))
% now take square root
disp('and then take square root to get the RMS error:')
disp('sqrt(errorVec''*errorVec/prod(size(errorVec)))')
rmsError=...
    sqrt(errorVec'*errorVec/prod(size(errorVec)))
disp('Another way to get this:')
disp('sqrt(dot(errorVec,errorVec)/prod(size(errorVec)))')
rmsError=...
    sqrt(dot(errorVec,errorVec)/prod(size(errorVec)))
disp('Still another way to get this:')
disp('sqrt(sum(errorVec.^2)/prod(size(errorVec)))')
rmsError=...
    sqrt(sum(errorVec.^2)/prod(size(errorVec)))
% print it out neatly
fprintf('RMS difference between A x and b: %.1E\n',...
        rmsError)
```

```
The errors in ASing x = bSing were:
errorVec =
    0
    4
    0
```

```
Maximum difference between ASing xSing and b: 4.0E+00
A dot product gives the sum of the square errors:
summedSquareErrors =  16
The size function says how many errors there are:
sizeErrorVec =
   3   1
totalSizeErrorVec =  3
Use that to get the average square error:
meanSquareError =  5.3333
and then take square root to get the RMS error:
sqrt(errorVec'*errorVec/prod(size(errorVec)))
rmsError =  2.3094
Another way to get this:
sqrt(dot(errorVec,errorVec)/prod(size(errorVec)))
rmsError =  2.3094
Still another way to get this:
sqrt(sum(errorVec.^2)/prod(size(errorVec)))
rmsError =  2.3094
RMS difference between A x and b: 2.3E+00
```

Special 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. The symbol for a zero matrix is typically Z.
A *unit matrix* (or *identity 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. The symbol for a unit matrix is typically "I".
A *symmetric matrix* is the same as its transpose. So A is symmetric iff $A^{\mathrm{T}} = A$. Symmetric matrices occur in many highly important engineering applications.

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

% make a bigger matrix to test
disp('First, let''s make a bigger matrix for testing:')
Big=[A AT]
disp('The size of a matrix is [rows columns]:')
BigSize=size(Big)

% a zero matrix consists of all zeros
disp(' ')
disp('Create a zero matrix for Big+Z or Z+Big:')
Z=zeros(3,6)
```

```
disp('The correct way to do this: Z=zeros(size(Big)):')
Z=zeros(size(Big))
disp('Adding a zero matrix makes no difference:')
Big=Big
BigPlusZ=Big+Z
ZPlusBig=Z+Big
disp('Create a square zero matrix for Z*Big:')
[mBig nBig]=size(Big)
disp('The correct way to get Z: Z=zeros(mBig):')
Z=zeros(mBig)
disp('Multiplying by a zero matrix produces zero:')
Z_Star_Big=Z*Big
disp('A "zero vector" for Z*Big: Z=zeros(1,mBig):')
Z=zeros(1,mBig)
disp('Multiplying by a zero matrix produces zero:')
Z_Star_Big=Z*Big
disp('Create a square zero matrix for Big*Z:')
[mBig nBig]=size(Big)
disp('The correct way to get Z: Z=zeros(nBig):')
Z=zeros(nBig)
disp('Multiplying by a zero matrix produces zero:')
Big_Star_Z=Big*Z
disp('Create a zero vector for Big*Z: Z=zeros(nBig,1):')
Z=zeros(nBig,1)
disp('Multiplying by a zero matrix produces zero:')
Big_Star_Z=Big*Z

% a unit matrix has ones on the main diagonal
disp(' ')
disp('Create a unit matrix for I*Big:')
I=eye(3)
disp('The correct way to do this:')
[mBig nBig]=size(Big)
disp('The correct way to get I: I=eye(mBig):')
I=eye(mBig)
disp('Multiplying by a unit matrix makes no difference:')
Big=Big
I_Star_Big=I*Big
b=b
I_Star_b=I*b
disp('Create a unit matrix for Big*I:')
[mBig nBig]=size(Big)
disp('The correct way to get I: I=eye(nBig):')
I=eye(nBig)
disp('Multiplying by a unit matrix makes no difference:')
Big=Big
```

```
Big_Star_I=Big*I

% look at a symmetric matrix
disp(' ')
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:
First, let's make a bigger matrix for testing:
Big =
   1   2   3   1   0   7
   0   5   6   2   5   8
   7   8   9   3   6   9
The size of a matrix is [rows columns]:
BigSize =
   3   6

Create a zero matrix for Big+Z or Z+Big:
Z =
   0   0   0   0   0   0
   0   0   0   0   0   0
   0   0   0   0   0   0
The correct way to do this: Z=zeros(size(Big)):
Z =
   0   0   0   0   0   0
   0   0   0   0   0   0
   0   0   0   0   0   0
Adding a zero matrix makes no difference:
Big =
   1   2   3   1   0   7
   0   5   6   2   5   8
   7   8   9   3   6   9
BigPlusZ =
   1   2   3   1   0   7
   0   5   6   2   5   8
   7   8   9   3   6   9
ZPlusBig =
   1   2   3   1   0   7
   0   5   6   2   5   8
   7   8   9   3   6   9
Create a square zero matrix for Z*Big:
```

```
mBig =  3
nBig =  6
The correct way to get Z: Z=zeros(mBig):
Z =
   0   0   0
   0   0   0
   0   0   0
Multiplying by a zero matrix produces zero:
Z_Star_Big =
   0   0   0   0   0   0
   0   0   0   0   0   0
   0   0   0   0   0   0
A "zero vector" for Z*Big: Z=zeros(1,mBig):
Z =
   0   0   0
Multiplying by a zero matrix produces zero:
Z_Star_Big =
   0   0   0   0   0   0
Create a square zero matrix for Big*Z:
mBig =  3
nBig =  6
The correct way to get Z: Z=zeros(nBig):
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
Multiplying by a zero matrix produces zero:
Big_Star_Z =
   0   0   0   0   0   0
   0   0   0   0   0   0
   0   0   0   0   0   0
Create a zero vector for Big*Z: Z=zeros(nBig,1):
Z =
   0
   0
   0
   0
   0
   0
Multiplying by a zero matrix produces zero:
Big_Star_Z =
   0
   0
```

```
   0

Create a unit matrix for I*Big:
I =
Diagonal Matrix
   1   0   0
   0   1   0
   0   0   1
The correct way to do this:
mBig =  3
nBig =  6
The correct way to get I: I=eye(mBig):
I =
Diagonal Matrix
   1   0   0
   0   1   0
   0   0   1
Multiplying by a unit matrix makes no difference:
Big =
   1   2   3   1   0   7
   0   5   6   2   5   8
   7   8   9   3   6   9
I_Star_Big =
   1   2   3   1   0   7
   0   5   6   2   5   8
   7   8   9   3   6   9
b =
   3
   2
   9
I_Star_b =
   3
   2
   9
Create a unit matrix for Big*I:
mBig =  3
nBig =  6
The correct way to get I: I=eye(nBig):
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
```

```
Multiplying by a unit matrix makes no difference:
Big =
    1    2    3    1    0    7
    0    5    6    2    5    8
    7    8    9    3    6    9
Big_Star_I =
    1    2    3    1    0    7
    0    5    6    2    5    8
    7    8    9    3    6    9

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
```

Parts of matrices

When we created singular matrix `ASing`, we already saw that you can address a single number in a matrix using `(ROW,COLUMN)`. For example, the element in row 2 and column 1 of `ASing` was `ASing(2,1)`.
You can also address multiple elements in a matrix by using `START:END` constructs. Below are some examples.

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

% use the big matrix
Big=Big
sizeBig=size(Big)

% taking parts of rows out (note row-column!)
disp('row2part=Big(2,2:4):')
row2part=Big(2,2:4)

% taking an entire row out
disp('row2all=Big(2,:):')
row2all=Big(2,:)
disp('Bad, as less readable: row2allBAD=Big(2,1:end):')
row2allBAD=Big(2,1:end)
disp('Worse (Big may change): row2allWORSE=Big(2,1:6):')
```

```
row2allWORSE=Big(2,1:6)

% taking columns out of a matrix (important)
disp('col4all=Big(:,4):')
col4all=Big(:,4)

% taking three columns out at the same time
disp('col345all=Big(:,3:5):')
col345all=Big(:,3:5)

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

```
Try taking parts out of matrices:
Big =
    1    2    3    1    0    7
    0    5    6    2    5    8
    7    8    9    3    6    9
sizeBig =
    3    6
row2part=Big(2,2:4):
row2part =
    5    6    2
row2all=Big(2,:):
row2all =
    0    5    6    2    5    8
Bad, as less readable: row2allBAD=Big(2,1:end):
row2allBAD =
    0    5    6    2    5    8
Worse (Big may change): row2allWORSE=Big(2,1:6):
row2allWORSE =
    0    5    6    2    5    8
col4all=Big(:,4):
col4all =
    1
    2
    3
col345all=Big(:,3:5):
col345all =
    3    1    0
    6    2    5
```

```
    9     3     6
Let's delete column 2 in AT:
ATDel =
    1     0     7
    2     5     8
    3     6     9
ATDel(:,2)=[]:
ATDel =
    1     7
    2     8
    3     9
```

EIGENVALUES AND EIGENVECTORS

A vector $\vec{e}$ is an eigenvector of a given square matrix A if $\vec{e}$ is nonzero and:

$$A\vec{e} = \lambda\vec{e}$$

where λ 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.

```
disp(' ')
disp('EIGENVALUES AND EIGENVECTORS:')

% see what is available to do so
disp(' ')
disp('lookfor eigenvalue')
```

```
EIGENVALUES AND EIGENVECTORS:

lookfor eigenvalue
```

A simple example

```
disp(' ')
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]
disp('Get the eigenvalues with lambda=eig(S):')
lambda=eig(S)
disp('Get the eigenvectors with [E Lambda]=eig(S):')
[E Lambda]=eig(S)
disp('Separate the eigenvalues out as lambda(N):')
lambda1=lambda(1)
lambda2=lambda(2)
lambda3=lambda(3)
disp('Or separate the eigenvalues out as Lambda(N,N):')
lambda1=Lambda(1,1)
lambda2=Lambda(2,2)
lambda3=Lambda(3,3)
disp('Separate out the eigenvectors as E(:,N):')
e1=E(:,1)
e2=E(:,2)
e3=E(:,3)
disp(' ')

% let's check that Matlab found the right vectors
disp('Check that S eN = lambdaN eN:')
S_Star_e1=S*e1
lambda1e1=lambda1*e1
errors1=S*e1-lambda1*e1
maxError1=max(abs(errors1))
S_Star_e2=S*e2
lambda2e2=lambda2*e2
errors2=S*e2-lambda2*e2
maxError2=max(abs(errors2))
S_Star_e3=S*e3
lambda3e3=lambda3*e3
errors3=S*e3-lambda3*e3
maxError3=max(abs(errors3))
```

```
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
Get the eigenvalues with lambda=eig(S):
lambda =
   -1
    0
    1
Get the eigenvectors with [E Lambda]=eig(S):
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 the eigenvalues out as lambda(N):
lambda1 = -1
lambda2 = 0
lambda3 =  1
Or separate the eigenvalues out as Lambda(N,N):
lambda1 = -1
lambda2 = 0
lambda3 =  1
Separate out the eigenvectors as E(:,N):
e1 =
   -0.70711
    0.70711
    0.00000
e2 =
    0
    0
    1
e3 =
    0.70711
    0.70711
    0.00000

Check that S eN = lambdaN eN:
S_Star_e1 =
    0.70711
   -0.70711
```

```
    0.00000
lambda1e1 =
    0.70711
   -0.70711
   -0.00000
errors1 =
    0
    0
    0
maxError1 = 0
S_Star_e2 =
    0
    0
    0
lambda2e2 =
    0
    0
    0
errors2 =
    0
    0
    0
maxError2 = 0
S_Star_e3 =
    0.70711
    0.70711
    0.00000
lambda3e3 =
    0.70711
    0.70711
    0.00000
errors3 =
    0
    0
    0
maxError3 = 0
```

About symmetric matrices

As already noted, a matrix A is symmetric iff it equals its transpose; $A^{\mathrm{T}} = A$. 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. They are the unit vectors along the "principal axes" of the matrix.

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.

```
disp(' ')
disp('Since matrix S is symmetric, the eigenvectors')
disp('should have length 1:')
% the length of the vectors can be computed using norm
disp('Lengths of the eigenvectors using norm:')
e1Norm=norm(e1)
e2Norm=norm(e2)
e3Norm=norm(e3)

% or dot the vector with itself and take square root
disp('Lengths of the eigenvectors using dot products:')
sqrt_e1T_Star_e1=sqrt(e1'*e1)
sqrt_e2T_Star_e2=sqrt(e2'*e2)
sqrt_e3T_Star_e3=sqrt(e3'*e3)

% vectors are orthogonal if their dot product is zero
disp('Since matrix S is symmetric, the eigenvectors')
disp('should be mutually orthogonal (zero dot product):')
e1T_Star_e2=e1'*e2
e2T_Star_e3=e2'*e3
e3T_Star_e1=e3'*e1
```

```
Since matrix S is symmetric, the eigenvectors
should have length 1:
Lengths of the eigenvectors using norm:
e1Norm =  1
e2Norm =  1
e3Norm =  1
Lengths of the eigenvectors using dot products:
sqrt_e1T_Star_e1 =  1
sqrt_e2T_Star_e2 =  1
sqrt_e3T_Star_e3 =  1
Since matrix S is symmetric, the eigenvectors
should be mutually orthogonal (zero dot product):
e1T_Star_e2 = 0
e2T_Star_e3 = 0
e3T_Star_e1 = 0
```

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.
Use of `linsolve` also allows you to get a (I presume approximate and, for the experts, L1) condition number. That may be of interest for very big systems, as finding cond(A) may take nontrivially more effort than actually solving the system. But I cannot find info in the Matlab documentation on the actual condition number returned.
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.
If not using Matlab, the normal efficient way to solve equations will likely be referred to as "LU decomposition". If you have a band matrix, look for a dedicated LU-decomposition subroutine for those.

End lesson 5