Fourier Transforms Worksheet

Spring 2013

Start up a copy of Matlab and get comfortable: over the course of going through this document, you will learn/review some of the basics of both Matlab and Fourier transforms. For your own sake, please do not copy/paste Matlab commands from this document, since typing them yourself is minimal extra effort and will help get these concepts "under your fingers". Matlab commands in this document will appear in a special font `like this (except not like this, since this is not a Matlab command)`. Type in and run *every* piece of Matlab code in this document; it is there for a reason. Explanations of Matlab commands are in boxes on the right. (If you are very familiar with Matlab, you can skip these boxes.) If you want a challenge, instead of copying the Matlab code from this document, try figuring it out yourself after only a glance; this might be useful if you are going through this document a second time.

Linear combinations

Construct a random vector with 100 elements and store it in the variable `a`:

```
a=randn(1,100)
```

A few comments, if you are unfamiliar with Matlab: type `help randn` *for information and other ways to use the function, and type* `doc randn` *for slightly more complete information in a new window;* `randn` *produces random numbers with mean 0, according to the normal distribution; the function* `rand` *without the n produces real numbers uniformly between 0 and 1; the parameters* `1` *and* `100` *specify that you want 1 row and 100 columns; if you end a Matlab statement with a semicolon ";" then output is suppressed; ending a statement with nothing, or with a comma, prints the result to the command window.*

Just to verify that our vector is distributed reasonably, we could plot a histogram of it:

```
hist(a)
```

For each of the 10 bins of the histogram, the y axis plots the number of values in a falling in each bin, and the x-axis represents the locations of the bins. Alternatively, `hist(a,20)` *would plot a histogram with 20 bins.) Further, just to check that* `a` *is the right size, try* `size(a)` *and make sure the output is* `1    100`.

Create a second random vector b with

```
b=randn(1,100)
```

Tip: at the command prompt you can press the up or down arrows to cycle through your command history, so you can bring up the `a=randn(1,100)` *command and just change the first letter. Even fancier, you can type the first few letters of something you entered previously, and press the up arrow for Matlab to search for the most recent command matching those letters.*

You can plot a or b with `plot(a)` or `plot(b)`. To plot them both on the same axis is a bit more tricky: we can concatenate a and b into a matrix with 2 rows and 100 columns as `[a;b]` though plotting this is not what we want. When plotting 2-dimensional arrays, Matlab plots each column separately, which would lead to 100 plots of length 2, instead of 2 plots of length 100. To get what we want, we need to *transpose* the matrix, switching rows and columns, with the single quote operator: `plot([a;b]')`

Note, for matrices with complex *numbers in them, the single quote operator does not quite do the transpose; we will get back to this later.*

Now for some fun: let's encode a "secret message" with the help of a and b: suppose our secret message is the two numbers 3 and -5; we will encode it as

```
c=3*a-5*b
```

This is called a "linear combination" of a and b, we just multiply each vector by a number and add the results together. Feel free to plot it; it will look like noise. What we are doing now is warmup for the Fourier transforms soon to come: given a linear combination of functions e^{ikx}, our goal is to "decode this" by revealing what coefficients will let us combine these functions into our target. At the moment, instead of using special functions, we are trying this process on completely random ones. After all, if you can do this "decoding process" with random vectors a and b, then how bad can functions e^{ikx} be? Throughout what follows, try to imagine a and b as sine or cosine or e^{ixk} functions, and visualize how things would change.

Imagine you and your friend have shared a and b beforehand, and now you want to secretly communicate $(3, -5)$ to them. You send them c. Your friend now knows a, b, and c, and wants to "discover the 3 and the -5" hidden in your message c.

As a first step, suppose we just had `c=3*a` and wanted to recover the 3. One way to do this is that seems more elaborate than necessary is to take the dot product of c with a, and divide by the dot product of a with itself—one could write this as $\frac{3a\cdot a}{a\cdot a}$ which is clearly 3. To find the dot product of two row vectors (vectors with 1 row and many columns), we can transpose the second vector to a column vector, and take their product. Thus the dot product of c and a is `c*a'`, and Matlab should return 3 if we try `c=3*a, (c*a')/(a*a')`

We have put two expressions on the same line, ending the first expression with a comma; if we had ended it with a semicolon, its output would have been suppressed.

Returning to our original challenge, letting `c=3*a-5*b`, we can *approximately* extract the 3 and the -5 as:

```
(c*a')/(a*a')
(c*b')/(b*b')
```

Why only approximately? Because a and b may be slightly confused with each other: their dot product will be close to zero, but not exactly 0: `a*a’` and `b*b’` will each be much larger than `a*b’` (which equals `b*a’`).

By the way, to get a more detailed sense for what is going on here, remember the dot product of two vectors is the sum of the element-by-element product of the vectors. In Matlab the element-by-element product of a and b can be computed as `a.*b`; the dot product is the sum of all the entries in this element-by-element product, which we can compute with `sum(a.*b)`. If you look at the elements of `a.*a` versus `a.*b` you will notice that the elements of each are roughly the same size, though in the first case they are all positive, while in the second case they are roughly half positive and half negative, so will tend to cancel out, leading to a much smaller dot product. Recall that two vectors whose dot product is 0 are called *orthogonal*. The takeaway message here is that vectors that oscillate around zero in different ways tend to be (close to) orthogonal to each other; while this is *approximately* true with random vectors, it will be *exactly* true with sine and cosine functions. While we were only able to *approximately* recover the coefficients 3 and -5 above, we will be able to *exactly* recover Fourier coefficients because sines and cosines of different frequencies are exactly orthogonal to each other.

We did the above calculations with vector arithmetic; as we get to more complicated examples, it will be useful to be able to do this with matrices. Below we will go through the above calculations slightly faster, with matrices. If you are unfamiliar with matrices, use this as a chance to remind yourself of the intuitions between how matrix multiplication works. It is important to keep in mind the *reason* we are doing each step, which is the same as above.

Create a matrix containing two rows, the first containing a and the second containing b:

```
ab=[a;b]
```

Note: in the top right corner of your Matlab window is a sub-window called “Workspace”, where you can double-click on any variable to examine it; if you do not see the Workspace window, you can bring it up via Matlab’s “Desktop” menu.

We can create c as above by multiplying the row vector $[3 - 5]$ by ab:

```
c=[3 -5]*ab
```

Square brackets “[” and “]” let you construct 2-dimensional arrays by concatenating in both the vertical and horizontal direction; semicolons “;” concatenate vertically, while commas or spaces concatenate horizontally.

We can compute the dot product of c with each of a and b as

```
c*ab’
```

Similarly, we can compute the dot products between all possible combinations of a and b as:

```
ab*ab’
```

Note that this is a 2×2 matrix, where the top left entry is the dot product of a with itself, the bottom right entry is the dot product of b with itself, and the off-diagonal entries are the dot product of a with b. To repeat the calculations above in matrix form, we want to divide each of the entries of `c*ab'` by each of the *diagonal* entries of the matrix just computed (try entering `diag(ab*ab')` to be clear on what is going on here):

```
(c*ab')./diag(ab*ab')'
```

This should return *approximately* `3 -5`. Look at each part of the above expression closely to make sure you understand the parts: we first compute `c*ab'`, which is a 1×2 vector; the "`./`" notation means element-by-element division; Matlab's `diag` function extracts the diagonal entries of the 2×2 matrix `ab*ab'`, and returns them as a column vector; the final single quote transposes this to a row vector so that the "`./`" operator will not return an error.

We have thus shown how to combine random vectors, and then "decode" after the fact which linear combination we used. Doing exactly these same steps below cosines and complex exponentials will give us the cosine transform and Fourier transform respectively.

One thing worth noting here is that while this element-by-element division will *approximately* extract the components $(3, -5)$, what we really want is a matrix inversion. We can invert the matrix `ab*ab'` via `inv(ab*ab')`, and the following expression should return exactly `3 -5`:

```
c*ab'*inv(ab*ab')
```

The cosine basis

We will try the process above, except instead of using random vectors, we will use cosine functions (sine functions would work equally well). To generate and plot a vector of length 100 that contains a cosine shape that oscillates three times, try:

```
plot(cos((.5:100)/100*pi*3))
```

(In Matlab, the colon operator ":" generates *ranges* of numbers; `1:5` produces `[1 2 3 4 5]` and `.5:5` produces `[.5 1.5 2.5 3.5 4.5]`. After generating the range, we multiply it by $\frac{3\pi}{100}$ to map it uniformly to the interval 0 to 3π, for three (half-) oscillations of the cosine function. As a side note, the colon operator can take a third parameter which specifies the increment: `0:.1:.5` generates `[0 .1 .2 .3 .4 .5]`.)

We can thus modify `ab` from above to consist of a vector of frequency 3 (as we just computed) and a vector of frequency 4, as:

```
ab=[cos((.5:100)/100*pi*3);cos((.5:100)/100*pi*4)];
```

Try plotting `ab` with `plot(ab')`, as above.

Now, as above, let `c=[3 -5]*ab;`, and perhaps plot all three vectors, with `plot([ab;c]')`; can you guess how to recover our "secret code" $(3, -5)$ from this plot? As above, `(c*ab')./diag(ab*ab')'`. Note that here we have recovered the coefficients *exactly*. Why? Because our basis vectors, the cosine functions, are orthogonal to each other: the matrix `ab*ab'` only has entries on its diagonal. What we have done is not quite a Fourier transform, because we used cosine functions, instead of complex exponential functions. This is called the *cosine transform*.

Before we move on, it is worth noting how convenient it is that the two cosine functions we picked, `cos((.5:100)/100*pi*3)` and `cos((.5:100)/100*pi*4)` are orthogonal to each other. If we change the frequencies 3 and 4 to a different pair of numbers, will this remain true? Yes, if the frequencies are *different* integers between 0 and 99. (Remember, vectors a and b are orthogonal if their dot product is 0; due to rounding errors, you may get something like `-8.9928e-015`, denoting that at least the first 14 digits of their dot product is 0, and numerical error is responsible for the rest.) If you are curious, try computing the dot products `ab*ab'` though replacing 3 and 4 in the definition of ab above with two other numbers of your choice.

Up until now, these manipulations have not given us more than we started: we started with a linear combination of some basis vectors, and ultimately recovered the coefficients. However, the fundamental claim of the cosine transform (and the Fourier transform) is that *any* vector can be expressed in this fashion, at least, if we express it in terms of cosines of *all* frequencies, instead of just 3 and 4. Instead of the matrix ab with just cosines of frequencies 3 and 4, we will construct a matrix M with all the frequencies. The following is a not-very-efficient way of constructing it:

```
M=[]; for j=0:99, M=[M;cos((.5:100)/100*pi*j)]; end
```

The above `for` loop should be fairly self-explanatory; it runs the code inside it for each value of j from 0 to 99. Matlab `for` loops must end with an `end`. The code starts with an empty matrix `M=[]`, and in every iteration adds a row containing a cosine of the desired frequency.

Recall from above that we used the code `(c*ab')./diag(ab*ab')'` to find what linear combination of the rows of ab produced c (provided the rows of ab were orthogonal. We compute the full cosine transform identically. If you have not changed c from above, it should still be 3 times the vector `cos((.5:100)/100*pi*3)` minus five times the vector `cos((.5:100)/100*pi*4)`, and we can see this via the cosine transform, as:

```
plot(0:99,(c*M')./diag(M*M')')
```

The `0:99` in the plot command specifies x-coordinates to plot at; otherwise, by default, Matlab would plot the 100 values from 1 to 100, which would be misleading.

We can see (possibly after zooming in) that we have completely recovered the "secret code": the cosine at frequency 3 has a coefficient of 3, and the cosine at frequency 4 has a coefficient of -5.

You can zoom into plots in Matlab by clicking the magnifying glass in the plot window. Double-clicking zooms out fully. If the plot gets messed up somehow, just run the `plot` or `imagesc` command you used to generate the graph again to reset everything.

Now that we have taken the cosine transform of something that we already know how to express via cosines, we are ready to get adventurous. Pick your favorite 100 values and store it as f, or let f be the 1×100 vector such that `plot(f)` draws your favorite shape. If you are unfamiliar with Matlab, or not feeling inspired, try `f=[zeros(1,45) ones(1,10) zeros(1,45)];`, which is a box.

We can construct a 1-row 10-column vector of ones in Matlab with `ones(1,10)`*. An all-zeros vector with 45 columns can similarly be constructed as* `zeros(1,45)`*. We place these next to each other in the above expression. Be careful, these functions with a single argument produce a square array:* `ones(10)` *yields a* 10×10 *array, which is not what we want.*

Plot f with `plot(f)`, and plot its cosine transform, by first computing the transform

```
fcos=(f*M')./diag(M*M')';
```

Then plot it with `plot(0:99,fcos)`. Remember, the claim is that this odd pattern on screen tells you *the* way to combine cosines to produce the function f. This is somewhat mysterious and far from intuitive. How do "smooth" cosines combine to form something with corners? What is the meaning of the left half of the graph (representing coefficients of low frequency components) or the right half of the graph (representing high frequency components)? Questions like this are at the heart of the Fourier transform.

To get a sense for how these cosines can do this we should take a closer look at `fcos`. The claim is that $fcos$ represents cosine coefficients, and that we can reconstruct f as `fcos*M` – try plotting `fcos*M` to convince yourself; or if you are more skeptical, compute the maximum of the absolute value of the difference between this and f with `max(abs(fcos*M-f))`.

In some sense this works so well that it is boring. To get a more interesting notion of what is going on here, we can break it: try using only the first 30 cosines, as in

```
plot(fcos(1:30)*M(1:30,1:100)
```

This is the first time we have accessed a portion of an array in Matlab. The basic syntax is fairly straightforward: for a one-dimensional array $a1$*, we can access its 5th element as* `a1(5)`*, or its 5th and 7th elements as* `a1([5 7])`*. For two-dimensional arrays make sure to use two indices—if you only use one index, Matlab will not throw an error, just try to access the memory as though it was a one-dimensional array, which is probably not what you want. There are two useful features of the colon operator when indexing an array: using the colon as an index with no parameters will index that entire dimension, so we could have indexed* M *above as* `M(1:30,:)`*; also, we can use the keyword* `end` *to index "until the last possible element", as in* `M(1:30,1:end)`*. As you might have guessed by now, Matlab indexes its arrays starting from 1, instead of 0.*

The "first 30 cosines" represent the slowest-oscillating cosine functions. You could try only the fastest 30 cosines with `plot(fcos(end-29:end)*M(end-29:end,1:100))`. For most functions, the slowly-oscillating components are visually a lot more recognizable than the others. This is what many image and video compression algorithms are based on. Play around with the above concepts and code until something "clicks".

At this point it is worth taking a look at the mysterious scaling factors that we have been dividing by, `diag(M*M')'`—remember that we have been using such scaling factors ever since the first part of this document, when we were using random vectors. The diagonal of `M*M'` stores the dot products of each of our cosines *with itself*; dividing by these entries, element-by-element was just to make sure that the resulting coefficients have the right scale. Go back and review the first section if you are confused. Plot these scaling coefficients, with `plot(diag(M*M’)’)`. It turns out that the first scaling factor is 100, and all the rest are 50. Not so mysterious after all. In future, instead of computing this matrix product, we will use the much simpler expression `[n n/2*ones(1,n-1)]`, which is equivalent but clearer.

Cosine transform of an image

Go to the public directory in the course folder [???], and then load a picture of a hobbit:

```
im=imread(’hobbit.jpg’);
```

(Note: you can feel free to do this with an image of your choice. Make sure to use a variant of the operation in the next paragraph to crop it to a square shape and extract the grayscale component.)

Matlab stores the image as a 3-dimensional array, with separate 2-dimensional slices for the red, green, and blue color information. We sum along the third dimension to get a grayscale version of the image, and also crop it to be conveniently square-shaped:

```
im=sum(im(:,end-397:end,:),3);
```

When Matlab displays arrays as images, it makes use of a “colormap”, which in this case should map 0 to black, and big numbers to white.

```
colormap(’gray’)
imagesc(im)
axis image
```

The `imagesc` command automatically scales the values in the image so that the smallest maps to black and the largest maps to white. (If you want, you can type `colorbar` to add a legend to the side of the image saying what numbers map to what colors.) The final command, `axis image`, tells Matlab to “make the axes appropriate for an image”, namely make sure the pixels are square.

We can now try some cosine transforms on this image. The image is 398×398, which means we should recalculate our transform matrix M:

```
n=398;M=[]; for j=0:n-1, M=[M;cos((.5:n)/n*pi*j)]; end
```

As mentioned at the end of the last section, instead of computing `M*M’` each time we compute coefficients, we can instead just use the vector `[n n/2*ones(1,n-1)]`—assuming you do not change the variable n in the interim. Thus to take the cosine transform of all the rows in the image, we multiply the image array by `M’`, and then divide each row element-wise by `[n n/2*ones(1,n-1)]` (recall that multiplying a matrix A by a matrix B can be thought of as multiplying each row of A by the matrix B). In order to rescale the elements of each row, we could use a for loop, but it might be conceptually easier to just replicate the scaling factor `[n n/2*ones(1,n-1)]` so that there is one copy of it for each row. One clever way to do this is to multiply `ones(n,1)` by it, leading to a cosine transform of the rows of `im`: `imR=(im*M’)./(ones(n,1)*[n n/2*ones(1,n-1)]);`

Feel free to look at our result now, with `imagesc(imR)`, but we are not done yet: we also need to cosine-transform each column. The easiest way to do this is to flip the rows and columns of `imR`, do the cosine transform on each new row, and then flip rows and columns back: `imcos=((imR’*M’)./(ones(n,1)*[n`

You can look at this with `imagesc(imcos)`, though you may need to zoom in on the top left corner before you see anything: only the low-frequency cosine components of an image have much action. To see something about the whole structure of the cosine transform, you might have to "squash" the range, for example, by displaying the fourth-root of the absolute values:

```
imagesc(abs(imcos).^.25)
```

The Matlab operator ".^" takes the element-by-element power of an array.

We can recover the original image by multiplying by M, then transposing and multiplying by M again (to transform the columns), and then transposing again:

```
imagesc(((imcos*M)'*M)')
```

However, it might be interesting to modify the cosine transform before we invert it. We will use `imcosm` to play around with modifications. We can, for example, take the first 30×30 cosine components and make them 0:

```
imcosm=imcos; imcosm(1:30,1:30)=0; imagesc(((imcosm*M)'*M)')
```

This shows us only the high-frequency, rapidly-oscillating parts of the image. Alternatively, we could see only the components that are oscillating slowly in their first coordinate (vertically, in Matlab), with:

```
imcosm=imcos; imcosm(31:end,:)=0; imagesc(((imcosm*M)'*M)')
```

Note how the edge of Frodo's sword is still sharp, as is the corduroy pattern on his jacket on the left of the image. Play around with this for a bit. For example, one other odd effect, is if we *negate* the small cosine components:

```
imcosm=imcos; imcosm(1:30,1:30)=-imcosm(1:30,1:30); imagesc(((imcosm*M)'*M)')
```

One bit of terminology, so that you can speak with the rest of the world about what you have learned: the process of expressing a signal in terms of cosines as we have been doing is called the *inverse (discrete) cosine transform*, which for our vector c we computed as `(c*M')./diag(M*M')`, while the process of taking coefficients and computing the corresponding linear combination of cosines is called the *(discrete) cosine transform*, which we computed by just multiplying by M.

Fourier transforms

Now that we have seen the cosine transform, the Fourier transform can be seen as just a minor variation. The Fourier transform uses *complex numbers*, however, which can add some headaches. In particular, one thing to be careful of is that the single-quote operator we have been using to transpose matrices in Matlab actually computes the *complex conjugate of the transpose*, which turns out to be fairly useful. But if we want just the transpose, add a dot in front of the quote, as in `M.'`

Complex numbers are also harder to visualize. Depending on the circumstances, to visualize a complex array such as `z=[1 2 3+1i 4i]`*, you might want to use* `plot` *or* `image` *with its real part* `real(z)`*, its imaginary part* `imag(z)`*, or its magnitude,* `abs(z)`*.*

While the cosine transform involved expressing a signal as a linear combination of cosines, $\cos(kx)$, for different k, the Fourier transform involves expressing a signal as a combination of the complex analog of this, $e^{-ixk} = \cos(xk) - i\sin(xk)$ for different k. For example, the function that oscillates 3 times around the unit circle between 0 and 100 is:

```
z=exp(-1i*(0:99)/100*2*pi*3)
```

While you are free to redefine the variable `i` *to anything you want, the notation* `1i` *will always refer to the imaginary number i. In Matlab,* `exp` *computes e to the power of its argument, even if it is complex.*

These expressions, where we change 3 to any value between 0 and 99, are known as the *Fourier basis.*

We can now plot the real and imaginary parts of z with `plot(0:99,[real(z);imag(z)]')` (real is in blue, imaginary is in green), or we can plot them in the complex plane even more simply, with `plot(z)`, which is equivalent to `plot(real(z),imag(z))`. Since our function goes around the unit circle three times, if you zoom in you should be able to see that the circle is actually drawn 3 times.

It is worth noting that the negative sign inside the exponent is a convention, and different definitions of the Fourier transform will involve different signs, possibly multiplying the answer by n or $\sqrt{n}$ or something involving π. None of these variants fundamentally change what is going on.

In analogy with how we constructed the matrix M for the cosine transform above, we will now construct the corresponding matrix for the Fourier transform:

```
n=100;F=[]; for j=0:n-1, F=[F;exp(-1i*(0:n-1)/n*2*pi*j)]; end
```

The functions in the rows of this matrix are the Fourier basis. Comparing this with our construction of M above, we see there are a few minor differences: the range inside the exponential starts at 0 instead of .5, and there is an extra factor of 2, but other than that, the Fourier transform is *very* similar to the cosine transform.

To do a Fourier transform, that is, given 100 coefficients, computing the corresponding linear combination of the basis functions, can be done as above simply by multiplying by F. To convince ourselves that we are doing the right thing, we can create a random vector r, and compare our Fourier transform, to Matlab's built-in `fft` function:

```
r=rand(1,100); rf=r*F; max(abs(fft(r)-rf))
```

Computing the inverse of this, namely, figuring out how to express a function as a linear combination of the Fourier basis, is the same as above: given rf, we compute:

```
r2=(rf*F')./diag(F*F')';
```

This should now be very close to r, which we can verify with `max(abs(r-r2))`. Actually, if you look at the scaling factors `diag(F*F')'` you will see that they are all 100, namely n, so we can simplify our expression for the inverse Fourier transform to just: multiply by F' and divide by n.

While in the previous section we took the cosine transform of our image, we can now take the Fourier transform. First construct the matrix for $n = 398$:

```
n=398;F=[]; for j=0:n-1, F=[F;exp(-1i*(0:n-1)/n*2*pi*j)]; end
```

And then transform the rows, transpose it (with `.'` now since the values are complex), transform again, and transpose back:

```
imF=((im*F).'*F).';
```

We can display the results, as above, with `imagesc(abs(imF).^.25)` where the .25 power is to compress the range to where we can see it. Note that, unlike for the cosine transform, there is now action in all four corners of the image. This is because, while the top left corner of the image still corresponds to low frequencies in the x and y directions, the bottom right corner corresponds to low *negative* frequencies, and the top right corner corresponds to low negative frequencies in the x direction and low positive frequencies in the y direction. It makes intuitive sense to "circularly shift" the image so that all four corners are plotted together, which we can do with `imagesc(circshift(abs(imF).^.25,[199,199]))` where the parameter `[199,199]` means to shift by 199 (which is $n/2$) in each dimension. To get the figure from class, change the colormap to `colormap('hsv')`.

Structure of the Fourier transform matrix

We will take a closer look at the matrix F we have constructed. We will start with a very small version:

```
n=4;F=[]; for j=0:n-1, F=[F;exp(-1i*(0:n-1)/n*2*pi*j)]; end; F
```

In row j, column k (indexing from 0), the value will be $e^{-2\pi ijk/n}$, which, if we let ω ("omega") be defined to be $e^{-2\pi i/n}$, means that the (j, k)th entry of F is just ω^{jk}. The exponents form a multiplication table. Further, the matrix F is symmetric—swapping j and k (swapping rows and columns) does not change anything. The Fourier transform is "multiplication by a matrix consisting of ω raised to powers that form a multiplication table". Remember that ω itself is what is called a *primitive nth root of unity*, meaning that if you raise it to the nth power, it will be 1. ("Primitive" means that no power smaller than n will take ω to 1.)

Since multiplication by F gives the Fourier transform, and multiplication by F'/n gives the inverse Fourier transform, it turns out that the Fourier transform and its inverse are really almost the same thing: the only difference between F and F' is that F' is the complex conjugate of F (since transposing F does not change it).

Thus in Matlab, instead of using the inverse Fourier transform function `ifft`, we could instead just take the complex conjugate of the `fft` of the complex conjugate of our vector, and divide by the length of the vector. We can compare these two approaches for a random complex vector r:

```
n=100; r=rand(1,n)+1i*rand(1,n); max(abs(ifft(r)-conj(fft(conj(r)))/n))
```

Convolution

Remember convolution: given two vectors a and b, then their convolution is the result of the following pseudocode: create an all-zeros vector c, and for each j indexing a and each k indexing b, add $a(j)b(k)$ to $c(j+k)$. For example, try to make sense of the result of:

```
conv([1 1 1 1],[1 1 1])
```

The basic fact underlying this section (and in fact a lot of the practical usefulness of convolution) is that convolution becomes element-by-element multiplication in Fourier space.

Recall from class that we took an image and created two copies of it, separated horizontally by 100 pixels, by convolving the image with the vector `[1 zeros(1,99) 1]`. We can do this in Fourier space as follows (remember to reset `colormap('gray')` if you had changed the colormap earlier). First we will create the *convolution kernel*, which has two 1's in it, distance 100 apart horizontally, and has the same size as the image:

```
kern=zeros(398,398); kern(1,1)=1; kern(1,101)=1;
```

Then we can convolve this with `im` and plot the results as:

```
imagesc(ifft2(fft2(im).*fft2(kern)))
```

(Note the "2" in the Fourier transform calls, which tells Matlab to transform along both dimensions; if you forget these 2's, your code will produce odd results.)

You should now be looking at two copies of your image, 100 pixels horizontally shifted from each other. One slightly strange thing is that the right edge of the image wrapped around to the left, for the copy that we shifted right by 100 pixels—we might have hoped it would expand the image to be 100 pixels wider in the horizontal direction, but of course the Fourier transform does not change the size of the arrays it operates on. (On your homework you will need to implement convolution where you expand the arrays manually).

Try coming up with your own convolution kernels:

```
kern=zeros(398,398); kern(1:10,1:10)=1;imagesc(ifft2(fft2(im).*fft2(kern)))
```

The above kernel blurs the image by taking averages of 10-by-10 blocks of pixels (though note how the bottom of the image wraps around and bleeds into the top).

```
kern=zeros(398,398); for j=1:10,kern(j,j)=1;end;imagesc(ifft2(fft2(im).*fft2(kern)))
```

The above kernel blurs the image diagonally.

```
kern=zeros(398,398); kern(1,1)=1; kern(1,2)=-1;imagesc(ifft2(fft2(im).*fft2(kern)))
```

Negative numbers lead to more interesting effects: the above kernel computes the difference between horizontally adjacent pixels, effectively becoming a "vertical edge detector".

```
kern=zeros(398,398); kern(1,1)=1; kern(2,1)=-1;imagesc(ifft2(fft2(im).*fft2(kern)))
```

Switching things slightly leads to a vertical edge detector. Or, for something a bit stranger, if you are looking at Frodo, we can build a Frodo's sword detector by using convolution to try to match a section of Frodo's sword:

```
kern=zeros(398,398); sword=im(199:-1:100,379:-1:320);kern(1:100,1:60)=sword-mean(sword(:));
imagesc(ifft2(fft2(im).*fft2(kern)))
```

One of the most intriguing features of convolution via this Fourier transform setup is that this setup also lets you *invert* the operation. Suppose you know an image has been blurred in a certain way (perhaps motion-blurred in a certain direction?) then, instead of multiplying by the Fourier transform of the kernel, we can divide by it to invert the operation. In some sense, you can invert blur filters to get sharpening filters. Be careful though, if any of the Fourier coefficients of your kernel are 0 or small, then dividing by it could lead to divide-by-zero errors, or numerical blowup in other ways. One particularly useful application of this is to solve differential equations: derivatives are in some sense convolutions (we computed "cheap" derivatives above to detect horizontal and vertical lines), so if f' and f'' can be expressed as convolutions of certain kernels with f, (and of course f itself is the convolution of the trivial kernel 1 with f), then any linear differential equation can be represented via convolution as $conv(f, K) = g$, and we can solve the differential equation by taking the inverse-convolution of g with K to find f.

Divide and conquer algorithm for Fourier transforms

Thus far, we have computed Fourier transforms either by multiplying by the square matrix F in quadratic time, or by using Matlab's built-in `fft` function. In this section, we will see how to compute Fourier transforms in $O(n \log n)$ time, via a divide and conquer algorithm. This algorithm will only work for transform sizes that are powers of 2, though related algorithms work well for products of small primes.

Recall the matrix F that we had constructed, for various sizes, to help us compute the Fourier transform. The goal of this section is to understand F so that we can multiply by F in a divide-and-conquer fashion that takes $O(n \log n)$ time. We will construct F in an easier way than we did above, now that we understand what F means, we can cheat and use Matlab's built-in `fft` function:

```
F=fft(eye(8))
```

In Matlab you can construct an identity matrix of size 8 *via* `eye(8)`*, which has ones on the diagonal and zeros everywhere else.*

The `fft` function transforms each of the rows of this matrix. Essentially, the first row of F tells us how Matlab transforms entries in the first column of a vector; the second row of F tells us how Matlab transforms entries in the second column of a vector, etc., which is exactly how we defined F above. (As a side comment, the `fft` function actually transformed each *column* of its input, not each row, but since F is symmetric, this is the same thing.)

The first thing we note about F is that the submatrix consisting of rows 1,3,5,7 (indexed from 1) and the first four columns, is the same as the submatrix of F consisting of the *last* four columns of these rows:

```
F(1:2:end,1:4)
F(1:2:end,5:end)
```

We can easily prove this fact by recalling the definition of F from the "Structure of the Fourier transform matrix" section, namely that for $\omega = e^{-2\pi i/n}$ being an nth root of unity, the (j,k)th entry of F, indexing *from 0*, is ω^{jk}. The claim is that for even j, and $k < \frac{n}{2}$, $\omega^{jk} = \omega^{j(k+\frac{n}{2})}$. Since j is even, $j\frac{n}{2}$ in the exponent is a multiple of n, and since $\omega^n = 1$, this term will vanish, leaving just ω^{jk} as claimed.

This is progress towards our goal of a divide-and-conquer algorithm, that two parts of the multiplication by F are actually the same. But what we would really like is for them to be related to a Fourier transform of a *smaller size*. As it turns out, we are in luck: the matrix F for Fourier transforms of size 4 is

```
fft(eye(4))
```

And this matrix is identical to the two sub-matrices of the 8×8 transform matrix we computed previously. Thus we have divided the 8×8 Fourier transform matrix into four 4×4 submatrices (though we were a bit tricky with the rows), and we have looked at two of these submatrices and found that they are the 4×4 Fourier transform matrix, which means our divide-and-conquer algorithm can solve all of these recursively.

Now we need to worry about the two other portions of F, the even rows (or odd rows, if you are thinking in terms of 0-indexing):

```
F(2:2:end,1:4)
F(2:2:end,5:end)
```

There are two disasters here: first, these two matrices are not equal, and second, neither of them equals the 4×4 Fourier transform matrix.

The first disaster, though, is easily averted: the second matrix is just the negation of the first matrix. And the second disaster turns out to be as benign: each row of these matrices is a fixed multiple of the corresponding row of the 4×4 Fourier transform matrix, meaning that to compute the product of a vector with one of these matrices, we just multiply the vector by the 4×4 Fourier transform matrix, and then multiply each answer by the appropriate scale. Perhaps you can figure out by now why the scaling coefficients should be $(1, \omega, \omega^2, \omega^3)$, with the pattern extending when $\frac{n}{2}$ is larger than 4.

We will write a function `myfft` to implement the full divide-and-conquer Fourier transform algorithm (for vectors whose length is a power of 2). To create a new function, type `edit myfft`, enter the following code, and save it.

```
function x=myfft(x)
n=length(x);
omega=exp(-1i*2*pi/n);
scale=omega.^(0:n/2-1);
if n>1,
    even=myfft(x(1:2:end));
    odd=myfft(x(2:2:end));
    x=[even+scale.*odd, even-scale.*odd];
end
```

This function accepts one input, which will be called `x` *locally, and whatever value is in* `x` *when the function exits will be the output of the function. Type* `doc function` *for much more information about declaring functions. Like* `for` *loops,* `if` *statements also must be matched with an* `end`*. Note that this code only accepts row vectors as arguments, unlike Matlab's* `fft` *which can operate on each row of a multi-dimensional array, and many other things.*

Spend some time and try to figure out how this code corresponds to the above explanation; or write your own code if that would be easier; or check Wikipedia, or your Dasgupta et al. textbook. We can check that the code is doing the right thing by comparing it with the built-in `fft`, on a vector of length, say, 1024:

```
r=rand(1,1024); max(abs(fft(r)-myfft(r)))
```

Congratulations on making it through this worksheet! You are now ready to do your homework.