5 LINEAR ALGEBRA

Contents

```
% make sure the workspace is clear
if ~exist('____code____','var') ; clear ; end
% reduce needless whitespace
format compact
% reduce irritations (pausing and buffering)
more off
% start a diary (in the actual lecture)
%diary lectureN.txt

% Tell the students to save their work space before
% leaving: save lectureN
```

LESSON SUMMARY

There are three main parts to this lesson.

The first main part is the solution of a linear system of equations for a set of unknowns. The number of equations must equal the number of unknowns. To solve a given linear system of equations with Matlab, perform the following steps:

1. First clean up the system of equations as needed. The terms of the form of a coefficient times an unknown should be in the left hand sides, with the unknowns ordered and vertically lined up. The terms without any unknowns should be in the right hand sides.

2. Put the coefficients of the unknowns in the left hand sides of the equations in a two-dimensional Matlab array, i.e. a table. This 2D array is called the "coefficient matrix". It will be denoted by A in this summary, or `A` in Matlab. In terms of Matlab array row and column numbers, for the component `A(rowNo,colNo)` of `A`, the row number `rowNo` should equal the equation number and the column number `colNo` should equal the unknown number.

   Put the right hand sides of the equations in a one-dimensional Matlab column array, i.e. a column of numbers. This 1D column array is called the "right-hand-side vector". It will be denoted by $\vec{b}$ in this summary, or `b` in Matlab. In terms of Matlab row numbers, for the component `b(rowNo)` of `b`, the row number `rowNo` should equal the equation number.

3. Before trying to solve the system, you should first check that the system has a reasonable solution in the first place. Not all systems do. To check, evaluate the "condition number" of matrix A. You can get this number in Matlab as `cond(A)`. The CliffsNotes version is now that if the condition

number turns out to be comparable to 10^{16} or even larger, the solution to the system found by Matlab will be crap. It is then unjustified to proceed with solving the system.

4. If the condition number is nowhere as big as 10^{16}, you can meaningfully solve the system for the unknowns using Matlab. Matlab will put the values of the unknowns in a one-dimensional column vector, called the "solution vector". It will be denoted by $\vec{x}$ here, or `x` in Matlab. Matlab allows you to find this vector using "left division", as in `x=A\b`. The values of the unknowns in `x` should be reasonably accurate assuming that you gave Matlab the correct equations to solve. So then you are done.

5. But suppose that some constants in your system of equations were, say, measured rather than exact. Then these will have errors and that will cause errors in the solution. You should then examine whether these errors might be serious. To do so, you will first need to ballpark the maximum relative error in the equations you gave to Matlab. Then multiply that maximum relative error in the equations by the condition number to get the maximum possible relative error in the solution. Multiply by 100 to get this error in percent, and then decide whether you can live with that kind of error. If not, investigate.

The second main part of this lesson is to learn some basic terms and manipulations in linear algebra. In particular:

- You should know that if you put a quote on an matrix in Matlab, it "transposes" the matrix, i.e. it turns rows into columns and vice-versa. (Actually, if the matrix is complex, the quote also swaps the sign of $\sqrt{-1}$, but we will only look at real matrices.)

- You should know that in linear algebra vectors are considered to be a special form of matrices in which either the number of rows is 1 (a row vector) or the number of columns is 1 (a column vector).

- You should know that multiplying matrices in linear algebra is always *"row-column"*; it always takes *dot* products between row and column vectors. In those terms, the system of equations discussed earlier can be written in the form $A\vec{x} = \vec{b}$. Here the matrix multiplication $A\vec{x}$ takes dot products between the rows in matrix A and the column vector $\vec{x}$.

  You get Matlab to do matrix multiplication (as opposed to elementwise multiplication) by *not* putting a point before the $*$: `A*x` instead of `A.*x`. The same for $\hat{}$ and /. (The "right division" in the Matlab assignment statement `x=b/A` makes `x` the solution to the equation `x*A=b`, or $\vec{x}A = \vec{b}$).

  You should also know that matrix multiplication is *not* commutative: `A*B` is normally not the same as `B*A`.

- You should still know from physics that the dot product of a vector with itself gives the square length of the vector. And you should know that if the dot product of two different vectors is zero, then these two vectors are orthogonal to each other.

- You should also know that Matlab calls the length of a vector the "norm" of the vector. This idiocy allows Matlab to use the term "length" for something else that is completely useless.

- You should know that a "square" matrix has the same number of rows as columns.

- You should know that the "main diagonal" of a matrix consists of the elements for which the row number equals the column number. It starts at the top-left corner and goes down to (for a square matrix) the right-bottom corner.

- You should know how various special matrices look and what they do: zero matrices, unit matrices, matrices consisting of ones, and symmetric matrices.

- You should know how to take parts out of matrices, and how to delete parts of matrices.

- You should *NOT* know what a determinant is or what an inverse matrix is. These concepts are far worse than useless in the numerical work that you can do yourself. Forget whatever you know about them right now. Except remember one thing: if you are using a determinant or an inverse matrix, you are doing it wrong.

The third and final main part of this lesson is about eigenvalues and eigenvectors. Here we are interested in finding solutions to a set of equations of the form

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

or in Matlab terms `A*e==lambda*e`. Here A is again a matrix, and $\vec{e}$ is again a vector of unknowns. The big difference from a normal system of equations is that the right hand side is not a *given* vector but some unknown multiple λ, or `lambda` in Matlab, times the vector of unknowns. For any solution to the above equation in which $\vec{e}$ is not completely zero (which would be trivial), the number λ is called the "eigenvalue" and vector $\vec{e}$ the eigenvector.

Eigenvectors are not unique; if a vector $\vec{e}$ satisfies the eigenvalue problem above, then, say, so does $2\vec{e}$. Just substitute it in and divide out the 2. However, if n is the number of unknowns (which is also the number of equations), then normally it is possible to find exactly n fundamentally different eigenvectors $\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_n$ with corresponding eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$. (In special cases there may be less than n different eigenvectors, but never more than n.)

There is no time to go into the details of eigenvalue problems in this lesson but there are some things you should be able to do:

1. Given an eigenvalue problem, you should be able to find the matrix A (which is just like for the linear system of equations earlier), as well as identify what is the eigenvalue λ in terms of the variables in the given problem.

2. Then you should be able to find an array `lambdaVals` with the eigenvalues of the matrix in it using an `lambdaVals = eig(A)` type assignment statement. You should also be able to find both a matrix E, or `E` in Matlab, with the eigenvectors in it and a matrix Λ, or `Lambda` in Matlab, with the eigenvalues in it (on the main diagonal), using an `[E Lambda] = eig(A)` type assignment statement.

3. You should be able to take the individual eigenvectors and eigenvalues out of these matrices.

4. You should be able to check if matrix A is symmetric. And you should know that if it is, the eigenvectors found by Matlab will be of length one and mutually orthogonal. (That is much like the unit vectors $\hat{\imath}$, $\hat{\jmath}$, and $\hat{k}$ of a Cartesian coordinate system are of length one and mutually orthogonal.)

5. You should know how to check the length and the mutual orthogonality of the eigenvectors using the `norm` and `dot` functions, as well as by using direct matrix multiplication.

Key areas of the online book

Before the first lecture, in the online book do:

- 8.1 Dimensional properties of arrays: do PA 8.1.1. I doubt you will need the examples and figures.
- 19.1 Vectors: skip the stuff after PA 19.1.5.
- 19.2 Matrices: skip the stuff from the "Special Square Matrices" subsection onwards.
- 19.7 Linear systems: all.
- 19.8 Square matrices - Solving Ax = b: all.

Before the second lecture, in the online book do:

- 5.1 1D element-wise arithmetic operators: all except the final CA 5.1.3. Warning: before doing this horribly written section, first refresh your memory about arrays at the end of lesson1.

- 6.5 Summation function: skip CA.
- 7.3 Concatenation: all.
- 7.4 Multi-element 2D array indexing: skip the stuff after PA 7.4.3.
- 7.5 Indexing using a single colon: do PA 7.5.1.
- 8.2 Elementary 2D arrays: do PA 8.2.1-2. I doubt you will need the examples and figures.
- 19.3: Matrix transpose: do PA 19.3.1 all and PA 19.3.2 question 1 only.
- 19.4: Matrix calculations: Skip PA 19.4.10-end.

Note that while the online book has a section 20.11 on eigenvalues and eigenvectors, you will find it is far too mathematical for you to understand. All you will need to know is that eigenvalues and eigenvectors exist, what they are, and how Matlab function `eig` can find them for you.

SOLVING LINEAR SYSTEMS OF EQUATIONS

In the next subsections, 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. But whether the system is small or large, the number of equations should be the same as the number of unknowns.

The example problem that we want to solve

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

$$x_1 = 3 - 2x_2 - 3x_3 \qquad 5x_2 + 6x_3 - 2 = 0 \qquad 7x_1 + 8x_2 + 9x_3 = 9$$

for the three unknowns x_1, x_2, and x_3.

Cleaning up the equations

The first step is to clean up the system. Move all the terms that take the form of a constant coefficient times an unknown to the left hand sides of the equations. Move the remaining terms, that do not involve any unknown, to the right hand sides of the equations.

Also order the unknowns and line up the terms vertically so that the same unknowns in different equations are in the same column.

Also x_1 is the same as $1x_1$ and no x_1 is the same as $0x_1$. So you get:

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

for the cleaned up system.

The coefficient matrix and right hand side vector

The constants in the left hand sides of the equations, the coefficients of the unknowns, form a "coefficient matrix" (table of numbers), call it A. Similarly the right hand sides of the equations form a "right hand side (column) vector", call it $\vec{b}$:

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

Note that the coefficients of unknown 1 all go into the first column of the matrix, those of unknown 2 in the second column, etcetera.

Also remember that it is customary in linear algebra to use capitalization for the names of matrices and lowercase for vectors.

In Matlab, a "matrix" is just a different name for a two-dimensional array (numbers arranged in rows and columns). And in Matlab a "column vector" is just a different name for a one-dimensional column array (which Matlab treats as a two-dimensional array with just a single column).

The most straightforward way to create such arrays is by writing out the numbers that they contain within square brackets. This is shown below for A:

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

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

Note that you can replace newlines inside the brackets with semicolons:

```
% create A, alternative form
A=[1 2 3; 0 5 6; 7 8 9]
```

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

However, that code is much less readable. You are responsible for making your code easy to read for everybody.

Right-hand-side vector $\vec{b}$ can be created similarly:

```
% create b
b = [3
     2
     9]
```

```
b =
   3
   2
   9
```

I would say that the more concise:

```
% create b, alternative form
b = [3  2  9]'
```

```
b =
   3
   2
   9
```

would also be OK and save paper.

The online book disagrees. It says people might not see the quote. I never have that trouble. The book also says people might read "3 2" above as "32", so you should separate using commas. I never have that trouble. I am more likely to read "3,2" as "3.2". And "3, 2" as "3; 2" or vice-versa. I also think A=[1,2,3;0,5,6;7,8,9] is a mess.

Check whether the system is solvable

To solve any system of equations properly in Matlab, first you must check that Matlab will be able to find an accurate solution to the system in the first place. That might not be the case. If not, you should not proceed; any solution you would compute would be grossly misleading.

Now do not start thinking of the "determinant" of Precalculus. The *only* thing that you cannot forget about determinants in numerical work is:

WARNING: *If you are using a determinant, you are doing it wrong!*

Ignore that the online book does it; that they are clueless does not mean you should be too.

The correct way to check whether an accurate solution can be computed is to check the so-called "condition number" of the matrix A. In Matlab you can get the condition number using `cond(A)`.

The condition number may not be too large. In Matlab it should be several orders of magnitude less than 10^{16}. If the condition number becomes comparable to 10^{16} or even bigger, call it quits. The results will be crap.

Consider *why* this is the case. First of all,

DEFINITION: *The condition number is the maximum factor that relative errors can magnify in solving the system.*

Normally in Matlab numbers are stored to only about 16 (decimal) significant digits, starting from the first nonzero one. That corresponds to a typical relative error of about 10^{-16}. (More accurately, the maximum relative error due to storing a normal number can be found as `eps` in Matlab.) The stored *system to solve* is only accurate to this 10^{-16} relative error. To ballpark the corresponding relative error in the *solution*, multiply by the condition number. So if the condition number is 10^{16}, then the relative error in the solution ballparks at $10^{-16} \times 10^{16} = 1$ or 100%. Clearly, if your solution has 100% error, it is no good.

(Note that while technically the condition number only gives the *maximum* increase in relative error, i.e. the worst-case scenario, in real life you are never lucky. To imagine why, think of all the errors made due to storing the intermediate numbers in the solution process. Condition numbers this large are also not accurate themselves.)

Check the condition number:

```
% find the condition number
condA=cond(A)
```

```
condA =   37.939
```

Since `cond(A)` is much smaller than 10^{-16}, Matlab should be able to find a very accurate solution to the system.

Solve the system

Next, *only if the system is accurately solvable according to the test above,* solve it.

Now do not start thinking of the "inverse matrix" of Precalculus. The *only* thing you cannot forget about inverse matrices in numerical work is:

WARNING: *If you are using an inverse matrix, you are doing it wrong!*

The correct way to solve a generic system of equations in Matlab is using "left division"

```
x = A \ b    % left division: A \ b instead of b / A
```

This will put the computed values of the unknowns in the "solution (column) vector x" (or $\vec{x}$ in mathematics).

```
% solve the system
x = A \ b
```

```
x =
     1
    -2
     2
```

This should be the correct solution to the system of equations as given, to about 14 digits.

(Actually, the solution above is exact, but that has to do with everything being an integer here.)

Check whether the solution will be accurate enough

Even if the condition number is not that large at all, like the condition number 38 above, there is another potential source of errors you must watch out for. If the equations you give Matlab are not exact but have errors, then that will introduce *additional* errors in the computed solution. These errors are *very likely* to be a lot worse than the errors that Matlab itself introduces in the numbers. For example, some of the constants in the equations you give Matlab may be measured experimentally. There are *very few* known constants in nature that are measured to 16 significant digits accuracy, the accuracy of numbers in Matlab. Also when typing in numbers, you yourself may have entered them to less than 16 significant digits accurate.

So, to do a good job, you should have some sort of ballpark in your mind for how big the relative errors are in the equations that you give to Matlab. Then, if these errors, when multiplied by the condition number, become a nontrivial percentage, you should treat the obtained solution with caution. Maybe try experimenting with different possible values of the constants that you give Matlab, and see how the solution changes. (Note that in this case, it is not at all uncommon that the actual error is much smaller than the worst-case scenario estimated using the condition number.)

To explore these issues, let's assume that the numbers in the equations, the data given to Matlab, are not exact, but have errors of say 1%:

```
% the ballpark relative error in the data
dataRelerr=0.01
```

```
dataRelerr =   0.010000
```

Then the computed solution vector may have a relative error up to `condA` as large:

```
% the corresponding maximum error in the soluton:
xRelerr=dataRelerr*condA
```

```
xRelerr =   0.37939
```

While a 1% error might be acceptable for engineering applications, a 38% error is probably not! Without checking the condition number, we would have had no clue of this potential problem. Some experimenting with the numbers you give Matlab is clearly in order.

One additional warning:

WARNING: *Do not check a solution by substituting it into the equations and seeing whether they are accurately satisfied.*

Substituting the obtained solution back into the equations can show you whether you solved the system correctly, but *not* whether your solution is accurate. Typically, even very inaccurate solutions obtained from left division will satisfy the equations quite accurately. You must use the condition number to say anything meaningful about the accuracy of the solution.

An unsolvable example

Consider now the modified system of equations

$$\begin{aligned} 1x_1 + 2x_2 + 3x_3 &= 3 \\ 4x_1 + 5x_2 + 6x_3 &= 2 \\ 7x_1 + 8x_2 + 9x_3 &= 9 \end{aligned}$$

The only change is the leading 4 instead of 0 in the second equation. But now there is no solution to the system. It is said that the matrix is "singular". (If the matrix is singular, there is either no solution at all, like here, or if there is one, there are infinitely many different solutions. In the latter case, you still would not know which of these infinitely many solutions is the one you want.)

We want to check that if we try to solve this singular system in Matlab *using proper procedures,* we will correctly conclude that it cannot be done. In other words, we want to check that we will not end up taking a nonsensical solution seriously.

To form the new matrix, which we will call `ABad` in Matlab, we want to take the old matrix `A` and just change the zero in row 2, column 1 into a 4. We can do that with "indices". Always remember:

REMEMBER: *For matrices, the proper order is row-column.*

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

Create `ABad`:

```
% copy A into ABad
ABad=A;

% change the 0 element in row 2 and column 1 into a 4.
ABad(2,1)=4
```

```
ABad =
   1   2   3
   4   5   6
   7   8   9
```

Now try to solve using proper procedures. First check the condition number:

```
% evaluate the condition number
condABad=cond(ABad)
```

```
condABad =     6.0262e+16
```

The above condition number is excessive. Matlab cannot solve this system even if it is given the exact numbers in the equations. The 10^{-16} relative error in storing numbers in Matlab prevents that:

```
% the relative error in the solution due to inaccuracy
xRelerrDueToMatlab=condABad*eps
```

```
xRelerrDueToMatlab =   13.381
```

The 10^{-16} relative error in Matlab would mean a relative error in the solution of about 1,300%!

Actually, large condition numbers are likely to be inaccurate, like here. The true condition number in this case is infinite. There simply *is* no solution. The correct relative error in any solution is infinite.

In any case, checking the condition number did show us that we cannot meaningfully solve this system in Matlab. So we should stop here and not try to solve it at all.

But suppose that we would *not* have checked the condition number. What would have happened? We would have solved and found some supposed "solution" (where there is none):

```
% result of stupidly "solving" the equations:
xBad = ABad \ b
```

```
warning: matrix singular to machine precision, rcond =
   2.20304e-18
xBad =
   0.50000
   0.33333
   0.16667
```

Nice numbers, but they are all wrong. *Do not solve a singular system unless told so!*

Actually, I am somewhat surprised by the above numbers produced by Octave. I would have expected a solution like that produced by Matlab instead:

```
xBadMatlab = [-3.6E16  7.2E16  -3.6E16]'
```

```
xBadMatlab =
  -3.6000e+16
   7.2000e+16
  -3.6000e+16
```

These numbers are, of course, all wrong too. But they are very large suggesting an infinite solution, rather than a few nice numbers.

MATRIX MANIPULATIONS

For advanced applications in linear algebra you must know how to do certain tasks. Some of this you should already have seen in MAC 1140.

Transposes

The "transpose" of a matrix A is in mathematics 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 earlier,

REMEMBER: *To transpose in Matlab, append a quote.*

(Actually, if the matrix is complex, the quote also swaps the sign of $\sqrt{-1}$, but we will only look at real matrices here.)

```
% our good old vector b
b
% its trandpose
bT=b'
```

```
b =
    3
    2
    9
bT =
    3    2    9
```

The column vector became a row vector by the transpose.
Note that a second transpose always undoes the first; `b"` is the same as `b`:

```
% transpose of bT is b
bTT=bT'
```

```
bTT =
    3
    2
    9
```

Transposing a matrix like `A` goes the same way:

```
% our good old matrix A
A
% its transpose
AT=A'
% the transpose of its transpose
ATT=AT'
```

```
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

Linear algebra has its own rules of multiplying matrices together. These rules are *different* from the elementwise multiplications of arrays in Matlab that we have seen so far. One key thing to remember:

REMEMBER: *Matrix multiplication is always row-column.*

More precisely, matrix multiplication takes *dot* products of the *rows* of the first matrix with the *columns* of the second matrix. A dot product consists of elementwise multiplication, *followed by a summation of all the obtained products.* That summation results in a single final number, a "scalar". That is the reason that the dot product is also often called the "scalar" product. It produces a single number.

Note further that this applies to vectors as well as to matrices. Linear algebra considers vectors to be just special cases of matrices. In particular a column vector is just a matrix with a single column and a row vector is just a matrix with a single row.

As an example, consider what we get if we multiply, in the matrix way, matrix A of the earlier system of equations to the solution vector $\vec{x}$:

$$A\vec{x} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 1 \\ -2 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \times 1 + 2 \times (-2) + 3 \times 2 \\ 0 \times 1 + 5 \times (-2) + 6 \times 2 \\ 7 \times 1 + 8 \times (-2) + 9 \times 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 9 \end{pmatrix}$$

Note that this involved three dot products, each between a row of A and the column vector $\vec{x}$, and that each dot product produced a single number (3, 2, and 9 respectively).

To achieve matrix manipulation in Matlab, do *not* put a point before the $*$:

```
% multiply A and x the matrix way
A_Star_x=A*x
```

```
A_Star_x =
    3
    2
    9
```

If by mistake you do put a point before the $*$, you do not, unfortunately, get a message that you are doing it wrong. Instead of warning you, Matlab will instead give you some weird set of numbers that you do not want. Indeed, the Matlab interpretation of `A.*x` is a prime example of what *not* to do in order to promote readable, easily understandable code.

Note also that the result of the multiplication $A\vec{x}$ is the right hand side vector $\vec{b}$. Indeed, in terms of matrix multiplication, a linear system of equations always

takes the simple form $A\vec{x} = \vec{b}$. That is why, in some sense, we want to divide $\vec{b}$ by matrix A to get $\vec{x}$.

Let's test how accurately `A*x` really equals `b` by taking differences:

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

```
show_A_x_b =
    1    2    3    1    3
    0    5    6   -2    2
    7    8    9    2    9
errors =
    0
    0
    0
Maximum error: 0.0E+00
```

Doing the same for the singular matrix shows that the Octave solution is all crap:

```
% recall ABad, xBad, and b
show_ABad_xBad_b=[ABad xBad b]
% evaluate the difference between ABad xBad and b
errors=ABad*xBad-b
% evaluate the maximum error
fprintf('Maximum error: %.1E',max(abs(ABad*xBad-b)))
```

```
show_ABad_xBad_b =
    1.00000    2.00000    3.00000    0.50000    3.00000
    4.00000    5.00000    6.00000    0.33333    2.00000
    7.00000    8.00000    9.00000    0.16667    9.00000
errors =
   -1.3333
    2.6667
   -1.3333
Maximum error: 2.7E+00
```

For the Matlab solution, things are somewhat different:

```
% recall ABad, xBadMatlab, and b
show_ABad_xBadMatlab_b=[ABad xBadMatlab b]
% evaluate the difference between ABad xBad and b
```

```
errors=ABad*xBadMatlab-b
% evaluate the maximum error
fprintf('Maximum error: %.1E',...
        max(abs(ABad*xBadMatlab-b)))
```

```
show_ABad_xBadMatlab_b =
 Columns 1 through 4:
   1.0000e+00   2.0000e+00   3.0000e+00  -3.6000e+16
   4.0000e+00   5.0000e+00   6.0000e+00   7.2000e+16
   7.0000e+00   8.0000e+00   9.0000e+00  -3.6000e+16
 Column 5:
   3.0000e+00
   2.0000e+00
   9.0000e+00
errors =
  -3
  -2
  -9
Maximum error: 9.0E+00
```

Note that relative to the great size of `xBadMatlab`, the errors are actually very small. The solution is still no good, but it does satisfy the equations relatively accurately. The Octave one does not.

Next, since matrix multiplication dots *rows* of the *first* matrix with *columns* of the *second*, the matrices are not interchangable, Indeed, normally AB is not at all the same as BA. In other words,

REMEMBER: *Matrix multiplication does not commute.*

For example, $\vec{x}A$ is not at all the same as $A\vec{x}$. In fact $\vec{x}A$ does not even exist: each *row* in column vector $\vec{x}$ has only a single component, but each *column* in A has three components. So there is no way to take dot products between them. This illustrates another important point:

REMEMBER: *Rows and columns involved in matrix multiplication must have the same number of components.*

Matrices A and ABad can be multiplied together in either order, but the result will not be the same:

```
% recall A and ABad
show_A_ABad=[A ABad]
% matrix product A ABad
A_Star_ABad=A*ABad
% recall ABad and A
show_ABad_A=[ABad A]
```

```
% matrix product ABad A
ABad_Star_A=ABad*A
```

```
show_A_ABad =
    1    2    3    1    2    3
    0    5    6    4    5    6
    7    8    9    7    8    9
A_Star_ABad =
     30      36      42
     62      73      84
    102     126     150
show_ABad_A =
    1    2    3    1    2    3
    4    5    6    0    5    6
    7    8    9    7    8    9
ABad_Star_A =
     22      36      42
     46      81      96
     70     126     150
```

Check yourself that, for example, the (2,3) component of `A*ABad` is $0*3+5*6+6*9=84$, but the (2,3) component of `ABad*A` is $4*3+5*6+6*9=96$.

Note also that the *elementwise* multiplications `A.*ABad` and `ABad.*A` are possible and do produce the same result:

```
% recall A and ABad
show_A_ABad=[A ABad]
% elementwise product A ABad
A_PointStar_ABad=A.*ABad
% elementwise product ABad A
ABad_PointStar_A=ABad.*A
```

```
show_A_ABad =
    1    2    3    1    2    3
    0    5    6    4    5    6
    7    8    9    7    8    9
A_PointStar_ABad =
     1     4     9
     0    25    36
    49    64    81
ABad_PointStar_A =
     1     4     9
     0    25    36
    49    64    81
```

However, the result is completely different from the matrix product. For example, while the (2,3) component of matrix product `A*ABad` was $0*3+5*6+6*9 = 84$, the (2,3) component of elementwise product `A.*ABad` (as well as of `ABad.*A`) is $6 * 6 = 36$.

Dot products

Matrix multiplication always takes dot products between the rows of the first matrix and the columns of the second matrix. Therefore, if you want to obtain the dot product between two vectors using matrix multiplication, you must make sure that the first vector is a row vector and the second vector a column vector.

REMEMBER: *In matrix multiplication terms, any dot product between vectors must be row-column.*

For example, you cannot do a matrix multiplication between `x` and `b`. Matlab will refuse because the single element rows of `x` cannot be dotted with the three element column vector `b`. But since the transpose `x'` of `x` is a row vector, you can multiply `x'` by `b` to get the dot product of the two vectors:

```
% the dot product of x and b as a matrix product
x_Dot_b=x'*b
% the dot product of b and x as a matrix product
b_Dot_x=b'*x
```

```
x_Dot_b =  17
b_Dot_x =  17
```

Note that the dot product of `b` and `x` is the same as that of `x` and `b`. (However, this is no longer completely true if the vectors are complex.)

Another way to find the dot product is to use the Matlab `dot` function:

```
% the dot product found using the Matlab "dot" function
x_Dot_b=dot(x,b)
```

```
x_Dot_b =  17
```

From physics, you should still remember that if two vectors are orthogonal, then their dot product is zero. Since the dot product of `x` and `b` is not zero, they are not orthogonal to each other.

From physics, you should also still remember that the dot product of a vector with itself gives the square length of the vector. To get the length, take a square root. For example, the length of vector `x` is

```
% recall x
x
% find the length of x using matrix multiplication
xLength=sqrt(x'*x)
```

```
x =
     1
    -2
     2
xLength =  3
```

Indeed, $|\vec{x}| = \sqrt{1^2 + (-2)^2 + 2^2} = 3$ according to the Pythagorean theorem, which agrees with the value above.

You might think you should be able to get the length of a vector also with the `length` function. But these idiots at Matlab already use the name `length` for something else that is completely useless. As a result of this stupidity, to get the length of a vector, you will have to use the `norm` function:

```
% find the length of x using the "norm" function
xLength=norm(x)
```

```
xLength =  3
```

There are a couple of other useful functions for arrays. If you want the total number of components (elements) in a vector or array, use the `numel` function:

```
% number of elements in x and A
xNumel=numel(x)
ANumel=numel(A)
```

```
xNumel =  3
ANumel =  9
```

Note that it is just coincidence that both the length and number of components of the current vector `x` are 3. They are completely different things.

If you want the number of rows and columns, use the `size` function:

```
% rows and columns in x and A
x
xSize=size(x)
A
ASize=size(A)
```

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

So, what is this useless `length` function that Matlab defines? Matlab defines `length(A)` to be `max(size(A))`, a normally useless quantity. You should *never* use the Matlab `length` function; that is confusing programming because the name "length" suggests a length to normal (nonMatlab) people. If you can find a use for `max(size(A))`, (and I cannot think of one), write it as `max(size(A))`.

Special matrices

A *zero matrix* is the matrix equivalent of the number zero. Adding or subtracting a zero matrix A to a matrix does not change the matrix. Multiplying by a zero matrix produces zero. A zero matrix contains all zeros. The symbol for a zero matrix is typically Z.

To test this by example, let's first create a bigger matrix than `A` for testing. One way to do so is put both `A` and its transpose `A'` in a matrix:

```
% create a bigger matrix from A and AT
Big=[A A']
% its number of rows and columns
BigSize=size(Big)
```

```
Big =
    1    2    3    1    0    7
    0    5    6    2    5    8
    7    8    9    3    6    9
BigSize =
    3    6
```

Next let's show that if we add or subtract a zero matrix to `Big`, it does not change `Big`. Note that:

REMEMBER: *Added or substracted matrices must have the same size.*

In Matlab you create a zero matrix using the `zeros` function.

```
% created the needed 3 row, 6 column zero matrix
Z=zeros(3,6)
% a better way to do this (still works if we change Big)
Z=zeros(size(Big))

% recall Big
Big

% add or subtract the zero matrix
Big_Plus_Z=Big+Z
Big_Minus_Z=Big-Z
Z_Plus_Big=Z+Big
```

```
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     0     7
     0     5     6     2     5     8
     7     8     9     3     6     9
Big_Plus_Z =
     1     2     3     1     0     7
     0     5     6     2     5     8
     7     8     9     3     6     9
Big_Minus_Z =
     1     2     3     1     0     7
     0     5     6     2     5     8
     7     8     9     3     6     9
Z_Plus_Big =
     1     2     3     1     0     7
     0     5     6     2     5     8
     7     8     9     3     6     9
```

Next let's show that if you *postmultiply* `Big` by a zero matrix `Z`, (as in `Big*Z`), you get zero.

Do recall from before that to do the multiplication, the number of components of the *rows* of `Big` must equal the number of components of the *columns* of `Z`. The number of components of the rows of `Big` equals 6, its number of columns, or more generally, BigSize(2). Similarly, the number of components of the columns

of `Z` is its numbers of rows, the first component of `size(Z)`. The number of columns in `Z` can be anything. For example we can take the number of columns equal to 1, making `Z` a column vector:

```
% create a zero column vector of 6 rows
Z=zeros(6,1)
% the better way to do this
Z=zeros(BigSize(2),1)
% postmultiply Big by this zero matrix
Big_Star_Z=Big*Z
```

```
Z =
     0
     0
     0
     0
     0
     0
Z =
     0
     0
     0
     0
     0
     0
Big_Star_Z =
     0
     0
     0
```

Another posibility is to the take number of columns in `Z` equal to its number of rows, creating what is called a "square" zero matrix.

```
% create a square zero matrix of the right size
Z=zeros(6,6)
% the better way to do this
Z=zeros(BigSize(2),BigSize(2))
% postmultiply Big by this zero matrix
Big_Star_Z=Big*Z
```

```
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
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_Star_Z =
    0     0     0     0     0     0
    0     0     0     0     0     0
    0     0     0     0     0     0
```

In either case, the result of the multiplication is a zero matrix, but its size is different.

If we want to *premultiply* `Big` by `Z`, as in `Z*Big`, the number of columns of `Z` must equal the number of rows in `Big`, 3, or more generally BigSize(1). Let's first try premultiplying by a 3 column row vector:

```
% create a zero row vector of the right size
Z=zeros(1,3)
% the better way to do this
Z=zeros(1,BigSize(1))
% premultiply Big by this zero matrix
Z_Star_Big=Z*Big
```

```
Z =
    0     0     0
Z =
    0     0     0
Z_Star_Big =
    0     0     0     0     0     0
```

Next let's try premultiplying by a 3 by 3 square zero matrix:

```
% create a square zero matrix of the right size
Z=zeros(3,3)
% the better way to do this
Z=zeros(BigSize(1),BigSize(1))
% premultiply Big by this zero matrix
Z_Star_Big=Z*Big
```

```
Z =
    0     0     0
    0     0     0
    0     0     0
```

```
Z =
     0     0     0
     0     0     0
     0     0     0
Z_Star_Big =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
```

A *unit matrix* (or *identity matrix*) is the matrix equivalent of the number 1; multiplying by a unit matrix does not change anything.

In analogy to the zero matrix, you might think that a unit matrix contains all ones, which would be given by the Matlab `ones` function.

But actually, a unit matrix is square (the number of rows equals the number of columns) and only contains ones on the "main diagonal" that goes from top left corner to bottom right corner. The rest of the unit matrix is zero. The symbol for a unit matrix is typically "I". In Matlab you can create a unit matrix with the `eye` function (somebody's joke).

To be able to postmultiply `Big` by a unit matrix `I`, the number of rows and columns in `I` must equal the number of columns `BigSize(2)` of `Big`, so

```
% create a unit matrix of the right size
I=eye(6)
% the better way to do this
I=eye(BigSize(2))
% recall Big
Big
% postmultiply Big by this unit matrix
Big_Star_I=Big*I
```

```
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
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    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
```

To be able to premultiply `Big` by a unit matrix `I`, the number of rows and columns in `I` must equal the number of rows `BigSize(1)` of `Big`, so

```
% create a unit matrix of the right size
I=eye(3)
% the better way to do this
I=eye(BigSize(1))
% recall Big
Big
% premultiply Big by this unit matrix
I_Star_Big=I*Big
```

```
I =
Diagonal Matrix
    1    0    0
    0    1    0
    0    0    1
I =
Diagonal Matrix
    1    0    0
    0    1    0
    0    0    1
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
```

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.

```
% an example symmetric matrix
S = [3   4   5
```

```
     4   6   7
     5   7   8]
% show that the transpose is the same
ST=S'
```

```
S =
    3    4    5
    4    6    7
    5    7    8
ST =
    3    4    5
    4    6    7
    5    7    8
```

Parts of matrices

When we created singular matrix `ABad` from `A`, 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 `ABad` was `ABad(2,1)`.

You can also address multiple elements in a matrix by using `START:END` constructs. Below are some examples. For example, consider how to take (parts of) rows out of a matrix:

```
% recall the example big matrix
Big
% take out part of row 2 (note row-column!)
row2part=Big(2,2:4)
% take out the entire row 2 (a bare : means "all")
row2all=Big(2,:)
% a bad way to do this, as it is less readable
row2all=Big(2,1:end)
% a worse way to do this, as it is less understandable
row2all=Big(2,1:6)
```

```
Big =
    1    2    3    1    0    7
    0    5    6    2    5    8
    7    8    9    3    6    9
row2part =
    5    6    2
row2all =
    0    5    6    2    5    8
row2all =
    0    5    6    2    5    8
```

```
row2all =
    0    5    6    2    5    8
```

A far more common thing is to have to take an entire column out of a matrix. This goes similarly:

```
% recall the example big matrix
Big
% take out column 4 (note row-column!)
col4all=Big(:,4)
```

```
Big =
    1    2    3    1    0    7
    0    5    6    2    5    8
    7    8    9    3    6    9
col4all =
    1
    2
    3
```

You may also have to take a set of columns out:

```
% recall the example big matrix
Big
% take out columns 3, 4, and 5 (note row-column!)
col345all=Big(:,3:5)
% take out columns 3 and 5
col35all=Big(:,[3 5])
```

```
Big =
    1    2    3    1    0    7
    0    5    6    2    5    8
    7    8    9    3    6    9
col345all =
    3    1    0
    6    2    5
    9    3    6
col35all =
    3    0
    6    5
    9    6
```

Conceivably, you might have to delete some columns from a matrix. This is done by setting the part of the matrix to delete equal to the "nil" matrix `[]`. For example, to delete rows 3 and 5 from `Big`:

```
% copy Big to BigReduced
```

```
BigReduced=Big
% now delete its columns 3 and 5
BigReduced(:,[3  5])=[]
```

```
BigReduced =
   1    2    3    1    0    7
   0    5    6    2    5    8
   7    8    9    3    6    9
BigReduced =
   1    2    1    7
   0    5    2    8
   7    8    3    9
```

EIGENVALUES AND EIGENVECTORS

Eigenvalues and eigenvectors are crucial to many engineering application. Unfortunately, here we can only give a brief first look at them. First their definition: 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 (or `lambda` in Matlab) 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 using Matlab.

See what is available to do so:

```
% the next is commented out; it must be run interactively
%lookfor eigenvalue
```

A simple example

As an example we will use Matlab function `eig` to find the eigenvalues and eigenvectors of a matrix of interest in fluid flows like lubrication.

```
% The (given) example matrix
C=1
S = [0 C 0
     C 0 0
     0 0 0]
% get the three eigenvalues as a vector
lambdaVals=eig(S)
```

```
C =  1
S =
    0    1    0
    1    0    0
    0    0    0
lambdaVals =
   -1
    0
    1
```

To get readable code, you will probably want to take the individual eigenvalues out of the vector. That can be done as:

```
% get the individual three eigenvalues
lambda1=lambdaVals(1)
lambda2=lambdaVals(2)
lambda3=lambdaVals(3)
```

```
lambda1 = -1
lambda2 = 0
lambda3 =  1
```

If you also want the eigenvectors, the format is somewhat different:

```
% get both the three eigenvalues and their eigenvectors
[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
```

This gives the three eigenvalues on the main diagonal of a matrix `Lambda` and the three eigenvectors as the columns of a matrix `E`.

To get readable code, you will probably want to take the individual eigenvalues and eigenvectors out of these matrices. That can be done as:

```
% get the individual three eigenvalues
lambda1=Lambda(1,1)
lambda2=Lambda(2,2)
lambda3=Lambda(3,3)
% get the individual three eigenvectors
e1=E(:,1)
e2=E(:,2)
e3=E(:,3)
```

```
lambda1 = -1
lambda2 = 0
lambda3 =  1
e1 =
  -0.70711
   0.70711
   0.00000
e2 =
   0
   0
   1
e3 =
   0.70711
   0.70711
   0.00000
```

Let's check that Matlab found the right vectors. If they are OK, for each, the left hand side in $S\vec{E} = \lambda\vec{e}$ must equal the right hand side:

```
% check the first eigenvalue and its eigenvector
LHS1_RHS1_difference=[S*e1 lambda1*e1 S*e1-lambda1*e1]
% check the second eigenvalue and its eigenvector
LHS2_RHS2_difference=[S*e2 lambda2*e2 S*e2-lambda2*e2]
% check the third eigenvalue and its eigenvector
LHS3_RHS3_difference=[S*e3 lambda3*e3 S*e3-lambda3*e3]
```

```
LHS1_RHS1_difference =
   0.70711   0.70711   0.00000
  -0.70711  -0.70711   0.00000
   0.00000  -0.00000   0.00000
LHS2_RHS2_difference =
```

```
   0    0    0
   0    0    0
   0    0    0
LHS3_RHS3_difference =
   0.70711    0.70711    0.00000
   0.70711    0.70711    0.00000
   0.00000    0.00000    0.00000
```

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.
2. 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 `S` was symmetric, let's check whether Matlab found the right eigenvalues and eigenvectors.

The eigenvalues, -1, 0, and 1, are indeed real, OK.

As we saw before, to check that the length of each eigenvector is 1, we can either use the Matlab `norm` function or matrix multiplication. Here we will use matrix multiplication:

```
% the length of eigenvector e1 must be one
e1Length=sqrt(e1'*e1)
% the length of eigenvector e2 must be one
e2Length=sqrt(e2'*e2)
% the length of eigenvector e3 must be one
e3Length=sqrt(e3'*e3)
```

```
e1Length =  1
e2Length =  1
e3Length =  1
```

Note that we did not really need the square roots; if the square length is 1, then so is the length.

As we saw before, to check whether two vectors are orthogonal, we must check whether their dot product is zero. The dot product can be found with the Matlab `dot` function or by matrix multiplication. Here we will use matrix multiplication:

```
% e1 and e2 are orthogonal if their dot product is zero
e1_Dot_e2=e1'*e2
% e2 and e3 are orthogonal if their dot product is zero
e2_Dot_e3=e2'*e3
% e3 and e1 are orthogonal if their dot product is zero
e3_Dot_e1=e3'*e1
```

```
e1_Dot_e2 = 0
e2_Dot_e3 = 0
e3_Dot_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, L_1) 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 precise 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 near the main diagonal, 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 band matrices.