Fourier Transforms Worksheet

Fall 2014

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.

Fourier transforms preview

Matlab has built-in functions for computing the Fourier transform and its inverse. By the end of this worksheet you will understand some of what Fourier transforms can do. For now, try one out to make sure it works. Create a column vector with 4 elements, filled with your favorite numbers, for example:

```
c=[3;5;2;1]
```

> *Square brackets* `[` *and* `]` *let you build up vectors or matrices in Matlab by concatenating what is between them. Semicolons "*`;`*" concatenate elements vertically, while spaces or commas concatenate elements horizontally.*
>
> *The semicolon may also be used, optionally, at the end of a Matlab command to suppress output from the command. Otherwise, the results of commands are displayed to the command window.*

Take the Fourier transform of `c`:

```
fft(c)
```

This will output a vector of the same size as `c`, with 4 rows and 1 column, of complex numbers. We can check that the *inverse* Fourier transform properly inverts the effects of the Fourier transform. The following expression should return `c` to us:

```
ifft(fft(c))
```

At the end of homework 1 we saw a matrix for implementing the Fourier transform of length 4. The matrix was composed of a 4th root of unity, ω, raised to powers that formed a multiplication table. We will reproduce that here. There are many ways to generate such a table in Matlab, but one of the easiest is to use vector-vector multiplication. Recall that multiplying a column vector by a row vector yields a matrix. Thus we compute our multiplication table as 0, 1, 2, 3 concatenated *vertically*, multiplied by 0, 1, 2, 3 concatenated *horizontally*:

```
mt=[0;1;2;3]*[0 1 2 3]
```

We now define our 4th root of unity, which will be -1i, and raise it to the power of each element of our multiplication table (why is there a negative sign? convention. There are several Fourier transform conventions, and they differ by small things like signs, and factors of $n = 4$):

```
w=-1i
M=w.^mt
```

> *The variable* `i` *can be used like any other in Matlab, but when it immediately follows a number, as in* `1i` *above, it always represents the imaginary number* i.
>
> *Make sure you remember the dot in the operator* `.^` *—in Matlab, adding a dot before an operator makes the operator act element-by-element, instead of on the entire matrix. Without the dot, Matlab would try to raise the number* ω *to a "matrix power" using matrix exponentiation, which is definitely not what we want. Similarly, be aware of the difference between* `*` *which is matrix multiplication, and* `.*` *which multiplies each element separately; and* `/` *is matrix division or linear equation solving, while* `./` *is element-by-element division.*

We can now see if the matrix `M` correctly reproduces the Fourier transform. Multiply it by the vector `c` from above, and compare with the Fourier transform of `c` computed above:

```
M*c
```

Further, as you computed at the end of homework 1 the inverse of the Fourier transform is produced by the complex conjugate of the matrix `M`, divided by $n = 4$ (the size of the transform). Thus the following should get `c` back:

```
conj(M)*M*c/4
```

> *The Matlab function* `conj` *computes complex conjugates.*

Polynomial values and coefficients

Now that we can work with Fourier transforms, let us go back to basics and see where they come from.

There are two rather different ways of looking at Fourier transforms, which are completely equivalent. The Fourier transform describes

- The amount of energy at each *frequency* of its input, represented as a complex number that stores a magnitude and an angle (a "phase", that differentiates between shifted versions of the same signal, like $sin(x)$ versus $sin(x + 0.123)$); the Fourier transform is a way of converting signals between "regular" representation, and a "frequency" representation.
- The Fourier transform converts between polynomials represented by their (first n) coefficients and polynomials represented by their values (at all the n^{th} roots of unity).

We will start (as the Dasgupta et al. textbook does) with the polynomial formulation. Instead of starting with roots of unity, we will pick real numbers for the moment since they are easier to work with.

We will interpret the vector `c` from above as *coefficients of a 3rd degree polynomial*. Reading from smallest to largest, we interpret `c=[3;5;2;1]` to represent the polynomial $3+5x+2x^2+x^3$. We now represent `c` in a different manner, by its values at 4 points—recall that any 3rd degree polynomial is uniquely determined by its values at 4 points. Pick 4 x–coordinates now, and store them in a column vector x. For example:

```
x=[0; 1; 3; 4]
```

We must now evaluate our polynomial at these 4 points described by `x`. One easy way is to construct a table of all the powers of these x–coordinates, 0 through 3, and then multiply by the coefficients `c`. Make sure you understand why this does what it should.

```
P=[x.^0 x.^1 x.^2 x.^3]
y=P*c
```

> *It is tempting to try generating* `P` *by something like* `P=x.^[0 1 2 3]`*, but this will give a Matlab error because what we really need is for Matlab to "expand each of the singleton dimensions" of* `x` *and* `[0 1 2 3]` *before the power operation. Luckily, there is a function built into Matlab for this purpose,* `bsxfun`*, the "Binary Singleton eXpansion FUNction", which takes as arguments a handle to a binary function, and then the two arguments to be expanded. In Matlab, function handles are denoted with the* `@` *symbol, so we can generate* `P` *as* `P=bsxfun(@power,x,[0 1 2 3])`

We have thus computed the y–coordinates of our polynomial at the x–coordinates in the vector `x`. Note that however we change the coefficients c, the result y is just a linear transformation of c (depending on x). It is perhaps useful to see this process at "high resolution" and plot the results. Instead of evaluating our polynomial at 4 points we can evaluate it at a high resolution grid:

```
xh=(-5:.01:5)'
Ph=bsxfun(@power,xh,[0 1 2 3])
plot(xh,Ph*[3;5;2;1])
```

> *The colon operator generates* **ranges** *in Matlab. Type* `help colon` *for details. With two arguments,* `0:3`*, the colon operator generates the range* `[0 1 2 3]`*. With three arguments, the middle argument is the* **increment***. Thus* `xh` *stores 1001 numbers, starting at -5, ending at 5, each number* .01 *higher than the previous one. The single quote at the end computes the* **transpose** *so that* `xh` *becomes a column vector.*
>
> *The plot function has many forms, but here we give it simply two arguments: the x–coordinates and the y–coordinates to plot.*

Replace the vector `[3;5;2;1]` above with different variants to try plotting your favorite 3rd degree polynomials. It is slightly counterintuitive that matrix multiplication lets us do this. Examine the matrices involved if you are confused how this works.

Now that we can evaluate polynomials, converting from coefficient form to evaluated form, we try the reverse process. Matlab can easily invert this process using matrix inversion, which should recover our original `c`:

```
inv(P)*y
```

We can thus convert back and forth between polynomial coefficients and polynomial values.

Convolution

One of the main reasons for talking about Fourier transforms in terms of polynomial evaluation is that it makes the following trick very clear: *Fourier transforms let us multiply polynomials very quickly.*

Consider two simple polynomials: $1 + x$ and $1 + 2x - x^2$, which we can write in the format of the previous section as `[1; 1; 0; 0]` and `[1; 2; -1; 0]`. Multiplying these polynomials looks to be a quadratic-time operation, as we need to multiply all pairs of coefficients. However, if we express polynomials via their values instead of via their coefficients, then multiplying involves just multiplying their values, which is linear time!

Let's try this. Store these two lists of coefficients as vectors:

```
c1=[1; 1; 0; 0]
c2=[1; 2; -1; 0]
```

Evaluate these polynomials at 4 points x using the matrix P computed above:

```
y1=P*c1
y2=P*c2
```

Now we can multiply the *values*, element-by-element:

```
y1.*y2
```

And invert the transform to recover the coefficients of the polynomial that is the product of $1 + x$ and $1 + 2x - x^2$:

```
inv(P)*(y1.*y2)
```

What goes wrong if the total degree of the two input polynomials is more than 3?

We can also check our work with Matlab's built-in convolution function:

```
conv(c1,c2)
```

Fourier transforms, again

The Fourier transform is exactly the above polynomial evaluation/recovery process, but on a very special set of x–coordinates: the complex nth roots of unity. Try setting `x=w.^[0;1;2;3]` and then recomputing the matrix `P=bsxfun(@power,x,[0 1 2 3])` as above.

> *Tip: at the command prompt you can press the up or down arrows to cycle through your command history. 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. For example, type* `P=` *and press the up arrow.*
>
> *The colon operator lets us construct* `[0 1 2 3]` *as* `0:3`*, but for the corresponding column vector, use the transpose operator, which is a single quote. Thus* `x=w.^(0:3)'` *would be equivalent to the above. Actually, the single quote operator does not exactly transpose: it take the complex conjugate of the transpose, which is usually the same thing. Type* `help punct` *for help on the "punctuation" symbols.*

The matrix P should equal the Fourier transform matrix M that we started with, the element-wise power of ω to the "multiplication table" `(0:3)'*(0:3)`—can you see why? Multiplying by P should produce the Fourier transform. (Check it! Remember how?) In general, for a Fourier transform of size n, we can generate the matrix M as follows:

```
n=100
mt=(0:n-1)'*(0:n-1);
w=exp(-1i*2*pi/n)
M=w.^mt;
```

> *Note that we end some lines with a semicolon to prevent the screen from getting filled up with output. Double-click on the variables in the "Workspace" display in the top right to see their contents. The* `exp` *function computes the exponential function, as you would expect, and* `pi` *stores* π *by default.*

We can plot parts of `M` to see what it looks like. Remember, the real part will look like cosines, and the complex part will look like sines. Plot the 3rd, 10th, and 96th columns of `M` as follows:

```
plot(real(M(:,[3 10 96])))
```

> *We have accessed a submatrix of* `M` *here. The colon operator in an array access can have no parameters, in which case it grabs "all valid indices". So the above command takes all the rows of* `M`*, and columns 3, 10, and 96. Type* `legend show` *to label the plot.*

The 3rd column of `M` corresponds to the "2" column from the multiplication table, namely, the roots of unity squared. Thus the plot oscillates twice before returning to its starting point. The 10th column oscillates 9 times. The 96th column, interestingly, has real part identical to the 6th column and thus oscillates 5 times. (Can you see why?)

One of the classic applications of Fourier transforms is to take a signal that is a mixture of different frequency components and "descramble" them, cleanly describing the recipe for constructing the signal as a mixture. Try the code below, which mixes together a frequency 2 signal, a large amount of the frequency 9 signal, and a frequency 95 (or -5!) signal, and then uses the inverse Fourier transform to reveal this recipe:

```
plot(real(ifft(M(:,3)+M(:,10)*2+M(:,96))))
```

As you may have seen in a linear algebra course, taking the dot product of vectors `a` and `b` is often called "taking the component of `a` in the direction of `b`". Thus if the vectors that comprise our Fourier transform matrix look like sines and cosines of all the different frequencies, what the Fourier transform computes is the "component of our input in all the different frequencies", which is exactly the other notion of Fourier transforms.

While any x–coordinates work for most of what we have done so far, the special thing about the nth roots of unity is that there is an $O(n \log n)$ algorithm for computing the Fourier transform. Further, as we have seen, inverting this transformation is no harder than computing it. We will not emphasize the Fourier transform algorithm in class, because the *applications* of the Fourier transform are more interesting than the algorithm itself. The algorithm is a divide-and-conquer algorithm, in chapter 2.6 of the textbook, and has a form rather similar to what you will find for the answer to the "John Mayer's matrix multiplication" problem on homework 2.

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