Matlab Handout 2 – Matrices

Like vectors, matrices in Matlab have both mathematical (linear algebraic) and computational natures. Matlab can perform a large number of Matrix operations/calculations. But individual matrix entries can be treated like individual variables – in this way the matrix is just a two-dimensional array of variables.

1) **creating, assigning and accessing values –**

Just like with vectors, brackets "[]" are used to create matrices (see "help paren", "help punct"). Also like in vectors, the standard input is by rows. Within the brackets, the rows of the matrix are separated by a semi-colon ";". For example:

2x2 matrix	a = [1 2; 3 4]
2x3 matrix	a = [1 2 3; 4 5 6]
3x2 matrix	a = [1 2; 3 4; 5 6]

The semi-colon denotes the end of each row. Each row has to be the same length, otherwise Matlab will give you an error (i.e. a = [1 2 3; 4 5; 6 7 8] will not work).

Another way to create matrices is to enter them by their individual entries. In vectors, we saw that "a(3)" accessed the third entry of the vector. It could also be used to assign values to the third entry as in "a(3) = 4". The same can be done with matrices, except now we need to specify the row and the column. For example, to access the the entry in the third row, and the fourth column of a matrix, you would use "a(3,4)". To build a matrix this way you could use:

b(1,1) = 1	puts a value of 1 into the **row** 1 **column** 1 entry
b(1,2) = 2	puts a value of 2 into the **row** 1 **column** 2 entry
b(2,1) = 3	puts a value of 3 into the **row** 2 **column** 1 entry
b(2,2) = 4	puts a value of 4 into the **row** 2 **column** 2 entry

This is a slightly tedious way of creating a matrix, but it illustrates the important point of accessing individual entries, and the array nature of Matlab's matrices. This can be useful if you need to calculate individual matrix elements, and then put them into a matrix. A good way to automate this process is to use a script file, and **for** loops. For example, you could create a script file with the following:

```
clear
for n = 1:10
        for m = 1:10
                a(n,m) = m*sqrt(n);
        end
end
a
```

In the above, we accessed an individual entry. We can generalize this, and instead of accessing or working on an individual entry, work on whole columns or rows of the matrix. We do this using the colon ":" operator (see "help colon"). For example:

```
clear
a(1,:) = [1  2  3  4]
a(2, :) = [5  6  7  8]
```

is equivalent to

```
a = [1  2  3  4;  5  6  7  8]
```

Let's dissect the above command. We know that in the statement "a(n, m)" that the first number (n) indicates the row, and the second number (m) indicates the column. In the statement "a(1, **:**)", we've indicated we want to work on the first row. Normally, we would have put a number after the comma "**,**" but this time we've used a colon "**:**". The colon tells Matlab that instead of working on only one column, we want to simultaneously access all of the available columns of the matrix "a". If we haven't yet created matrix "a" then our first statement defines the numbers of columns that "a" will have. If we have already created a matrix "a", then the number of columns in our first statement **has** to match the number of columns which the existing matrix "a" already has.

We can define a matrix by columns instead of rows by switching the position of the colon:

```
b(:,1) = [1  2  3  4]'
b(:,2) = [5  6  7  8]'
```

note the apostrophe after the bracket – without this apostrophe, we would be telling Matlab to assign a row of numbers to a column, which Matlab can't make any sense of.

We can also use the above statements to copy, access, select, etc. segments of matrices. Our "b" matrix above has 2 columns and 4 rows. If we want to get the first row, we could use:

```
c = b(1, :)
```

This tells Matlab to assign a vector to "c" that is made up of the first row of the matrix "b" (in this case [1 5]).

We can also get just a section of a column of a matrix. We would do this by:

```
c = b(2:4, 1)
```

Instead of using either the colon, or an individual number to indicate the row(s) we're interested in, we've used the statement "2:4". This tells matlab we want the rows two through 4.

You might recognize the statement 2:4 as also one which tells matlab to generate a vector [2 3 4]. It is actually doing the exact same thing in this case!! In generalizing from accessing individual entries, to rows and columns, we make the same sort of change in the type of entry we are accessing in a matrix. So instead of using just a number to indicate which row, we can give Matlab a list of rows, in the form of a vector. For example, look at the following set of commands:

```
clear
a = [1  2  3  4]
a(2,:) = [5  6  7  8]
a(3,:) =  [9  10  11  12]
a(4,:) = [13  14  15  16]
```

the matrix "a" then looks like:

```
    1     2     3     4
    5     6     7     8
    9    10    11    12
   13    14    15    16
```

we could access the bottom right quadrant of "a" using the command:

```
a(3:4,3:4)
```

or the top right quadrant by:

```
a(1:2,3:4)
```

or the top left quadrant by:

```
a(1:2, 1:2)
```

We could access the last column by

```
a(:,4)
```

or the last three entries of the last column by

```
a(2:4,4)
```

or the first and last entries of the last column by

```
a([1 4], 4)
```

We could get the last column, but reverse the order by

```
a(4:-1:1,4)
```

or

```
a([4  3  2  1], 4)
```

we could get the 2nd and 4th columns, by
a(:, [2 4])

or the 2nd and 4th columns in reverse order by
a(4:-1:1, [2 4])

Basically you can use any *integer* valued vector in the parenthesis above to access values in a matrix. Besides being integers, the values must be positive, and they must not exceed the size of the matrix.

2) **Matrix Operators**
Basically, these are similar to the cases for vectors, except now we have two dimensions. Again, there are two major classifications: vector operators and array operators.

Vector operators

+ and - addition and subtraction. The two matrices must have the same shape, each individual entry is added together, and the resulting entry is placed in a new matrix. For example:

```
clear
a(1,:) = [1  2  3]
a(2,:) = [4  5  6]
a(3,:) = [7  8  9]
b(1,:) = [11 12  13]
b(2,:) = [14  15  16]
b(3,:) = [17  18  19]

c = a + b
would yield
[12  14  16]
[18  20  22]
[24  26  28]
```

* matrix multiplication. This is based on the linear algebraic definition:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1*5+2*7 & 1*6+2*8 \\ \\ 3*5+4*7 & 3*6+4*8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Another way to remember this is that it is the vector dot product between the rows of the first matrix, and the columns of the second matrix:

$$= \begin{bmatrix} (1 \quad 2)\bullet\begin{pmatrix}5\\7\end{pmatrix} & (1 \quad 2)\bullet\begin{pmatrix}6\\8\end{pmatrix} \\ (3 \quad 4)\bullet\begin{pmatrix}5\\7\end{pmatrix} & (3 \quad 4)\bullet\begin{pmatrix}6\\8\end{pmatrix} \end{bmatrix}$$

Similar to vector multiplication (dot product, inner product) the matrices must have **complementary** shapes. This can be seen by looking at the dot products in the above definition. The matrices must have dimensions such that the dot products in the above definition are feasible. For example:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} * \begin{bmatrix}7 & 8\\9 & 10\\11 & 12\end{bmatrix} = \begin{bmatrix} (1 \quad 2 \quad 3)\bullet\begin{pmatrix}7\\9\\11\end{pmatrix} & (1 \quad 2 \quad 3)\bullet\begin{pmatrix}8\\10\\12\end{pmatrix} \\ (4 \quad 5 \quad 6)\bullet\begin{pmatrix}7\\9\\11\end{pmatrix} & (4 \quad 5 \quad 6)\bullet\begin{pmatrix}8\\10\\12\end{pmatrix} \end{bmatrix}$$

will work, because the vectors in the dot products have the correct complementary shapes. Another way to check this is to see if the inner dimensions of the matrices agree. In the above case, we have a matrix with dimensions (2,3) times a matrix with dimensions (3,2). The inner dimensions are the rows of the first matrix and the columns of the second matrix, and in this case they are both 3. The size of the resulting product matrix is determined by the outer dimensions (in this case, a 2 x 2 matrix). Switching the order of multiplication will also still work (since then the inner dimensions will be 2), but will produce a 3 x 3 matrix.

Try creating the above matrices (the 2x2 matrices first, then the 2x3 and 3x2 matrices), and try multiplying them together, and also mixing them up (ie 2x2 * 2x3, 2x2 * 3x2)

The first important special case of matrix multiplication is when a matrix is multiplied by a vector. Following the rules above, we know that the inner dimensions must agree – so the following two multiplications would be valid:

$$\underset{(2x3)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} * \underset{(3x1)}{\begin{bmatrix}7\\8\\9\end{bmatrix}} \qquad\qquad \underset{(1x2)}{\begin{bmatrix}7 & 8\end{bmatrix}} * \underset{(2x3)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}}$$

This operation is also called a transformation – the matrix is said to transform the vector. First, try to predict the shape of resulting vectors from these operations. Then carry them out in Matlab.

A second important case of matrix multiplication occurs when you take the **outer** product of two vectors. In contrast to the previous case of vector multiplication, in which we multiplied a row vector times a column vector, in the outer product we multiply a column vector by a row vector. This follows the exact rules of matrix multiplication as above – a column vector is treated like a matrix which has only 1 column, and a row vector is treated like a matrix that has only 1 row. To illustrate:

```
clear
a = [1  2]
b = [3  4]'
```

The inner product:

```
a*b
```

would yield “11”.

The **outer** product:

```
b*a
```

would be calculated by:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} \bullet \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 3*1 & 3*2 \\ & \\ 4*1 & 4*2 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix}$$

and yields a 2 x 2 matrix. We could have predicted this by looking at our “matrix” dimensions: (2,1) x (1,2). First, the inner dimensions agree (they are both 1). Second, the outer dimensions are both 2, indicating a 2x2 matrix will be produced[1].

This will become important when you work on density operators in quantum.

Array Operators

.* ./ .^ array multiply, divide, and “raise to the power of”. The matrices must have the same shape. These work in exactly the same way as for vectors – the individual elements are multiplied together to yield the new elements of the new matrix. For example:

[1] If instead the dimensions had been (2,1) x (1,**3**) we would have produced a 2 x **3** matrix.

```
clear
a(1,:) = [1  2]
a(2,:) = [3  4]

b(1,:) = [5  6]
b(2,:) = [7  8]

a.*b
would yield

5        12
21       32

a./b
would yield

0.2              0.3333
0.4286           0.25

and   a.^b
would yield

1                64
2187             65536
```

3) **Matrix Functions**

a) **Vector functions extend to matrices**

The same functions we called "vector functions" all work on matrices. When operating on a matrix they generally treat each column of the matrix as a vector, and perform their operation on those columns individually. The functions listed before for vectors were:

size	This gives direct information on the number of rows & columns in a matrix. It returns a row vector with 2 entries, the first of which indicates the number of rows, the second indicating the number of columns
length	for matrices, this returns the largest dimension, not that useful.
max	returns a row vector whose entries indicate what the maximum value in each of the columns of the matrix are
min	returns a row vector whose entries indicate what the maximum value in each of the columns of the matrix are
sort	sort in ascending order. Returns a matrix of the same shape with each column of the matrix sorted in ascending order.

sum	returns a row vector which indicates the result of summing each column of the matrix

Use the following:

```
clear
a = [1  2  3;  4  5  6;  7  8  9];
```

and carry out the above functions on a:

```
size(a)
length(a)
max(a)
etc…
```

b) **Matrix creation functions**
There are some functions which are mainly used in the creation of matrices (see "help elmat"). The major ones are

zeros — creates a matrix which is filled with the number zero. For example, the statements:

```
zeros(4,4)
or
zeros([4 4])
```

create the matrix:

```
0   0   0   0
0   0   0   0
0   0   0   0
0   0   0   0
```

ones — works exactly the same as zeros, excepts creates a matrix filled the number 1. Try

```
ones(4,4)
or
ones([4 4])
```

eye — works like the above, except it creates the **I**dentity matrix:

```
eye([4 4])
```

creates:

```
  1   0   0   0
  0   1   0   0
  0   0   1   0
```

```
0   0   0   1
```

diag — works on diagonals in matrices. Can be used to create matrices with specific entries along the diagonal, or can be used to extract specific diagonals from a matrix. To start with, let's look at extracting diagonals from an exisiting matrix "a" from above:

diag(a)
returns the diagonal of a, [1 5 9] (but in column form)

diag(a,-1)
returns the diagonal 1 unit below the main diagonal. In this case, that would be [4 8]

diag(a,1)
returns [2 6]

diag(a,2)
returns [3]

diag(a,3)
returns an empty matrix, because there is no diagonal 3 entries away from the main diagonal.

Now let's look at creating a new matrix, by putting entries on or near the diagonal:

diag([1 2 3])
creates the following matrix:

```
1   0   0
0   2   0
0   0   3
```

Our input vector ("[1 2 3]") is used as the argument to the diag function, which puts it along the diagonal of a matrix.

We don't have to put our vectors on the main diagonal; we can also offset them slightly:

diag([1 2 3], 1)
creates:

```
0   1   0   0
0   0   2   0
0   0   0   3
0   0   0   0
```

diag([1 2 3], -1)
creates:

```
0   0   0   0
1   0   0   0
0   2   0   0
0   0   3   0
```

diag([1 2 3], 2)
creates:

```
0   0   1   0   0
0   0   0   2   0
0   0   0   0   3
0   0   0   0   0
```

0 0 0 0 0

1 above the main diagonal

1 below the main diagonal

2 above the main diagonal

Diversion

Let's go back to our wavefunction example from handout 1. Let's calculate the first ten wavefunctions, but this time let's store them in a matrix/array. We could use the following commands in a script file:

```
clear
dx = 0.1;            %this sets the increment for the x-axis
L = 10;              %this sets the box length for the particle in a
                     %box

x = 0:dx:L;          %this creates the x-axis, incremented by dx,
                     %ending at L

num_pts = length(x);       %this figures out how many points are along
                           %the x-axis

num_basis_funs = 10;       %this creates a variable, and gives it the
                           %value 10.  This is intended to be
                           %represent the number of basis functions
                           %we are using

basis_funs = zeros(num_basis_funs,num_pts);  %this creates a
                                             %num_basis_funs by
                                             %num_pts size matrix (10 x
                                             %101).
                                             %We have 10 row vectors,
                                             %each of which
                                             %will contain the data for
                                             %a basis function.

for n = 1:num_basis_funs
   basis_funs(n,:) = sqrt(2/L)*sin(n*pi*x/L);   %this calculates each
                                                %basis function
end                                             %and stores it in an
                                                %array

for n = 1:num_basis_funs
   n                                %this displays the value of the
                                    %variable n

   dx*basis_funs*basis_funs(n,:)'   %this calculates the overlap
                                    %integral between
                                    %the "n" basis function and all of
                                    %the other
                                    %basis functions.
   figure(n)
   clf
   plot(x,basis_funs(n,:))     %these lines bring up/create a window,
                               %clear it
                               %and plot a basis function in it

   pause
end
```

Try saving this script as "bfun.m" and running it. Some key points:
i) use variables as constants. Try not to use "hard" numbers in the calculations. Any variable you might want to adjust later on should have a variable, that is assigned a value at the beginning, so that it is easy to change it and then re-run the script (ie – we can easily change the size of the box L, the x-axis increment dx, or even the number of basis functions)

ii) Comments – the percent symbol "%" tells Matlab to ignore all the text that follows it on that line. This is incredibly useful for explaining what you are trying to do on a given line. This has a large number of benefits -
1) If you type out what you are trying to do, it will help clarify it in your own mind.
2) If you give the code to someone else to look at, it will help them understand it
3) If you ever have to come back anytime in the future and use the code again, it will help you understand what you were trying to do!

Don't underestimate the value of number of 3. In an extreme case, I once wrote some code, and when I tried to come back to it a mere week later, I had no idea what I was trying to do!! (that's me really being weak-minded, but it illustrates the point).

iii) the line "`dx*basis_funs*basis_funs(n,:)'`"
now we see another example of the matrix speed of matlab. With this innocuous looking line, we instantly calculate 10 integrals!!! To understand how, remember that the middle part, "basis_funs" is a matrix with dimensions (num_basis_funs, num_pts). The last part of the line "basis_funs(n,:)'" is a vector of length "num_pts". The entries in the vector that are generated as a result of the matrix-vector multiplication are the result of the **dot** product between the each row of the matrix "basis_funs" and the vector "basis_funs(n,:)'". We know from the last handout that this dot product is the Riemann sum of the overlap integral. The "dx" factor in front is also from the Riemann sum, as described before. The net result: a one-line operation calculates whether or not our basis functions are ortho-normal!

To further help understand this operation, here's a schematic of the matrices involved:

$$dx * \begin{bmatrix} \begin{array}{c}\textbf{first}\text{ value of the}\\ \textbf{first}\text{ basis function}\end{array} & \cdots & \begin{array}{c}\textbf{last}\text{ value of the}\\ \textbf{first}\text{ basis function}\end{array} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ \begin{array}{c}\textbf{first}\text{ value of the}\\ \textbf{tenth}\text{ basis function}\end{array} & \cdots & \begin{array}{c}\textbf{last}\text{ value of the}\\ \textbf{tenth}\text{ basis function}\end{array} \end{bmatrix} * \begin{bmatrix} \begin{array}{c}\textbf{first}\text{ value}\\ \textbf{nth}\text{ basis}\\ \text{function}\end{array} \\ \vdots \\ \vdots \\ \begin{array}{c}\textbf{last}\text{ value}\\ \textbf{nth}\text{ basis}\\ \text{function}\end{array} \end{bmatrix}$$

this is the matrix “basis_funs” (10x101)   this is “basis_funs(n,:)
(101x1)

The first dot product will be between the first row of the matrix “basis_funs”, and the column vector “basis_funs(n,:)”. When this is multiplied by “dx”, we see that it is the overlap integral between the two functions. The second dot product will be between the second row of the matrix, and the column vector. This is repeated up to the tenth row. The resulting answer we see will be a column vector with 10 entries, each one corresponding to the overlap integral between each of the 10 basis functions & the supplied basis function in the column vector.

Now let’s make a final script, which takes advantage of the fact that we’ve already calculated our basis functions using “bfun.m” above. In this script we’ll create a superposition state wavefunction which is normalized and composed of the basis functions. **Warning:** this script will not work unless the above script is run first.

```
k = 1:num_basis_funs;
del_psi = 1;
k_not = 3;
psi_k = exp(-((k-k_not)/del_psi).^2);      %this creates a vector psi,
                                        %which
                                        %we will use for our weighting
                                        %coefficients of our basis
                                        %functions


norm_const = sqrt(psi_k*psi_k');
psi_k = psi_k/norm_const;            %this calculates the normalization
                                  %constant of our psi function,
                                  %and then applies it so that psi
                                  %is normalized.

figure(1)
clf
plot(psi_k)


psi_x = psi_k*basis_funs;       %this line multiplies the elements of
                               %psi_k
                               %by their corresponding basis function
                               %and then, for each index, sums them
```

```
                                        %together

figure(2)
clf
plot(x,psi_x)

dx*basis_funs*psi_x'        %this calculates the overlap integral
                            %between
                            %our wavepacket and each of the component
                            %basis functions.
```

The new idea here mainly lies in the statement “psi_x = psi_k*basis_funs”. To illustrate this matrix multiplication:

$$\begin{bmatrix}\textbf{first}\text{ value} & \cdots & \textbf{last}\text{ value}\\ \text{psi\_k} & & \text{psi\_k}\end{bmatrix} * \begin{bmatrix}\begin{matrix}\textbf{first}\text{ value of the}\\ \textbf{first}\text{ basis function}\end{matrix} & \cdots & \begin{matrix}\textbf{last}\text{ value of the}\\ \textbf{first}\text{ basis function}\end{matrix}\\ \vdots & & \vdots\\ \vdots & & \vdots\\ \begin{matrix}\textbf{first}\text{ value of the}\\ \textbf{tenth}\text{ basis function}\end{matrix} & \cdots & \begin{matrix}\textbf{last}\text{ value of the}\\ \textbf{tenth}\text{ basis function}\end{matrix}\end{bmatrix}$$

psi_k, (1x10)   basis_funs(10x101)

In this case we see that the first dot product will be between the first column of the matrix “basis_funs”, and the row vector “psi_k”. What exactly is the first column of the matrix? It is the value of each of the basis functions at the first point **on the x axis** we chose (in this case x = 0). This dot product then corresponds to the “psi_k” weighted sum of the values of each basis function, evaluated at x = 0. This then is repeated for the rest of the points/values of x for which we have calculated our basis functions. Based on the outer dimensions of the above multiplication, we see that we expect the result to be a (1x101) matrix. Each of the entries then corresponds to the **wavepacket** at each of the 101 points.

c) **Matrix calculations**   see “help matfun”
The operations generally only work on square matrices. There are three main ones:

det   calculates the determinant of a square matrix. Using matrix “a” from above,

det(a)

would be calculated by:

$$\begin{vmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{vmatrix} = 1*\begin{vmatrix}5 & 6\\8 & 9\end{vmatrix} - 2*\begin{vmatrix}4 & 6\\7 & 9\end{vmatrix} + 3*\begin{vmatrix}4 & 5\\7 & 8\end{vmatrix}$$
$$= [5*9-6*8]-2*[4*9-6*7]+3*[4*8-5*7]$$
$$= [-3]-2*[-6]+3*[-3]$$
$$= 0$$

inv calculates the inverse of the matrix. An inverse matrix is defined such that the expression
a*inv(a)
or
inv(a)*a

yields the identity matrix
try the following commands:
inv(a)
a*inv(a)

eig Arguably the single most important Matlab function, as far as 5.61 is concerned. This calculates the eigenvalues & eigenvectors of a matrix. That is to say, it solves the following equation:
$\mathbf{A} \cdot \mathbf{b} = \lambda \mathbf{b}$
where **A** is a matrix, **b** is a column vector (known as the "eigenvector"), and λ is a constant (the "eigenvalues"). Solving this equation is said to "diagonalize **A**", which means that it allows you to transform **A** such that the resulting matrix only has entries along the main diagonal. This is done by finding a matrix **P** such that in the following equation

$\mathbf{A = P*D*P^{-1}}$

the matrix **D** will be diagonal. The eigenvalues λ (from the first equation) are along the diagonal of the matrix **D**, and the eigenvectors **b** from the first equation are the columns of the matrix **P**.

The **eig** function in matlab takes as an argument your matrix **A** which you wish to diagonalize. It will then return either just the eigenvalues λ, or it will return both the eigenvalues and the matrix **P**, which contains the eigenvectors.

For a concrete example:

clear

```
a = diag(ones(1,5),-1) + diag(ones(1,5),1)
eig(a)
```

produces a column vector which contains the eigenvalues of the matrix “a”. Make sure you understand how the matrix “a” was created in the above. Also, note the fact that the eigenvalues are **not** sorted.
Now try:

```
[P D] = eig(a)
```

in this case a matrix “P” and a matrix “D” are created. “P” contains the eigenvectors in its columns, “D” has the eigenvalues along its diagonal. To see this try:

```
diag(D)
```

this should give the same list of eigenvalues as above. Also, try the following:

```
a*P(:,1)
```

This statement is multiplying the first eigenvector times our original matrix “a”. If the everything worked out correctly, this should produce the vector “P(:,1)” (the first column of the matrix “P”) times a constant. In order to verify this, we should divide the above statement by that constant – the first eigenvalue (this is stored in the first entry in the matrix “D”), and compare the resulting vector to our original vector. Try:

```
P(:,1)
D(1,1)
a*P(:,1)/D(1,1)
```

are the two vectors the same?

One last bit of useful information is about sorting eigenvalues. In quantum, eigenvalues correspond to interesting physical quantities (momentum, energy, etc.) so it’s nice to have these sorted, rather than trying to make sense of a somewhat random list. To see a sorted list of our above eigenvalues, we would use

```
sort(diag(D))
```

This is easier to work with, but if we then keep using the sorted list we have a problem, because now we don't know which eigenvalues are associated with which eigenvectors. In order to rectify this, we need to use a more advanced feature of the sort function. If we use the statement

```
[y  indices] = sort(diag(D))
```

then we create two vectors. The first vector "y" contains the list of sorted values, the second vector "indices" contains the corresponding list of indices which the values in "y" had when they were in their original, pre-sorted vector. For example,

```
[y indices] = sort([20  15  5])
```

gives us a "y" vector that looks like

```
[5  15  20]
```

and an "indices" vector that looks like

```
[3 2 1]
```

The first entry of "y" is a 5. In our original vector, this was the third entry, so the first entry in "indices" is a 3. The second entry of "y" is a 15, this was the second entry of the original vector, so the second entry of "indices" is a 2. The third entry of "y" is a 20, which was in the first spot originally, so the third entry of "indices" is a 1.

Calculate **by hand** what you would expect "y" and "indices" would be for the vector

```
[16    18    15     4     8]
```

then plug it into matlab and compare results, to make sure you understand what sort is doing.

Back to our original problem. Let's take the raw data we have in the matrices "P" and "D", and created sorted versions. First,

```
[D_sorted   D_indices] = sort(diag(D))
D_sorted = diag(D_sorted)
```

The first line creates the sorted list of eigenvalues. The second line takes that sorted vector list, creates a matrix with it along the diagonal (the diag command), and then overwrites the original D_sorted vector with the new matrix.

Now we need to sort our eigenvectors. Let's start with the eigenvector corresponding to the lowest valued eigenvector. We'll put this in the first column of our "P_sorted" matrix:

```
P_sorted(:,1) = P(:,D_indices(1))
```

The first part "P_sorted(:,1)" tells Matlab that we're working on the first column of the "P_sorted" matrix. The second part tells us that we want the column of "P" given by the first entry of the vector "D_indices". From above, we know that the first entry of "D_indices" should give us the location of the eigenvector which had the lowest valued eigenvalue.
Now we can repeat, for the higher values

```
P_sorted(:,2) = P(:,D_indices(2))
P_sorted(:,3) = P(:,D_indices(3))
P_sorted(:,4) = P(:,D_indices(4))
P_sorted(:,5) = P(:,D_indices(5))
P_sorted(:,6) = P(:,D_indices(6))
```

Now we should have successfully created our sorted eigenvector matrix. To test this, try the following

```
P_sorted(1,:)
a*P_sorted(1,:)/D_sorted(1,1)
```

This is the same test we performed above – we're just comparing the eigenvector original with the eigenvector after the calculation, divided by the eigenvalue.

You might remember that you can change the way a matrix is arranged by using a vector to designate the order of the entries you want to look at. Perhaps the most powerful application of this is to the above, in which case we can simplify the creation of our "P_sorted" matrix. This is done by using the following statement:

```
P_sorted = P(:,D_indices)
```

In this statement, we've used the vector D_indices to indicate the order we want to display/assign the matrix P. This then is assigned to the matrix "P_sorted". **This is exactly identical** to the above operations we performed, but you can see it saves a lot of time.