The Fast Fourier Transform Algorithm (FFT)

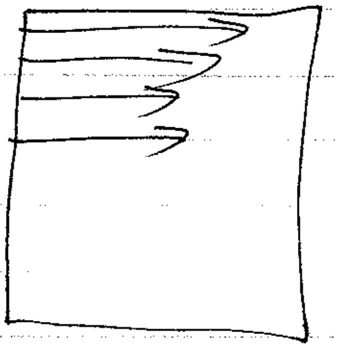
We mentioned before that it requires about $2n^3$ operations to calculate the Fourier transform of an image of size n directly. Suppose we wish to obtain the transform of a 4096×4096 image on a computer capable of 100,000,000 complex operations per second (a reasonably fast computer). Then the transform requires

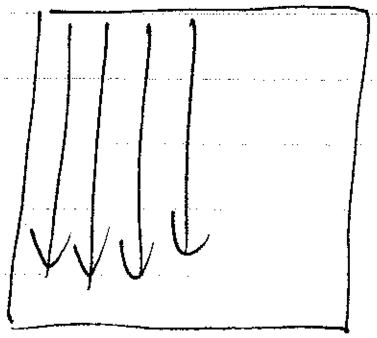
$$\frac{2 \cdot 4096^3}{10^8} \text{ seconds} = 1{,}374 \text{ seconds}$$

or about 23 minutes to compute. This is kind of slow!

Fortunately, there exists an algorithm called the Fast Fourier Transform that speeds this up considerably. For our purposes we want only to use this algorithm, so we will concentrate on how we do so, but for comparison, we note that a 1-D transform that required n^2 operations for direct integration requires only $\frac{n}{2}\log_2 n$ operations if the special fft algorithm is used.

For a two-d calculation, again we transform first the rows and then the columns:


rows, n transforms


Columns, n transforms

$2n$ transforms are needed, each with $\frac{n}{2} \log_2 n$ operations, for a total of $n^2 \log_2 n$ for the complete solution. In our previous example, we'd need

$$\frac{4096^2 \cdot \log_2 4096}{10^8} = \frac{4096^2 \cdot 12}{10^8} = 2 \text{ seconds}$$

Quite an improvement!

<u>Using the FFT</u>

The FFT accepts as input a string of numbers and outputs a second string of numbers corresponding to the Fourier coefficients of the transform. Many existing subroutines for FFTs require additional inputs giving the length of the transform and whether it is a forward or inverse calculation.

Fortunately, Matlab has a very simple call,

```
a = fft(b);
```

and for the inverse,

```
b = ifft(a);
```

The length is simply the length of the input vector. If the length is a power of 2, the calculation is especially fast. If it is not, a slower calculation is needed.

In 2-D, the corresponding calls are

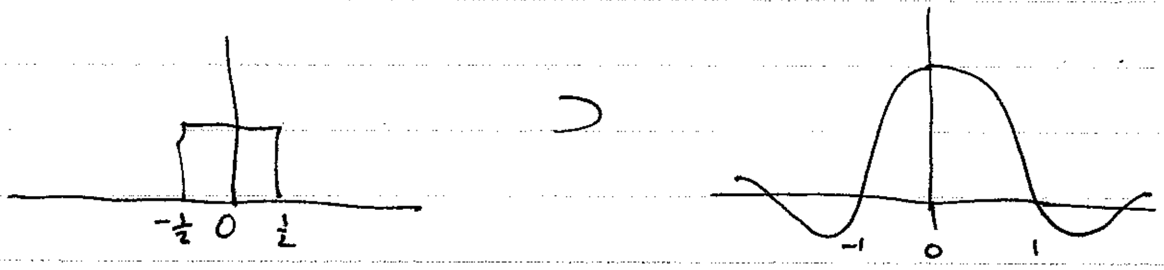
```
a= fft2 (b);
```

and

```
b= ifft2 (a);
```

<u>Where is zero in each domain?</u>

When we have plotted functions and their transforms, we have used the convention that zero appears in the center of the plot. We use this in both domains.

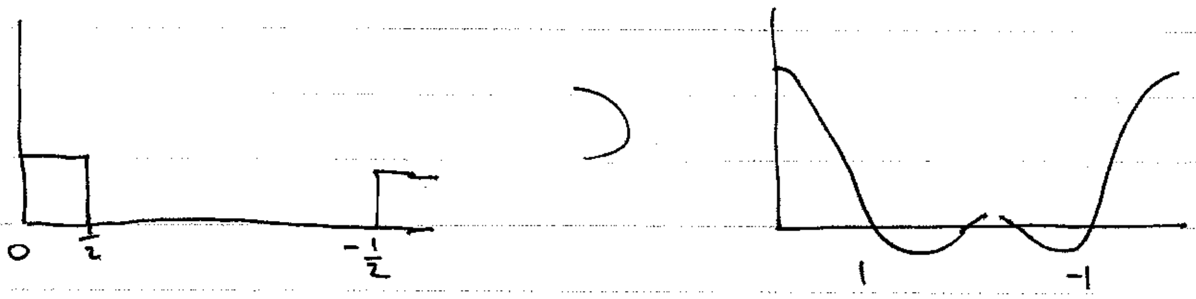


Space domain - rect (x)

Frequency domain - sinc s

However, the fft algorithm plots things differently - the very first bin is either time zero or frequency zero.

To make things even more confusing, negative frequencies or time are "wrapped" around the other end of the array. So as far as the fft is concerned, the above ought to look like

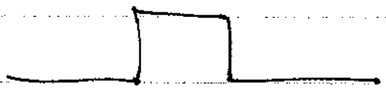


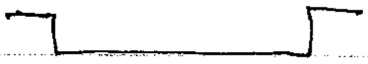
time domain -
rect(x)

sinc s in
frequency domain

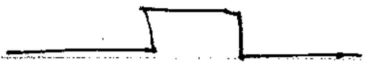
So, we have to mentally map functions to visualize correctly, and physically move things to make our image look "right."

Again, Matlab helps us here with a function called "fftshift". This routine takes a one-d or two-d function and does the mapping for us, as

a= rect (x) ⇒ 

b= fftshift (rect(x)) ⇒ 

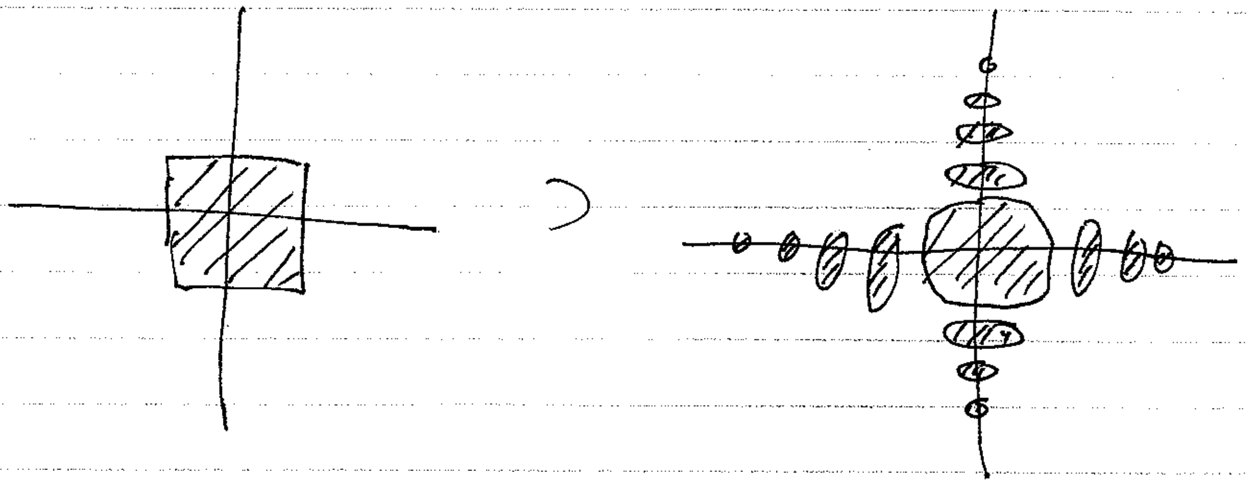
Then applying it again

c = fftshift (b) ⇒ 

to get the original again.

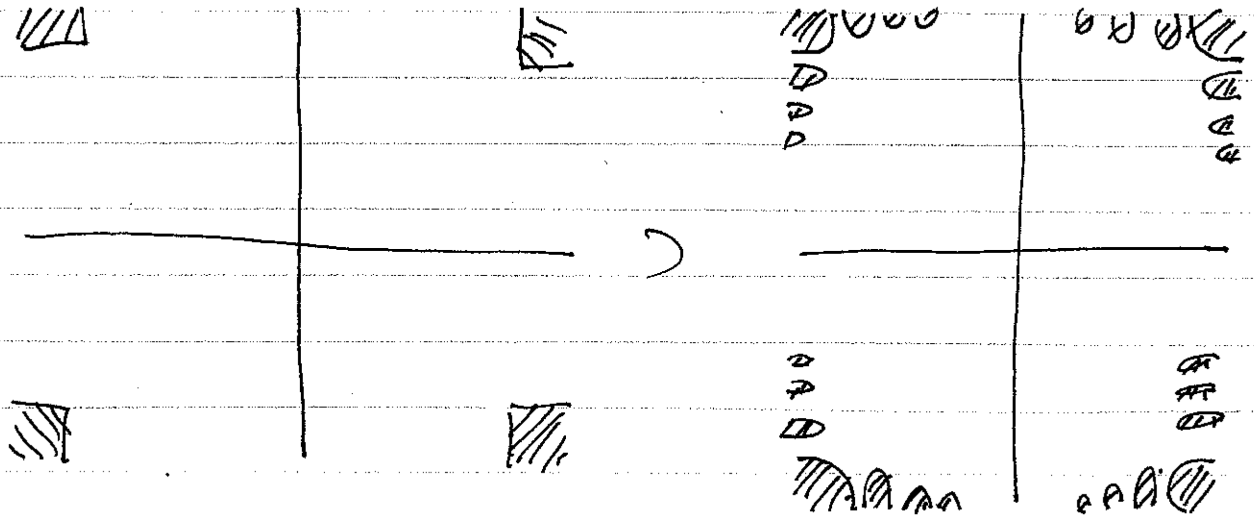
How does this look in 2 dimensions?

Our "natural" viewpoint, with zero in the center:



rect(x, y) sinc(x,y)

In the fft world, with 0 in the corners:

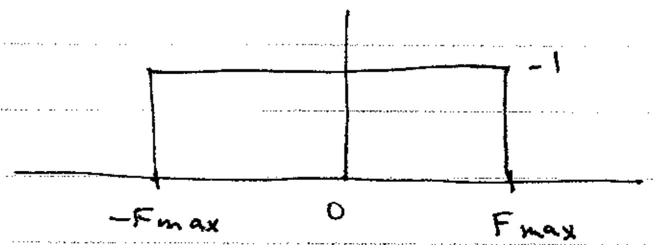


rect(x,y)

Filters using ffts

We can use fft techniques to filter images because we can selectively reduce or amplify particular frequency components. For example, consider a low-pass filter that allows frequencies below a given threshold value to pass, and blocks higher frequencies. In 1-D, we could show the frequency response of such a filter by the following plot:

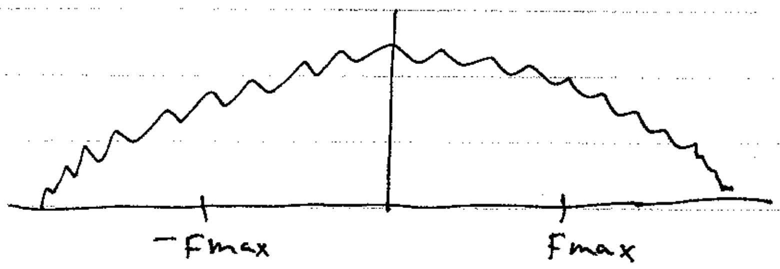
Low-pass filter:



This function is much like our rect function, as it is 1 for all values up to the maximum and zero beyond that. Note that we need to consider the negative frequencies as well as the positive, so that the filter is centered on 0.

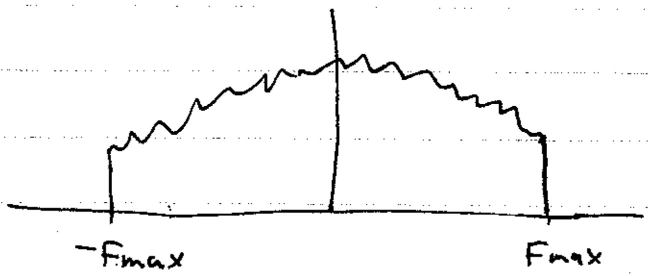
How do we apply this filter? Consider the power spectrum of the signal plotted on the same scale as the filter above:

Signal:



To apply the filter, simply multiply the two functions together, and obtain the filtered spectrum as follows:

Filtered spectrum:



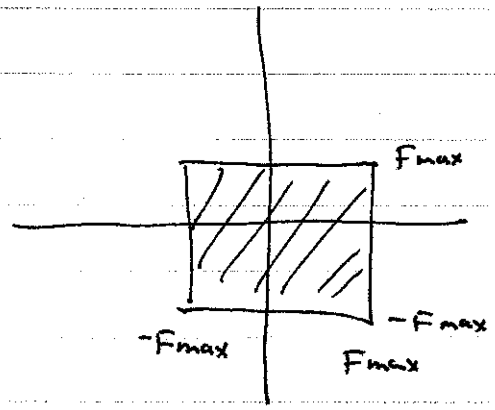
The new spectrum is unchanged within the passband, and is zero outside. But this is only the spectrum of the signal, so we must calculate the inverse transform to get the filtered signal back into the spatial domain.

So a recipe for this type of filtering is

1) Calculate the Fourier transform of the signal
2) Multiply the spectrum by the filter function
3) Inverse transform the result to get the filtered image.

Of course, you must be careful about the fft shifts. If you calculate the filter in our "natural" coordinates, you must fftshift the spectrum before multiplication. Then you must re-fftshift before calculating the inverse transform. It is sometimes easier to simply calculate the filter in fft coordinates directly.

What does the low-pass filter look like in 2-D? Since the 1-D lowpass filter was a 1-D rect function, the 2-D filter is a 2-D rect, as:



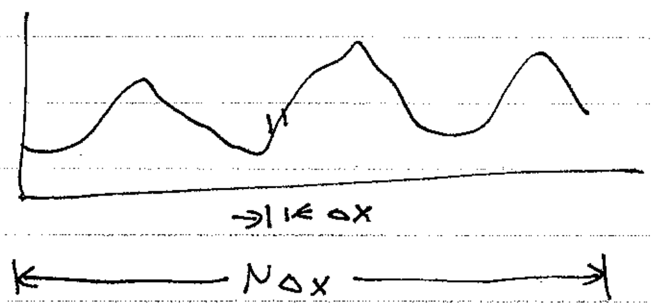
Frequency-domain view
of a lowpass filter

The recipe for applying this filter is the same as in 1-D, again taking care to apply fftshifts when necessary.

Units in the two domains

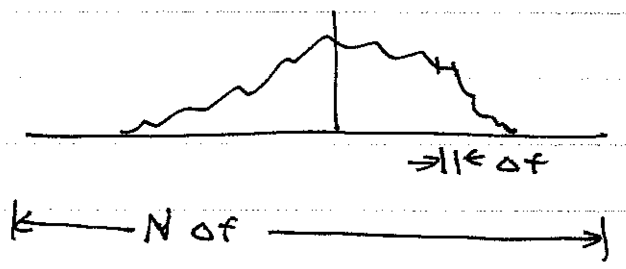
Once we start talking about filtering, we need to be able to label our axes in both the space and frequency domains. Here we need to take care of spacing in both frequency and space.

Consider first a 1-D spatial signal, sampled at a rate fixed at Δx meters. We might have



N samples spaced at Δx meters each, for a total length of $N\Delta x$ meters

If this sequence is transformed, we obtain another sequence, this time of frequencies with total "bandwidth" $N\Delta f$.



We have drawn this spectrum with zero in the middle, our "natural" coordinates.

The Δx and Δf values are the spacings of the measurements in time/space and frequency. How are these related in the Fourier transform? By the following

$$\Delta f = \frac{1}{N\Delta x}$$

and

$$\Delta x = \frac{1}{N\Delta f}$$

Using these, we need only two numbers to interrelate all the values, the length of the transform and the sample spacing, say.

Example: where would we set limits for a 1-D low-pass filter set to pass frequency less than 3 cycles/meter, if we have 128 samples at 0.1 m spacing?

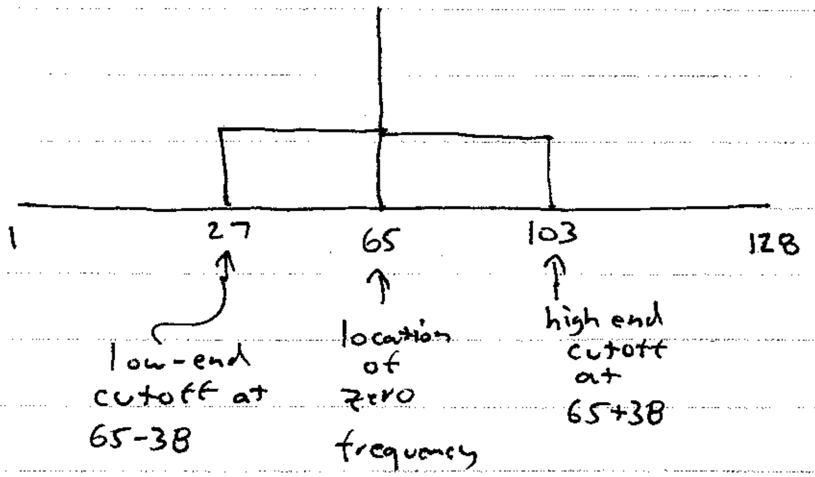
Here, $N = 128$
$\Delta x = 0.1$

so $\Delta f = \frac{1}{12.8} = 0.078$ cycles/m

Which Fourier coefficient does this correspond to for our filter? Since $\Delta f = 0.078$ cycles/m, the coefficient number for 3 cycles/m is

$$\text{coeff \#} = \frac{3}{0.078} = 38.4$$

So the filter function in terms of the coefficients is



This filter would limit the response to ≤ 3 cycles/m.

Complex values

Our last topic related to practical aspects of ffts is the question of complex values for our frequencies and recovered time-domain signals. Recall that the Fourier transform in general produces complex results, even though all of the examples we have looked at were real-valued.

For many operations the actual numbers we arrive at ~~for~~ our solutions are nearly real, with changes due largely to round-off errors in the computer. If our transfer functions are real and symmetric for our filters, we will obtain real transforms.

Practically speaking, if we remember to carry complex operations at each step and at the very end take the absolute value of the result, we will most often get the desired result. Consider for example the following piece of matlab code for a filtering operation.

Assume we have an image in array a, and have calculated a filter in array f. Then we might apply that filter as

```
b = fft2(a);          calculate the transform
c = b.*f;             do the element by element multiply
d = ifft2(c);         get the inverse transform
e = abs(d);           save the absolute values of the
                        complex numbers

disbytebw(e);         finally display the magnitude data
```

Sometimes we need to be more careful, but the above will often work.