

## Fourier MATLAB Assignment

### 1 Entering vectors

The simplest way to enter a vector is by listing its components as in the following example:

```
>> x=[3 16 -2.2 45 0]
```

(Here `>>` is the MATLAB prompt on the screen, you just type “`x=[3 16 -2.2 45 0]`”). This gives a vector called “`x`” with these components. Note that a space is needed between components. Type this and see how MATLAB prints out your vector.

Now type `>> x` again and see that the vector `x` reappears. It is stored for future use until you quit MATLAB or redefine it. The name given to a vector can contain up to 19 characters, but must begin with an alphabetical character.

You can get a vector with equally spaced increments via the format

```
>> b=2:3:98
```

This gives a vector starting with 2, with increments of 3 going up to a final value of 98. The format is always initial value:increment:final value. Note that since the components of `b` are all integers, the display only gave the integer value with no decimal places. Also note that the entire vector scrolls by on the screen. If you want to see some part of this vector, the command `>>x(k:l)` results in a display of the  $k^{\text{th}}$  through  $l^{\text{th}}$  components of `x`. Try

```
>> b(3:7)
```

and see what you get. If you just type `>>x(k)` the  $k^{\text{th}}$  component of `x` will be displayed.

You can concatenate vectors. For example, type

```
>> c=[x b x 5 2 -7]
```

and see the result. Note that decimals are displayed this time because the vector `x` has a non-integer component. Four decimal places are normally displayed, unless you change this option.

A useful vector is

```
>> n = 0:1:511
```

Type this and see the outcome. If you want to store a vector like `n` but you don't really need to see it displayed on the screen, type a semicolon at the end. That will suppress the printout. (This saves a lot of time on long computations.) Try this:

```
>> m=0:1:155;
```

Even though it is not displayed, the vector `m` is stored. To check this, just type

```
>> m
```

and `m` will be displayed.

There are a some convenient built-in functions that can be used to create vectors. For example,

```
>> w=ones(1,17)
```

will create a vector of length 17 whose entries are all 1. (The “1” is needed so that one gets a 1 by 17 matrix, that is, a vector. Most of the commands described here can be modified to create matrices, but we will stick with vectors for now because this is our main concern. The command `w=ones(17)` gives a 17 by 17 matrix whose entries are all 1.)

Similarly

```
>> w=zeros(1,17)
```

creates a vector of length 17 whose entries are all 0. Notice that we called this vector `w`, which we had used previously. This replaces the old definition of `w` by the new one.

The `linspace` command `linspace(x,y,N)` gives a vector of `N` components starting at `x` and changing linearly up to `y`. For example, try

```
>> w=linspace(0,20,41)
```

Notice that we needed a vector of length 41 to get increments of .5.

### 2 Operations on vectors

**MATHEMATICAL FUNCTIONS:** Given a vector, you can apply various mathematical functions to each component of the vector as in the following example:

```
>> y=cos(n)
```

Since `n` was the vector obtained above by `n=0:1:511`, `y` is a vector of length 512 whose components are the cosines of the corresponding components of `n`. If you are interested in seeing the plot of `y`, you can read Section 3 and then come back here. Actually the plot of `y` oscillates too fast for one to see very much, so you would get a more reasonable graph by letting `y=cos(n/64)` (see the text below for an explanation of the /64 part).

Any MATLAB manual should contain a list of MATLAB mathematical functions.

ADDING, SUBTRACTING, AND MULTIPLYING vectors componentwise: Given two vectors  $x$  and  $y$  of the same length, the command `>> z=x+y` produces the sum (componentwise) of  $x$  and  $y$ , while `>> z=x-y` produces the difference. These can be combined: try

```
>> p=n+n-n
```

Two vectors of the same length can be multiplied componentwise: `>> z=x.*y` forms the vector of the same length whose components are the product of the corresponding components of  $x$  and  $y$ . For example, with the vectors  $n$  as above, type

```
>> z= n.*n
```

You should get the vector whose components are the squares of the components of  $n$ . You can confirm this via `>>z(1:6)` for example. Similarly

```
>> t=n.*z
```

will give you the cubes of the components in  $n$ . (This can also be done directly as we will soon see.)

(The symbol `*` without the `.` is reserved for multiplication of matrices; if you type `>> z=n*y` you will get an error message telling you the matrices cannot be multiplied because the dimensions don't match correctly.)

You can add a scalar to every every component of a vector (or subtract a scalar from every component) as in this example

```
>>t=n-100
```

or

```
>> t=5+n
```

You can also multiply or divide a vector by a scalar componentwise, as follows. Try

```
>> r=3*n
```

and

```
>> s=r/3
```

EXPONENTIATING vectors componentwise: You can raise each element of a vector  $x$  to a power  $p$  with the command `>> z=x.^p` For example,

```
>> w=n.^2
```

gives the same result as `>> z=n.*n` above. The power  $p$  does not have to be an integer.

These operations can be combined. For example,

```
>> w=sin(pi*n.^2/10000)
```

gives a vector with values  $w(n) = \sin \frac{\pi n^2}{10,000}$ . Notice that  $\pi$  can be entered with the symbol `pi`. Similarly, the imaginary number  $\sqrt{-1}$  can be entered with the symbol `i`, unless you have defined `i` to be another variable (so it's better to not use `i` as a variable).

In general, MATLAB respects the usual order of operations. Parentheses can be added if needed. For example,

```
>> w=sin((pi*n).^2)
```

gives the vector  $w(n) = \sin(\pi n)^2$ . If you enter the vector

```
>> w=sin(pi*n.^2)
```

you may be surprised that the output is not 0 (it should be 0, since  $\sin \pi n^2 = 0$  for every  $n$ ). However, it is effectively 0. If you go back and look at the first few entries via the command `>>w(1:5)`, you will see a term  $1.0\text{e-}14$  in front of the vector. This means that the listed values are all multiplied by  $10^{-14}$ . The value is not exactly 0 due to round-off error in the MATLAB computations, which are done to 16 significant figures.

### 3 Plotting vectors

Vectors are plotted with the `plot` command. For example, define a vector  $m$  by

```
>> m = [6 2 -1 5]
```

and then try

```
>>plot(m)
```

A window labeled *Figure No. 1* should pop up on the screen with a plot of  $m$ . The window can be moved by putting the cursor on the top bar, holding the left side of the mouse down, and dragging the window to the location you want (this works with any window in the windows environment). The window can be resized by putting the mouse on the edge (whichever edge you are resizing) until you see the sign  $\leftrightarrow$  or  $\updownarrow$ , then holding the left side of the mouse down, and dragging the edge to the desired position.

WARNING: If  $x$  is a vector with complex-valued components, the command “`plot(m)`” will plot the imaginary part of each component above the real part. This is something we never really want to do, so make sure your vectors are real-valued before you plot them. The command “`real(z)`” will give you the vector whose components are the real

parts of the components of  $z$ , while “`imag(z)`” will do the same thing with the imaginary parts. You can also obtain the magnitude of each component with “`abs(z)`” and the phase angle with “`angle(z)`”.

Notice that in the plot of  $m$ , the value 6 is plotted at the point 1, then 2 at the point 2, -1 at 3 and 5 at 4. Without any further information, MATLAB assumes that a vector  $x$  of length  $N$  is thought of as  $[x(1), x(2), \dots, x(N)]$ . However, in this class we prefer to think of a vector of length  $N$  as  $[x(0), x(1), \dots, X(N-1)]$ . To get  $x$  plotted starting at 0, create the vector

```
>> r=[0 1 2 3]
```

and then use

```
>> plot(r,m)
```

In general the command “`plot(x,y)`”, for vectors  $x$  and  $y$  of the same length, plots the components of  $y$  above the corresponding components of  $x$ . This is why a vector like `n=0:1:511` is so useful for us; we will plot any vector  $z$  of length 512 by the command “`plot(n,z)`”. For example, with

```
>> z=n.^2
```

try

```
>> plot(n,z)
```

You should get a graph of the function  $f(t) = t^2$ . If you use “`plot(z,n)`” instead, you get a graph of the square root function.

Notice that this plotting property allows one to graph functions on whatever interval you like. For example, to graph  $f(t) = e^t$  on for  $0 \leq t \leq 1$ , one gets a very continuous-looking graph by plotting 1,000 evenly spaced points in  $[0,1]$ . So define

```
>> t=linspace(0,1,1000)
```

```
>> y=exp(t);
```

(recall that the semi-colon prevents the vector from scrolling on the screen)

```
>> plot(t,y)
```

You can plot more than one vector at a time. If you use “`plot(x1,y1,x2,y2)`” you will get the plots of  $y_1$  versus  $x_1$  and  $y_2$  versus  $x_2$ . (Similarly, for 3 or more plots.)

For example, let

```
>> l = 2*n;
```

```
>> k = 3*n;
```

and then try

```
>> plot(n,l,n,k)
```

For this  $x_1$  and  $x_2$  do not have to be the same size. Let

```
>> h=[n n+512];
```

```
>> p=[1 1];
```

and try

```
>> plot(n,k,h,p)
```

Changing the plot linestyle: Notice that all the plots so far have displayed a connect-the-dots graph, even for the vector `m=[6 2 -1 5]`. This is the default option, but there are others. They are obtained by adding an argument at the end of the plot command of the form ‘?’’, where here you do type the single quotation marks, and ? is any of the following symbols associated with the following types:

<u>line-types</u>	<u>symbol</u>
solid	-
dashed	- (typed as - - , it just looks like one keystroke)
dotted	:
dashdot	-.

<u>point-types</u>	<u>symbol</u>
point	.
plus	+
star	*
circle	o
x-mark	x

For example, try

```
>> plot(r,m,'x')
```

and

```
>> plot(r,m,'-.')
```

This is particularly useful when plotting more than one vector at a time. In this case, insert the '?' after each pair of vectors: try

```
>> plot(n,l,'-',n,k,':')
```

For this example, since '-' is the default option, you get the same result with

```
>> plot(n,l,n,k,':')
```

Setting your graph viewing window: Consider this example:

```
>> plot(n,n.^2)
```

(Before we did this by defining  $m=n.^2$ , but it can be done this way also, without explicitly defining  $m$ .) Notice that although  $n$  is a vector of length 512, the viewing window goes from 0 to 600 along the x-axis. For some reason MATLAB has a mind of its own when setting viewing windows, in both the x and y axes. However, you can set these windows to your own choice as follows. Suppose we want this plotted only for values along the x-axis going from 0 to 511, and along the y-axis we decide we want the limits to be -100,000 to 500,000. After obtaining the graph as we have already done, define a vector

```
>> v=[0 511 -100000 500000]
```

and then give the command

```
>> axis(v)
```

The result will be that the current graph is replotted in the window with bounds equal to the components of  $v$  (it doesn't have to be called  $v$ ).

CAPTIONS: You can add a caption on top of your graph window with the command `>> title('???')`, where `???` is your choice for the caption, and again you do type the single quotation marks. A caption along the x-axis can be added via `>> xlabel('???')` and one along the y-axis via `>> ylabel('???')`. These are added after one has obtained the graph. For example, with the graph already present, try

```
>>title('who cares')
```

```
then
```

```
>>xlabel('not me')
```

```
then
```

```
>>ylabel('me neither')
```

## 4 Printing MATLAB plots

In the window in which your plot is displayed, go to the option "File" at the top left of the window. Click on it and use the print option that appears.

## 5 The Discrete Fourier Transform (theory)

THE BASIC FACTS ON THE DISCRETE FOURIER TRANSFORM, A CONCISE REVIEW.

- Definition.** Let  $z = (z(0), z(1), \dots, z(N-1)) \in l^2(\mathbb{Z}_N)$ . For  $m = 0, 1, 2, \dots, N-1$ , define

$$\hat{z}(m) = \sum_{n=0}^{N-1} z(n) e^{-2\pi i m n / N}.$$

Write  $\hat{z} = (\hat{z}(0), \hat{z}(1), \dots, \hat{z}(N-1))$ . The map which takes  $z$  into  $\hat{z}$  is called **the Discrete Fourier Transform**, abbreviated DFT. Note that the indexing of entries in a vector here is not from 1 to  $N$ , but from 0 to  $N-1$ .

- Plancherel Formula.** We have the following identity

$$\|z\|^2 = \frac{1}{N} \|\hat{z}\|^2, \quad \text{where } \|u\|^2 := \sum_{n=0}^{N-1} |u(n)|^2.$$

Observe that this formula says that the length or the magnitude of the vector can be measured in either ways (up to normalization by  $N$ ): in the spacial domain (by  $z$ ) or in the frequency domain (by  $\hat{z}$ ).

- Definition.** For  $m = 0, 1, \dots, N-1$ , define  $F_m \in l^2(\mathbb{Z}_N)$  by

$$F_m(n) = e^{2\pi i m n / N} \quad \text{for } n = 0, 1, \dots, N-1.$$

Let  $\mathcal{F} = \{F_0, F_1, \dots, F_{N-1}\}$ . Then  $\mathcal{F}$  is called the (discrete) Fourier basis for  $l^2(\mathbb{Z}_N)$ .

*Remark:* this is an orthogonal basis, which means the vectors are "perpendicular" to each other (in the sense that the inner product = the dot product between them) will be zero.

4. **Expansion of  $z$  in the Fourier basis.** We have

$$z = \sum_{n=0}^{N-1} \hat{z}(m) F_m.$$

So we expanded the vector  $z$  in the Fourier basis with coefficients  $\hat{z}(m)$ ,  $m = 0, \dots, N - 1$ .

*In other words,  $\hat{z}(m)$  is the weight of the vector  $F_m$  used in making up  $z$ .*

## 6 The Discrete Fourier Transform (practice)

Given a vector  $z$ , its Discrete Fourier Transform (or DFT)  $\hat{z}$  is computed in MATLAB via the command

```
>>w=fft(z)
("fft" stands for "Fast Fourier Transform," the fast algorithm for computing the DFT). Try
>>z=[1 3 -2 7 8]
>>w=fft(z)
```

Note that  $w$  has complex-valued components, even though  $z$  is a real-valued vector. Hence, the warning above about plotting complex vectors applies. To see what I mean, try

```
>> plot(w).
```

Also, we would like to think of  $z$  and  $w$  as defined on  $0, 1, 2, 3, 4$ , so let

```
>>m=[0 1 2 3 4]
```

Then you can plot as follows

```
>>r=real(w)
>>plot(m,r,'x')
```

We choose an  $x$ -plot because it seems natural for short vectors; for long vectors the default is fine. Similarly you can plot

```
>>s = imag(w)
>>t=abs(w)
```

The inverse DFT is obtained via the command `ifft`. Try

```
>>p=ifft(w)
```

This should agree with the original vector  $z$ . Notice that this gives apparently complex values, even though the imaginary parts of all components seem to be 0. However, try

```
>>imag(p)
```

You will see the factor  $1.0\text{e-}14$  in front of the vector, meaning that its components are multiplied by  $10^{-14}$ . This should be exactly 0, but it isn't because of round-off error in the computations. One gets a similar small error by looking at  $z$  minus `real(p)`:

```
>>z-real(p)
```

## 7 Saving and Loading Variables

As you work in MATLAB, the variables you define are stored for future use. The command "who" tells you what variables are in your workspace. For example, define  $z$  and  $w$  by

```
>> z=0:1:10
>> w=z.^ 2
>> who
```

You should get a message telling you that your variables are  $z$  and  $w$ . A variable can be removed via the command "clear". For example, try

```
>> clear z
>> who
```

You should only have the variable  $w$  in your workspace. The command

```
>> clear
```

(with no variable name following) clears all workspace variables.

When you quit MATLAB, all variables in your workspace are automatically cleared unless you have saved some of them. If you are working on your own machine, you can save them to a location of your choosing. If you are working on a machine in a campus lab, store your data on an external memory device (such as zip drive, USB stick, etc) or copy it to your personal directory (eg. through ssh). For simplicity in what follows, I will call "a:" either your external memory drive or the directory/path where you save your data. The command

```
>> save a:\filename
```

where filename is the name of the variable you have in your workspace, saves all the variables in your workspace in a file called “filename.mat” in your directory “a:”. Then the command

```
>> load a:\filename
```

retrieves all these variables. For example, try

```
>> n512=0:1:511
>> nn512=n512.^2
```

```
>> save a:\nn512
```

```
>> clear
>> who (just to make sure you have no variables now in your workspace)
```

```
>> load a:\nn512
```

```
>> who
```

The message should tell you that you have two workspace variables, n512 and nn512. The reason n512 was saved is that it was one of your workspace variables when you typed

```
>> save a:\nn512.
```

This is a confusing feature of MATLAB, because by loading variables you may inadvertently and unknowingly overwrite variables with the same name that are currently in use. The way to avoid this is to use the command

```
>> save a:\filename variablex
```

which saves *variablex* in a file called filename.mat. If you want to save several variables in the same file, you can use

```
>> save a:\filename variablex variabley variablez
```

to save *variablex*, *variabley*, and *variablez* (or some other number of variables) in the file filename.mat. Then the command

```
>> load a:\filename
```

retrieves all three of these variables. For example, try

```
>> clear
>> n=0:1:511;
>> nn=n.^2;
>> nnn=n.^3;
```

```
>> save a:\powersofn nn nnn
```

```
>> clear
```

```
>> load a:\powersofn
```

```
>> who
```

You should now have nn and nnn in your workspace, but not n.

## 8 Discrete Convolution

In this section we discuss the discrete convolution. Suppose we have vectors  $z$  and  $w$  of the same length and we want to compute their convolution  $z * w$ . Let’s consider an example where we compute  $z * w$  for  $z = (1, 1, 0, 2)$  and  $w = (i, 0, 1, i)$ . Start with

```
>> z=[1 1 0 2]
>> w=[i 0 1 i]
```

Now MATLAB has a command “conv”, so let’s try that:

```
>> conv(z,w)
```

Unfortunately, this does not give what it should be (we need a circular convolution!). In fact, it does not even give a vector of the right length; it gives a vector of length 7. That’s because MATLAB is computing the linear convolution,

appropriate to vectors on  $\mathbb{Z}$ , not on  $\mathbb{Z}_N$  which needs the circular convolution. One of the ways to deal with that is to compute the circular convolution by using the relation

$$(z * w)^\wedge(m) = \hat{z}(m)\hat{w}(m), \text{ or } z * w = (\hat{z}\hat{w})^\vee.$$

For example, try

```
>> ifft(fft(z).*fft(w))
```

This should give the correct answer  $z * w = (2i, 2 + i, 1 + 2i, 1 + 3i)$ . There is one difficulty with this, however. To see it, let, for example,

```
>> x=[2 0 3 4]
```

and compute  $z * x$  via

```
>> u = ifft(fft(z).*fft(x))
```

MATLAB appears to give complex-valued components in the answer (for  $u(1)$  and  $u(3)$ ) even though  $z$  and  $x$  are real-valued vectors, and hence, their convolution should be real-valued. In fact, type

```
>> imag(u)
```

and you will see that MATLAB thinks that the imaginary part of  $z * x$  is non-zero, although its components are very small since the coefficient  $1.0e-16^*$  means that the values shown are to be multiplied by  $10^{-16}$ . This comes from round-off error in MATLAB. Since  $\text{fft}(z)$  and  $\text{fft}(x)$  are complex-valued, their computation requires using complex arithmetic. Then taking their product and then the inverse DFT comes out with a very small imaginary part, due to round off error. This can be a problem, especially for plotting, since MATLAB plots complex vectors by plotting the imaginary part as a function of the real part. If you know your vectors  $z$  and  $x$  are real (which will usually be the case for us), the easiest solution to this is to just take the real part:

```
>> u = real(ifft(fft(z).*fft(x)))
```

You should get the true convolution  $z * x = [6 \ 8 \ 11 \ 11]$ .

## 9 Function Files

Suppose there is some sequence of operations that you will be using often. Rather than type in the sequence every time, you can create your own MATLAB function that does this sequence for you. For example, suppose we want to compute  $z * x$  for real vectors  $z$  and  $x$  as described above, but we don't want to type

```
>> real(ifft(fft(z).*fft(x))) every time.
```

Instead we want to create a command, which in this example we call "realcon" (for "real convolution"; don't call it "conv" since that function already exists). Then we want to be able to find  $z * w$  by typing the command

```
>> realcon(z,x)
```

or, if we want to define  $u = z * x$ , type

```
>> u = realcon(z,x)
```

To do this, we need to use a texteditor to write our program on, and save the result. We will discuss where to save it below. You can use any texteditor you like (even a Notepad will suffice). Notepad can be accessed directly from the MATLAB prompt via

```
>> ! notepad
```

We now write a file which will have the name "filename.m" where filename is whatever we decide to call the function (in our case the file will be called "realcon.m"). The first line of the file must contain the word "function" followed by a space, followed by the name of the output variable (if there are multiple output variables, enclose them in brackets with commas in-between, as in  $[r,t]$ ), followed by an equal sign, followed by the name of the function ("realcon") followed by the input variable (if there are multiple input variables, enclose them in parentheses and separate by commas, as in  $(p,q)$ ). For example, in our case, the first line can be

```
function y=realcon(a,b).
```

It's best to follow this with a few comment lines, explaining what this program does for future reference. Comment lines should be preceded by the symbol %, which means that these lines are ignored when MATLAB runs the program. For example, we could have

```
% This function takes two real vectors of equal length and computes their  
% circular convolution
```

Next we have the program itself, which in our case is just one line but in the future may be more:

```
y=real(ifft(fft(a).*fft(b)));
```

It is a good idea to end every program line with ; since that way none of these lines will be printed when you run the program. So, all in all, create a file in notepad called "realcon.m" consisting of:

```
function y=realcon(a,b)
```

```
% This function takes two real vectors of equal length and computes their
```

```
% circular convolution
y=real(iff(fft(a).*fft(b)));
```

You need to save this file to an appropriate folder before you can run it in MATLAB. On some machines, you need to save the file to your external memory (e.g., the a: drive) by clicking on “File” in notepad and going to “Save as” and finding your “a:” drive. On other machines, you need to save your .m files in the folder called “work” inside the MATLAB folder (by clicking on “File” and going down the menu to “Save as”). Later, to keep your programs so that you can use them the next time you have a MATLAB session, you will need to save the file to your “a” drive. However, you may as well wait until you have tested your file and you have it in a final form before copying it to your “a” drive. Then the next time you use MATLAB, you can start by copying your files from your “a” drive to your “work” folder. Now return to MATLAB (you might need to close the texteditor before you return). Then try

```
>> clear
>> z=[1 1 0 2]
>> x=[2 0 3 4]
>> u=realcon(z,x)
```

The result should be the vector  $u = z * x = [6 \ 8 \ 11 \ 11]$  as before. There is one important thing to notice about this procedure. Although the input variables to the realcon program were called a and b, and the output was called y, these variable names are only used locally within the running of the program. They are not saved to your workspace. If you now type

```
>> who
```

you should get only the variables z, x, and u. Second, by typing “realcon(z,x)”, you automatically made z take the place of a and x take the place of b in the program. This means that you don’t need to remember the variable names used in the program; they are only used locally within that program. Thus, you have defined a new function called “realcon” which you can apply to any two vectors  $r$  and  $s$  of the same length by simply typing “realcon(r,s)”. This is a very convenient feature. Note that new functions that you have defined can be used within longer commands; they do not have to stand alone. For example, try

```
>> v=3.*realcon(z,x)
```

Working within a texteditor allows you to correct mistakes without retyping the body of the program. Suppose you create a file and save it to the appropriate folder as filename.m (all MATLAB programs should end in the suffix .m). Suppose you then try to run it and find it doesn’t work. Then type

```
>> ! notepad
```

and open the desired file from whatever location it is in. The filename ends in “.m”. Note that filename (without the “.m”) is also the name of the function that you call when running the program in MATLAB. Make sure you save your m-files before you quit MATLAB session.

## 10 Script Files in MATLAB

In Section 9 we considered function files, which are MATLAB programs that you store and use to carry out operations that you will repeat often. There is another kind of program file in MATLAB, called a script file, that is also useful. A script file is just a sequence of MATLAB commands. When you call the file, these commands will be executed just as if you typed them into MATLAB at that point. In particular, any input variable used in the script file needs to already be in your workspace with the same name at the time you execute the file, or else you get an error message. Moreover, the variables created by the script file will be part of your workspace after you execute the file. Script files are written in a texteditor and saved as “filename.m”, the same as function files, where “filename” is your choice of name for the script file. They are called via the command

```
>> filename
```

Unlike function files, you do not specify any variable names for the file to operate on; it operates on variables already in the workspace the same as if you type its commands directly. For example, start with

```
>> a=[1 2 3]
>> b=[5 4 3]
>> ! textedit
```

When the texteditor window comes up, type in it the following:

```
c=a+b
```

Comment lines can and should be used the same as for function files, but the first line starting with “function” that is needed in function files is not used for script files. The first line of a script file is usually a comment line starting with %; the comment lines are followed by the actual commands of the program. Now save this file with some name, say “simple.m”, in your same place that you save your function files (either your “a:” drive, or the “work” folder). Then go back to the MATLAB prompt (you may have to close the texteditor window) and type

```
>> simple
```

The result should be the vector  $c = [6 \ 6 \ 6]$ . If you check (via `>> who`), you should now have  $c$  as a workspace variable. If you try to run “simple” but you don’t have vectors  $a$  and  $b$  in your workspace, you will get an error message. The reason that script files are sometimes more useful than function files is that function files do not save the output variables to your workspace. When programming the wavelet transform, we will want to save the output vectors.

## 11 Compression

Suppose we have any orthonormal basis  $v_0, v_1, \dots, v_{N-1}$  for an  $N$ -dimensional vector space  $V$ . Then for any vector  $z \in V$ , we have the expansion

$$z = \sum_{n=0}^{N-1} \langle z, v_n \rangle v_n.$$

(This is just a decomposition of  $z$  in the basis  $V$ .) We are interested in approximating  $z$  using a sum of the same type but containing only  $K$  terms, where  $K < N$ . For that, we will find the  $K$  largest values of  $\langle z, v_n \rangle$ . Call  $S$  the set of indices where these largest  $K$  values occur. The best approximation to  $z$  using only  $K$  terms is  $\sum_{n \in S} \langle z, v_n \rangle v_n$ . We are interested in comparing these approximations for different compression rates  $K$ . To carry out the compression numerically, we will compute the coefficients in a Fourier basis, then find the  $K$  largest in magnitude, set all the others to 0, and then carry out the inverse transform. This gives the same result as  $\sum_{n \in S} \langle z, v_n \rangle v_n$ , since the other values have been replaced by 0. To see how to do this in MATLAB, consider the following example.

```
>> z = [-3 2 4 -5 -1 6]
```

Fortunately MATLAB has a very simple command to sort vectors in increasing order:

```
>> sort(z)
```

However, we want to consider absolute values, so try

```
>> h = sort(abs(z))
```

Suppose we want to keep the 4 components of  $z$  with the largest magnitude and zero out the two with smaller magnitude. MATLAB has a command called “find” for identifying the components of a vector where something happens. For example, type

```
>> q = find(abs(z) < h(3))
```

This will give the indices of the components of  $z$  for which the magnitude is smaller than  $h(3)$ , the third largest magnitude in  $z$ . (MATLAB gives the indices as 2 and 5 in this case, since MATLAB regards  $z(2)$  as 2 and  $z(5)$  as -1 here.) We would like to zero out the components of  $z$  with these indices. There is a command for this also:

```
>> z(q) = 0
```

Such a command sets to 0 all components of  $z$  with indices in the vector  $q$ .

Let’s try to write a general script file to zero out all but the  $K$  largest values in magnitude of a vector called  $w$  of length  $N$ , where our program will work for any given  $K > 0$ . We are tempted to just write

```
h = sort(abs(w));
```

```
q = find(abs(w) < h(N+1-K));
```

```
w(q) = 0
```

The term  $N + 1 - K$  may seem strange, but this is needed because MATLAB indexes its vectors from 1 to  $N$ . So for example if we take  $K = 1$ , meaning we only want to keep one component of  $w$ , we want to throw away all values  $w(j)$  where  $|w(j)| < h(N)$ , since  $h(N)$  is the largest absolute value of any component in  $w$ . The only difficulty with this is the problem of “ties”. For example, if  $w$  is a constant vector, then no component will have absolute value less than  $h(N - K)$ , so no values will be zeroed out, no matter what  $K$  is. To fix this, consider the example with length 9 and with  $K = 4$ :

```
>> w = [2 3 -4 4 -3 3 3 -2 1]
```

```
>> h = sort(abs(w))
```

```
>> q1 = find(abs(w) < h(6))
```

This gives the indices 1, 8, and 9 where  $|w(n)| < h(6) = 3$ . Then try

```
>> q2 = find(abs(w) == h(6))
```

The double equals sign is necessary to distinguish from the case where a vector is being defined. This gives 2, 5, 6, 7, the indices where  $|w(n)| = h(6) = 3$ . Since there are only 3 components in  $q1$ , we need to find two additional components of  $w$  to zero out, from among the four in  $q2$ . It doesn’t matter which ones we take, so just take the first 2 by defining

```
>> q3 = q2(1:2)
```

Then let

```
>> q = [q1 q3]
```

and proceed with

```
>> w(:,q)=0
```

as before.

In general, we have a vector of length  $N$  and we want to zero out  $N - K$  components, and  $\text{length}(q1)$  is always less than or equal to  $N - K$ . So we make up the difference using  $q2$  via

```
>> q3=q2(1:N-K-length(q1))
```

If there are no ties, then  $N - K = \text{length}(q1)$  and so  $q3 = q2(1:0)$ . Then  $q3$  is the empty matrix in MATLAB, and it does no harm, since then  $q = [q1 \ q3]$  just equals  $q1$ , which it should in this case. So altogether we have

```
N=length(w);
h=sort(abs(w));
q1=find(abs(w) < h(N+1-K));
q2=find(abs(w)==h(N+1-K));
q3=q2(1:N-K-length(q1));
q=[q1 q3];
w(:,q)=0;
```

To see the output (the compressed approximation to the original), type

```
>> w
```

Write a script file (called, say, “keeplarge.m”) that does this and try it on a few examples (with and without ties) to make sure it works right. Remember that one needs to define  $w$  and  $K$  prior to calling the file.

With the “keeplarge” script, compression is now relatively easy. For example, to compress a vector  $w$  of length 512 using  $K$  terms in its Euclidean expansion, it is enough to define  $w$  and  $K$  and then use

```
>> keeplarge
>> n=0:1:511
>> plot(n,w)
```

To compress  $w$  in the Fourier basis, first take the DFT

```
>> fft(w);
>> w=fft(w);
```

(This step is necessary because “keeplarge” at the next step must be applied to  $w$ .)

```
>> keeplarge
>> y=real(iff(w));
```

Here,  $y$  is the compressed approximation to  $w$ . By taking  $\text{iff}(w)$ , you have zeroed out all but the  $K$  largest DFT coefficients. Here it may not be true (even theoretically) that  $\text{iff}(w)$  is real, since zeroing out coefficients before taking the  $\text{iff}$  may destroy the symmetry property needed for the  $\text{iff}$  to be real. But since  $w$  is real, we know that any imaginary part is part of the error, so we simply do away with it.

## 12 Assignment

Access the directory <http://math.asu.edu/~svetlana/sound/> Take the original sound file (`handel100.wav`) and try compressing it with different  $K$  largest coefficients. There are samples when the compression rate was 25%, 50% and 75%. You should try different rate besides these too. Try compressing this sound file and see how the compression performs. Decide until what percentage of the compression of Fourier coefficients you can still “understand” or recognize the music. Now find or create your own wav file or get anything from the internet and try to do the same.

In case you don’t know how to use .wav files in MATLAB, here are some hints:

```
clear;
f = wavread('handel100');
w = transpose(f); (you might want to use this to organize your signal as a vector)
n = 0:1:N-1; (choose the length of your sound which is of power 2, e.g.  $N = 2^{16}$ )
Now choose  $K$  and do the compression Then you might want to plot the signal Write a title, for example,
title('Fourier Reconstruction after M % Compression')
wavwrite(y0,S,'handel-M'); (this creates a sound file from your compressed data of length S)
```