

Wavelets MATLAB Assignment 2

2.1: 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. In this course, we will construct certain filters that we will use repeatedly, so it will save a lot of time if we save 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.

2.2: Convolution

Suppose we have vectors z and w of the same length and we want to compute their convolution $z * w$. For example, let's check the result of Example 2.24 in the text, 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 we got in Example 2.24. 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} , as we will see in Chapter 4, rather than the circular convolution in Chapters 2 and 3 that is appropriate to vectors defined on \mathbb{Z}_N . One of the ways to deal with that is to compute the circular convolution is to use the relation

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

(If you know a better way, please let me know!) For example, try

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

This should give the answer $z * w = (2i, 2 + i, 1 + 2i, 1 + 3i)$ obtained in Example 2.24. 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 $z * x = [6 \ 8 \ 11 \ 11]$.

2.3: 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. One that you can access easily is the Notepad. It 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(ifft(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 (eg. 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 that we will use repeatedly.

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.

2.4: Control Flow Functions, Upsampling and Downsampling

Two operations that we will use repeatedly are upsampling and downsampling, defined in class. The easiest way to program these is to use certain "control flow" functions built into MATLAB. There are a number of these, such as "if", "else" and "while". However, we will focus on the control flow functions "for" and "end". "For" is used to repeat a command a certain number of times, depending on a certain index. For example, type

```
>> for k=1:10
x(k) = k.^2-k;
end
>> x
```

You should get the vector of length 10 with components equal to $k^2 - k$, except that MATLAB thinks that vectors start with index 1, so the first element is $1^2 - 1 = 0$ and the second is $2^2 - 2 = 2$. Note the following: (i): the first line starts with "for" and gives the limits on k; (ii) after that the MATLAB prompt >> does not appear again until you complete the commands with the command "end", which is necessary to get back to the prompt >>, (iii) it is advisable to use the semi-colon after the command defining x(k), since otherwise MATLAB displays a vector for each step of the iteration (so for example if k=1:1000, you will waste a lot of time watching numbers scroll by on the screen if you omit the semi-colon).

This can be used to perform the downsampling operation. Suppose for example that we start with

```
>> z=0:1:9
```

Try the following to downsample z:

```
>> for k=1:5
Dz(k)=z(2.*k);
end
>> Dz
```

You should get the vector [1 3 5 7 9], which is not what we want; we want [0 2 4 6 8]. The problem is that MATLAB regards the first component of z as z(1), not z(0). So z(2.*k) picks out what we would regard as the odd index components of z. We need to use the following:

```
>> for k=1:5
Dz(k)=z(2.*k - 1);
end
```

Now try

```
>> Dz
```

This should give the right answer. Since we will use it so often, it is worthwhile to write a MATLAB function file (see §2.3 above) called, for example "down.m" which performs downsampling. We would like this to work on a vector of any size. The easiest way to get this is to start (after your first line starting with "function", and your comment lines) with a line

```
n=length(z)/2;
for k=1:n
```

and then continue as before where z is the name you have given to your input variable for this function and Dz is the output variable (or you can just use y - remember that these variable names are local to the function file). Here "length(z)" is a MATLAB function that gives you the length of a vector. (If you start with a vector p whose length is an odd number $2j+1$, which you shouldn't do since downsampling is only defined on vectors of even length, down(p) will give you a vector of length j, since length(p) is $j + 1/2$ and the loop stops at j. But this isn't a real problem since you shouldn't ever apply "down" to a vector of odd length.)

With this function you can downsample any vector q of even length just by typing `down(q)`. Test a few examples to be certain that your function file is working correctly.

Upsampling can be done similarly. Create a function file called “up.m” with input variable z , output variable y , and main commands as follows:

```
n=length(z);
for k=1:n
y(2.*k-1) = z(k);
y(2.*k) = 0;
end
```

This may seem the opposite of what it should be, but remember again that MATLAB views $z(1)$ as the first component of z . Test this with a few simple examples, such as

```
>> a=[2 3 4 5 6]
>> v=up(a)
```

From now on, any vector q can be upsampled by the command `up(q)`.

2.5: The First Stage Wavelet Transform

Our goal is to compute the first stage wavelet transform of a vector $z \in \ell^2(\mathbb{Z}_{512})$. We pick 512 (and stick with it for simplicity), just because 512 is large enough to exhibit interesting examples. Of course we could do the same thing with vectors of different size (divisible by 2, of course). We will use two wavelet systems, the real Shannon wavelet system and the Daubechies D6 wavelet system. For these, we need to first obtain the basic filters u and v , which we call u_1 and v_1 , in preparation for the iteration step. We will also need \tilde{u}_1 and \tilde{v}_1 .

For the real Shannon wavelets, the formulas for \hat{u}_1 and \hat{v}_1 are given in Example 3.11 (take $N=512$). Enter these vectors - the commands “ones(1,m)” and “zeros(1,m)” may be useful, as will the fact that new vectors can be formed by concatenating old vectors (see §1.4 in the first MATLAB assignment). For example,

```
>> c=(2.^5).*ones(1,128)
```

will give you a vector with 128 components each of which is $\sqrt{2}$, which is what is needed at the start of \hat{u}_1 .

Name the vectors, say call $a = \hat{u}_1$ and $b = \hat{v}_1$. Then define the filters we want (call them “shanu1” and “shavn1” to distinguish them from the Daubechies filters) via

```
>> shanu1=real(iff(a))
>> shavn1=real(iff(b))
```

Although `iff(a)` and `iff(b)` should be real-valued, due to roundoff error they may have very small imaginary parts, which is the reason for taking the real part. We also need the tildes of these vectors (recall that $\tilde{z}(n) = \overline{z(-n)}$). These are most easily obtained by using the fact (Exercise 2.1.13) that $\hat{\tilde{z}}(n) = \hat{z}(n)$, hence $\tilde{z} = (\hat{z}(n))^\vee$. MATLAB has a built-in function that takes the complex conjugates of a vector via the command “conj(a)”, where “a” is vector you are taking the conjugate of. So we can obtain the tildes “shanu1til” and “shavn1til” of these vectors via

```
>> shanu1til=real(iff(conj(a)))
>> shavn1til=real(iff(conj(b)))
```

Once you have these filters and are sure they are correct (by doing Exercise 1), they are worth saving, say in a file called “shanwavefilt” - see §2.1 of this assignment on how to save and load variables.

The Daubechies filters u_1 and v_1 are given explicitly on p. 241 of the text. To enter these, you want to first define a , b , and c as on p. 240. Then for example, let

$$z_1 = \frac{\sqrt{2}}{32}(b + c)$$

and so on for z_2 , z_3 , z_4 , z_5 , and z_6 . Then you can define “dauu1” by

```
>> dauu1=[z1 z2 z3 z4 z5 z6 zeros(1,506)]
```

Then the tildes are most easily obtained by first taking the ffts of u_1 and v_1 and then computing $\tilde{u}_1 = (\hat{u}_1(n))^\vee$ by following the procedure above in the real Shannon wavelet case. Once these filters have been obtained, they are worth storing with names like “dauu1”, “dauv1”, “dauu1til”, and “dauv1til” in a file called something like “dauwavefilt”.

The process of computing the first generation wavelet transform is the same for any wavelet basis. Recall that the first generation wavelet coefficients of z are $\{z, R_{2^k}v_1\}_{k=0}^{255}$ and $\{z, R_{2^k}u_1\}_{k=0}^{255}$, for our case where

$N = 512$. For example, in Exercises 3, 4, 5, and 6, we use a vector z defined on p. 233 of the text. Recall that

$$\langle z, R_{2k}v_1 \rangle = z * \tilde{v}_1(2k) = D(z * \tilde{v}_1)(k)$$

and

$$\langle z, R_{2k}u_1 \rangle = z * \tilde{u}_1(2k) = D(z * \tilde{u}_1)(k).$$

Thus we only need to compute the vectors $x_1 = D(z * \tilde{v}_1)(k)$ and $y_1 = D(z * \tilde{u}_1)(k)$, which is easy with the function files “realcon” and “down” that we created above:

```
>> x1=down(realcon(z,v1til))
```

```
>> y1=down(realcon(z,u1til))
```

where v1til is either shanv1til or dauv1til and similarly for u1til. These are vectors of length 256, so they should be plotted at the points 0, 2, 4, ... 510, since these points are where the corresponding basis functions are concentrated. So it is useful to define

```
>> n1=0:2:510
```

so that to plot x1, for example, one should use “plot(n1,x1)”.

To check that this is right, you can try to recover z using the inverse wavelet transform. Recall that the inverse branches of the filter bank diagram are obtained by computing $v_1 * U(x_1)$ and $u_1 * U(y_1)$ and summing. So define

```
>> r=realcon(v1,up(x1)) + realcon(u1,up(y1))
```

where v1 is either shanv1 or dauv1 and similarly for u1. If you plot this (on $0 \leq n \leq 511$) it should look like the original z . To compare explicitly, plot z-r. One should see a factor of 10^{-14} occurring at the top on the vertical axis, to show that the difference is due only to roundoff error.

Because $\{R_{2k}u_1\}_{k=0}^{255} \cup \{R_{2k}v_1\}_{k=0}^{255}$ is an orthonormal basis for $\ell^2(\mathbf{Z}_{511})$, we have the representation

$$z = \sum_{k=0}^{255} \langle z, R_{2k}u_1 \rangle R_{2k}u_1 + \langle z, R_{2k}v_1 \rangle R_{2k}v_1.$$

If we delete the half of the terms corresponding to the higher frequencies, we obtain an approximation to z that we call the partial reconstruction of z at level -1, or $P_{-1}(z)$:

$$P_{-1}(z) = \sum_{k=0}^{255} \langle z, R_{2k}u_1 \rangle R_{2k}u_1 = u_1 * U(D(z * \tilde{u}_1)).$$

Assignment 2

Reproduce the following figures from the text. With each figure, please print out the MATLAB code you used in creating that figure. In all cases except Figure 26a, the limits on the horizontal axis are 0 to 511. The limits on the vertical axis in Figures 23a, 24b, and 25d are -1.2 to 1.2, and can be read off in the other figures. Note also that some figures use different plot types, such as ‘x’ or ‘.’, instead of ‘.’.

1.) Figures 12a and 12b

Hint: Given a vector $z \in \ell^2(\mathbf{Z}_{512})$, and k with $1 \leq k \leq 511$, one can obtain $R_k z$ as follows: define vectors $c=z(1:512-k)$ and $d=z(513-k:512)$. Then $z = [c \ d]$ (recall that MATLAB views the first component of z as $z(1)$) and $R_k(z) = [d \ c]$.

2. Figure 26a - This is the graph (in the window $245 \leq n \leq 265$) of $R_{256}dauv1$, where dauv1 is the v1 vector for Daubechies D6 basis - see §2.5 of this assignment.

3. Figures 23a and 23b - The vector z in Figure 23a is worth saving, since we will use it more later. Let’s call it “testvec” for future reference.

4. Figure 24b - This is a plot of the first generation wavelet coefficients $\{\langle z, R_{2k}v_1 \rangle\}_{k=0}^{255}$, where z is the vector plotted in Figure 23a, and v_1 is the mother wavelet for the real Shannon wavelet basis.

5. Figure 25d - This is the partial reconstruction $P_{-1}(z)$ for z as in Figure 23a and for the real Shannon wavelet basis - see §2.5 of this assignment.

6. Figure 27b - This is just like Figure 24b except that we use the Daubechies D6 mother wavelet instead of the real Shannon mother wavelet.