2D Fourier Transforms using Java

Learn how the space domain and the wavenumber domain in two-dimensional analysis are analogous to the time domain and the frequency domain in one-dimensional analysis. Learn about some practical examples showing how 2D Fourier transforms and wavenumber spectra can be useful in solving engineering problems involving antenna arrays.

Published: July 12, 2005
By Richard G. Baldwin

Java Programming, Notes # 1490

Preface

This is the first part of a two-part lesson. In this lesson, I will teach you how to perform two-dimensional *(2D)* Fourier transforms using Java. I will

- Explain the conceptual and computational aspects of 2D Fourier transforms
- Explain the relationship between the *space domain* and the *wavenumber* domain
- Provide sufficient background information that you will be able to appreciate the importance of the 2D Fourier transform
- Provide Java software to perform 2D Fourier transforms
- Provide Java software to test and exercise that capability

Two separate programs

I will present and explain two separate programs. One program consists of a single class named **ImgMod30**. The purpose of this class is to satisfy the computational requirements for forward and inverse 2D Fourier transforms. This class also provides a method for rearranging the spectral data into a more useful format for plotting. The second program named **ImgMod31** will be used to test the 2D Fourier transform class, and also to illustrate the use of 2D Fourier transforms for some well known sample surfaces.

A third class named **ImgMod29** will be used to display various 3D surfaces resulting from the application of the 2D Fourier transform. I explained this class in an earlier lesson titled Plotting 3D Surfaces using Java.

Using the class named ImgMod30

The 2D Fourier transform class couldn't be easier to use. To perform a forward transform execute a statement similar to the following:

```
ImgMod30.xform2D(spatialData,realSpect,
                 imagSpect,amplitudeSpect);
```

The first parameter in the above statement is a reference to an array object containing the data to be transformed. The other three parameters refer to array objects that will be populated with the results of the transform.

To perform an inverse transform execute a statement similar to the following:

```
ImgMod30.inverseXform2D(realSpect,imagSpect,
                        recoveredSpatialData);
```

The first two parameters in the above statement refer to array objects containing the complex spectral data to be transformed. The third parameter refers to an array that will be populated with the results of the inverse transform.

To rearrange the spectral data for plotting, execute a statement similar to the following where the parameter refers to an array object containing the spectral data to be rearranged.

```
double[][] shiftedRealSpect =
                 ImgMod30.shiftOrigin(realSpect);
```

Digital signal processing (DSP)

This lesson will cover some technically difficult material in the general area of *Digital Signal Processing*, or *DSP* for short. As usual, the better prepared you are, the more likely you are to understand the material. For example, it would be well for you to already understand the one-dimensional Fourier transform before tackling the 2D Fourier transform. If you don't already have that knowledge, you can learn about one-dimensional Fourier transforms by studying the following lessons:

- 1478 Fun with Java, How and Why Spectral Analysis Works
- 1482 Spectrum Analysis using Java, Sampling Frequency, Folding Frequency, and the FFT Algorithm
- 1483 Spectrum Analysis using Java, Frequency Resolution versus Data Length
- 1484 Spectrum Analysis using Java, Complex Spectrum and Phase Angle
- 1485 Spectrum Analysis using Java, Forward and Inverse Transforms, Filtering in the Frequency Domain
- 1486 Fun with Java, Understanding the Fast Fourier Transform (FFT) Algorithm

You might also enjoy studying my other lessons on DSP as well.

Will use in subsequent lessons

The 2D Fourier transform has many uses. I will use the 2D Fourier transform in several future lessons involving such diverse topics as:

- Processing image pixels in the wavenumber domain
- Advanced steganography *(hiding messages in images)*
- Hiding watermarks and trademarks in images

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

General Discussion

Time domain and frequency domain

In my earlier lessons on DSP, you learned about the relationship between the *time domain* and the *frequency domain.* For example, you learned that time has only one dimension. In the real world, time only goes forward.

> *(In the computer world, we can make it appear that time can also go backwards, but this still constitutes only one dimension.)*

The important point is that time can only go forward or backwards. It cannot go sideways.

A one-dimensional Fourier transform

You learned that you can perform a one-dimensional Fourier transform to transform your data from the time domain into the frequency domain. Similarly, you can perform an inverse one-dimensional Fourier transform to transform your data from the frequency domain back into the time domain.

You learned about several characteristics of Fourier transforms. For example, you learned that a Fourier transform is both linear and reversible. You learned that convolution in the time domain is equivalent to multiplication in the frequency domain, and that convolution in the frequency domain is equivalent to multiplication in the time domain.

You learned that with enough computational power, you can easily transform a given set of data back and forth between these two domains. This makes it possible to use the domain of choice to perform a given signal processing operation, even if the results need to be delivered in the other domain.

Time domain data is purely real

Although it is possible to use the Fourier transform to transform a set of complex data from one domain to the other, real world time domain data is not complex data. Rather, it is purely real. Assuming that the data in one domain is always purely real leads to some simplification of the computational requirements for performing the Fourier transform. In general, the previous DSP lessons assumed real data in the time domain and complex data in the frequency domain.

The space domain

In this lesson, we will extend the concept of the Fourier transform from the time domain into the *space domain.* In making this extension, we will encounter some significant additional complexity. For example, while time is one-dimensional, space is three-dimensional. While you can only move forward and backwards in time, you can move up, down, and sideways in space.

> *(In order to keep the complexity of this lesson in check, we will assume that space is only two-dimensional, allowing movement up, down, and from side to side only. This will serve us well later for such tasks as image processing. Three-dimensional Fourier transforms are beyond the scope of this lesson. I will write a lesson on using Fourier transforms in three-dimensional space later if I have the time.)*

It is also possible and very common to combine time domain signal processing with space domain signal processing. However, that also is beyond the scope of this lesson.

Time and space are analogous

We will consider the space domain to be analogous to the time domain, with the stipulation that the space domain has two dimensions. The unit of measure in the time domain is usually seconds, or some derivative thereof. The unit of measure in space is usually meters, or some derivative thereof.

As with the time domain, we will assume that all space domain surfaces are purely real *(as opposed to being complex).* This will allow us to simplify our computations when performing the 2D Fourier transform to transform our data from the space domain into the wavenumber domain.

(I must point out that from a practical viewpoint this assumption is much more limiting in the space domain than in the time domain. Complex space domain functions are quite common in such areas as antenna array processing.)

Frequency and wavenumber are analogous

We will consider the *wavenumber* domain to be analogous to the frequency domain. The unit of measure in the frequency domain is cycle per second, or some derivative thereof. The unit of measure in the wavenumber domain is cycles per meter or some derivative thereof.

Period and wavelength are analogous

The reciprocal of the typical unit of measure in the frequency domain is seconds per cycle, commonly referred to as the *period.* The reciprocal of the typical unit of measure in the wavenumber domain is meters per cycle, commonly referred to as the *wavelength.*

Some real world examples

With all of this as background, I will begin by discussing some real world engineering problems for which the solution lies in an understanding of the wavenumber domain. I will use these examples to show some of the practical uses of 2D Fourier transforms.

Following that *(in Part 2 of this lesson),* I will present and explain a class that you can copy and use to perform 2D Fourier transforms. Then I will present and explain a program that exercises and tests the 2D Fourier transform class for some common 3D surfaces.

A commercial radio station

Assume that you have just acquired an FCC license to build and operate a new commercial radio station in a small town in west Texas. As is frequently the case in west Texas, your town is situated at the intersection of two highways. One highway runs northeast and southwest. The other highway runs northwest and southwest. The two highways are generally perpendicular to one another. Like many highways in west Texas, each of these highways is straight as an arrow with very few curves.

Where people live

Your town has a small business district at the intersection of the two highways. Beyond that, most of the people who live in your town *(and who will listen to your radio station)* live along the two highways. Thus most of the population lives in the directions of northeast, southwest, northwest, and southeast from the center of town. There are very few people living in the directions of north, south, east, and west. That real estate is mostly populated by cows and cotton fields.

A limit on the transmitting power

Your new FCC license places a limit on the amount of power that you will be allowed to transmit. You would like to use that available power to reach a many human listeners as possible. If you simply construct an omnidirectional transmitting antenna and start broadcasting, approximately half of the power that you transmit will be available mostly to cows and cotton plants. As a result, the amount of power, and hence the *reach* of the power that you transmit to human listeners will be less than you would like for it to be.

A directional transmitting antenna

While the FCC won't allow you to increase the amount of power that you transmit, they will allow you to control the directions in which you choose to transmit that power. You will probably hire an expert in the transmission of radio signals to design a directional transmitting antenna system, which will broadcast most of the available power in the directions where the people live. Ideally, the antenna system will transmit very little of the available power in the directions of the cows and the cotton fields.

The design of the antenna system

The antenna designer will have many tools at her disposal. Whether or not she uses wavenumber terminology, many of the calculations that she performs will depend on wavenumber concepts. She will be concerned about the reciprocal wavenumber *(or wavelength)* of the radio signals that will be broadcast. She will be concerned with the lengths of the active elements in the antenna system, and the distance between active elements if she chooses to use an array of active elements.

Basic concepts

I will leave the radio station scenario at this point and discuss some more basic concepts. I will return to the radio station scenario later. We will need to start with simpler things and work our way up to the radio station scenario.

Much of this discussion will be couched in terms of receiving signals rather than transmitting signals. *(For most people, receiving is easier to understand than transmitting.)* However, most of the conclusions that we reach regarding antenna systems used for receiving signals are also applicable to antenna systems used for transmitting signals.

A one-dimensional space

Just to get us started down the right path, we will temporarily constrain space to have only one dimension. We will discuss the propagation of waves along a taut wire, as well as the measurement of the waves propagating along that wire.

Assume that a wire is fastened at both ends, is fairly taut, and is suspended between two walls so that it is free to move up and down only. Assume that we attach two sensors to the wire, one meter apart, and that each of these sensors is capable of generating an electrical signal that is proportional to the vertical displacement of the wire at the point where the sensor is attached. If the wire goes up, the sensor generates a positive signal. If the wire goes down, the sensor generates a negative signal.

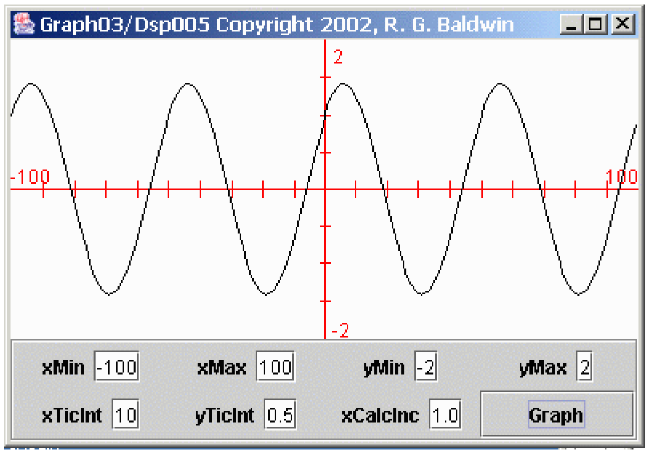
Standing waves

There are two ways that we can approach this analysis. If the wire is very long, we can think in terms of a deformation pulse that propagates along the wire passing by our sensors once and only once. This would fall into the category of transient analysis.

On the other hand, if the wire is shorter, we can think in terms of vibrating one end of the wire in such a way that a standing wave will develop on the wire. This is probably the easier of the two approaches to understand because you may have created standing waves on a rope as a child.

What does a standing wave look like?

Once a standing wave is set up on the wire, it will take on an appearance very similar to the sine wave shown in Figure 1.



xMin -100 xMax 100 yMin -2 yMax 2
xTicInt 10 yTicInt 0.5 xCalcInc 1.0

Figure 1 A stand wave on a wire

Figure 1 represents a snapshot taken at a single instant in time. Obviously the wire doesn't remain in the position shown in Figure 1 for very long. Rather a single point on the wire will move up and down with time with the overall appearance being as shown in Figure 1. The distance between any two positive peaks is what we would refer to as the *wavelength* of the standing wave.

Add the sensor output signals

Now consider what would happen if we were to electronically add the electrical outputs produced by the two sensors. The result would depend on the distance between the sensors relative to the wavelength of the standing wave. For example, if the two sensors were exactly one wavelength apart, the two sensors would move up and down in unison, and the sum of the two signals would be double the signal level produced by either sensor alone. The result would be the same as if the two sensors were located at the same place on the wire.

A one-half wavelength spacing

On the other hand, if the two sensors were exactly one-half wavelength apart, one would go be going up when the other is going down, and the electrical output from one would cancel the electrical output from the other. In space processing terminology, this would be referred to as a *null point.*

Change the wavelength of the standing wave

Now consider what would happen if you were to leave the two sensors in the same locations as before and do something to the wire to change the wavelength of the standing wave. The output from the sum of the two sensors would range from zero to maximum as the wavelength relative to separation varies from one-half wavelength to one full wavelength. For a one-half wavelength separation, the output would be zero. For a one-wavelength separation, the output would be at its maximum.

A two element array

We could refer to our two sensors as a two element array, and we could refer to the output produced by the sum of the two sensors as the response of the array. We could plot the response versus wavelength. However, in the same sense that it is more common to plot the response of electrical filters versus frequency *(instead of period)*, it is probably most common to plot the response of arrays versus wavenumber *(instead of wavelength).* Therefore, in this lesson, we will plot the response of the array versus wavenumber.

How can we compute the array response?

This is where Fourier transforms come into play. We could compute the response of our one-dimensional array for waves propagating along the length of the wire by treating the elements of the array as samples in space and performing a one-dimensional Fourier transform on the elements of the array.

A sampled space series

In this case, we would consider the array elements to constitute samples taken in space in the same way that we consider a sampled time series to constitute samples taken in time. In other words, the array elements constitute a sampled space series. The Fourier transform of a sampled time series is the frequency spectrum of the time series. The Fourier transform of a sampled space series is the wavenumber spectrum of the space series.

Three example wavenumber spectra

To set the stage for what we will be seeing later, Figure 2 depicts three different two element arrays with different spacing in the images across the top of the figure. The wavenumber response for each of the three arrays is shown immediately below the image of the array.

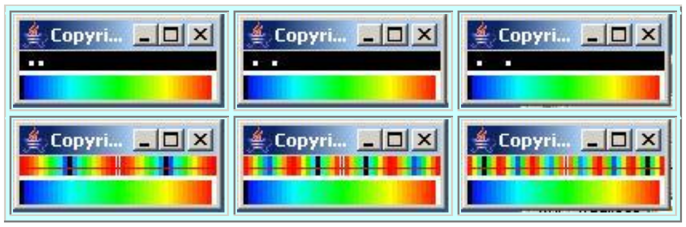


Figure 2

Plots of 3D surfaces

The images shown in Figure 2 were produced using the class named **ImgMod29**, which I explained earlier in the lesson titled Plotting 3D Surfaces using Java. You can refer back to that lesson for a detailed explanation of the display format. Briefly, however, each of the six individual images in Figure 2 is a plot of a 3D surface, with the elevation of the surface at any particular point being indicated by the color at that point based on the colors on the calibration scale below each plot.

(The calibration scale is the strip that changes color in a smooth gradient from blue through green and yellow to red with black at the left end and white at the right end.)

Black, white, and the colors in between

The lowest elevation in the plot is colored black. Hence the backgrounds are black in the top three plots. The highest elevation is colored white. Hence, the array elements are white in the top three plots.

Between black and white, the elevation is given by the color scale below the plot with the lowest elevation on the left of the scale and the highest elevation on the right. Thus, a green elevation is about half way between the lowest and highest elevations. Blue elevations are near the low end. Red elevations are near the high end. Aqua and yellow elevations fall in between as shown by the calibration scale.

The maximum array output

All three wavenumber plots have a maximum response at the center, which is the zero wavenumber origin. In effect, this corresponds to infinite wavelength. If the wavelength is infinite, it doesn't matter what the separation between the elements is, they will all move up and down in unison and their electrical outputs will add constructively to produce a maximum output.

Response of the leftmost array

Now consider the leftmost pair of images where the two array elements are relatively close together. The wavenumber response of this array has a pair of null points about midway between the center and either end, as indicated by the blue and black colors. This is the wavenumber for which the element spacing is an exact multiple of one-half of the wavelength, causing the elements to move in equal but opposite directions in response to the wave motion. Thus, the output from one element cancels the output from the other element producing zero voltage in the sum.

This same pair of images shows high responses at either end as indicated by the red areas. This is the wavenumber for which the element spacing is an exact multiple of one wavelength.

The Nyquist folding wavenumber

If this were a frequency spectrum analysis, we would say that the end points are at the Nyquist folding frequency, which is one-half of the sampling frequency. Thus, we can say that in the wavenumber domain, the end points on our plots are at the Nyquist folding wavenumber, which is one-half the sampling wavenumber.

The wavenumber spectrum is periodic. The section of the wavenumber spectrum that we are viewing represents one complete period of a periodic wavenumber spectrum ranging from minus the folding wavenumber, through zero wavenumber to plus the folding wavenumber.

> *(If you are already familiar with this sort of thing, you may have figured out that the separation between our elements in this example is twice the sampling distance that determines the location of the folding wavenumber.)*

Estimating the array response from the colors

The leftmost array response shows a white area at the center.

> *(The white area is split by the vertical component of red axes drawn on the plot. The color of the axes has nothing to do with elevations on the surface. I simply decided to draw them in red to make them stand out.)*

This array response also shows the two black areas mentioned earlier. You can use the orange, yellow, green, and aqua locations on the calibration scale to estimate the response of the array to different wavenumber values between maximum and minimum.

Additional separation between array elements

Now consider the two images in the center of Figure 2. The elements for this array are separated more than the elements in the leftmost pair of images. Again, the wavenumber response for this array has a maximum value at the origin, which is at the center of the wavenumber response in the lower image. In addition, this array response has a high *(red)* response at five different wavenumber zones, whereas the array on the left had only three red zones. Similarly, this array has a low response *(black and blue)* at four different wavenumber zones whereas the array on the left has a low response at only two wavenumber zones.

Thus, changing the separation between the elements has a significant impact on the wavenumber response of the array.

Separate the elements even more

Finally, consider the pair of images on the right where the elements are even further apart. This array response has even more peaks and valleys than the other two.

A wavenumber filter

The sum of the outputs from an array of sensor elements represents a form of wavenumber filter *(much as the correct combination of resistors, capacitors, and inductors represents a frequency filter).* If we need to pass signals having one wavenumber and to suppress signals having a different wavenumber, we may be able to adjust the separation between the elements so as to put a peak on the desirable wavenumber and to put a null point on the undesirable wavenumbers.

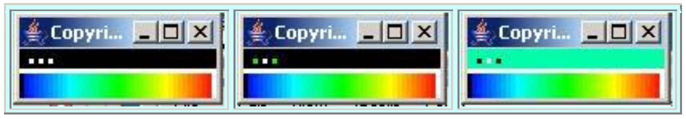
A two element array is fairly limiting

Of course, with only two elements, we don't have very many degrees of freedom to work with. We could exercise more control over our wavenumber filter if we had more elements. We could do even better if we had the ability to give each element a different weight *(including a negative weight)* when the signals from all the elements are added together. Finally, we could do even better still if we had the ability to insert a programmable time delay *(phase shift)* into the output from each of the elements before adding them together.

> *(The use of programmable time delays falls in the category of a space series that is complex rather than being purely real. Thus, that topic is beyond the scope of this lesson.)*

Let's apply some weights

Now let's modify our scenario and see what we can learn in the process. We are going to increase the size of the array from two to three elements. We are also going to assume that we can apply amplification, sign reversal, or both to the element output signals before adding those signals together. The results are shown in Figure 3. We will compare the results in Figure 3 with the results discussed earlier in Figure 2, so this may be a good time for you to open another copy of this lesson in a separate browser window if you haven't already done so.



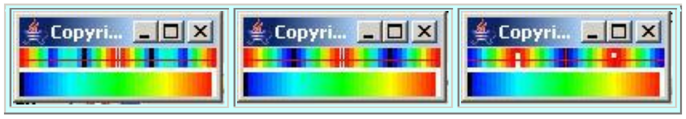


Figure3

The leftmost pair of images

The leftmost pair of images in Figure 3 is similar to the leftmost pair of images in Figure 2, except that we added a third element in Figure 3.

All three elements in Figure 3 are weighted equally prior to summation. The separation between the left and center elements is the same as in Figure 2. The separation between the center and right elements is the same as the separation between the left and center elements.

The wavenumber response

The most noticeable thing about the wavenumber response for this three element array is that the central peak is narrower than the central peak for the two element array at the left of Figure 2. In addition, the trough between the central peak and the peaks at the ends is deeper, broader, and probably flatter *(although the degree of flatness is hard to determine from this plotting format).*

Could continue lengthening the array

Although I won't demonstrate it, I can tell you that if I were to continue adding elements in this manner to increase the length of the array, the central peak and the peaks at the folding wavenumbers would continue getting narrower, and the trough between the peaks would continue getting deeper and probably flatter.

A more selective wavenumber filter

In other words, when viewed as a wavenumber filter, a long array is a more selective wavenumber filter than a short array. By properly designing an array to act as a wavenumber filter, it is possible to cause that filter to be very selective.

When we use a properly designed array to produce a directional antenna, it is possible to produce a highly directional antenna *(and avoid wasting our valuable radio frequency (RF) energy by sending it to cows and cotton plants).* I will have more to say about cows and cotton plants later.

A weighted three element array

Continuing with our three element array scenario, let's take a look at the center pair of images in Figure 3 and compare them with the leftmost images in Figure 3.

For this case, the array still contains three elements with the same spacing as before. However, the electrical output from the center element is amplified to make it twice as strong as the outputs from the other two elements before the three electrical signals are added together.

It is a little hard to tell what this does to the central peak in the wavenumber response, but it definitely changes the shape of the response in the trough between the peaks. Whether or not this would be a beneficial change would depend on the problem being addressed.

A three element array with negative weighting

Finally, take a look at the rightmost pair of images in Figure 3. Once again, the array contains three elements and the center element is weighted twice as heavily as the other two. In addition, the sign of the electrical signals from the two outer elements is inverted before the three are added together.

The wavenumber response

This has a major impact on the wavenumber response of the three element array.

There is no longer a peak at a wavenumber value of zero. Rather, there is now a null point at zero wavenumber as indicated by the black and blue colors at the center of the plot. There is now a peak on each side of zero *(as indicated by the white and red colors)*, half way between zero and the folding wavenumber.

A reassessment of the leftmost and center images

Remember that I told you earlier that the wavenumber response that we are viewing represents one complete period of a periodic wavenumber response extending from minus the folding wavenumber to plus the folding wavenumber.

> *(With these three element arrays, the separation between the elements is twice the assumed sampling grid in space. In other words, you could view each of these arrays as a five element array with an element having a weight of zero in between the center element and each of the outer elements.)*

Two full peaks but at different locations

If you consider the peaks at the ends of the wavenumber response for the leftmost and center images in Figure 3 to each represent only half a peak *(with the other half being off the scale to the left and the right)*, all three scenarios have two complete peaks in their wavenumber responses.

> *(You could think in terms of printing the wavenumber response on a piece of paper, cutting it out, and taping the two ends together to form a continuous ring. As you made a complete traversal of the ring, you would encounter two peaks.)*

However, the locations of the two peaks for the rightmost array are at completely different wavenumber values than are the peaks for the other two arrays. The two peaks exhibited by the rightmost array are in the locations of the two nulls for the center array. Similarly, the null points for the rightmost array are in the same locations as the two peaks for the center array.

What can we learn from these scenarios?

We learn that we can have a significant impact on the wavenumber response of an array by increasing the number of elements in the array. We can also have a significant impact on the wavenumber response by applying weights, *(including sign changes)*, to the electrical signals produced by the array elements before adding them together.

Extending into two dimensions

Now let's complicate things a bit by extending our array analysis into two dimensions. Up to this point, we have assumed that our sensors were attached to a wire that was free to move up and down only. As such, waves impinging on the array were constrained to approach the array from one end or the other. In this case the wavenumber was completely determined by the wavelength of the wave.

> *(For our purposes, the wavelength is given by the ratio of propagation speed in meters per second to frequency in cycles per second. Canceling out the units leaves us with wavelength in meters per cycle.)*

Move the array to a table top

Let's move our array of sensors from the wire to a large sheet of metal on the top of a table. For the time being, we will still place the elements in a line with uniform spacing. However, we will now assume that a wave can impinge on the array from any direction along the surface of the sheet of metal.

> *(For simplicity, we will assume that there is some sort of insulation between the sheet of metal and the table top to prevent waves from impinging on the array from below.)*

What does a wave look like in this scenario?

Imagine a piece of corrugated sheet metal or fiber glass. *(Material like this is sometimes used to build a roof on a patio.)* When you look at it from one end, it looks something like the sine wave in Figure 1. However, if you keep it at eye level and slowly turn it, the distance between the peaks will appear to become shorter and shorter until finally you don't see any peaks at all. What you see at that point is something that appears to have the same thickness from one end to the other. This is the view that one of our sensors sees as the wavefront of an impinging wave.

The angle of attack is important

We now have a much more complex situation. If the waves continue to impinge on the array from one end or the other, the situation will be exactly the same as when the sensors were on the wire. However, the *apparent* wavenumber or wavelength of a wave as seen by the array will depend on the angle of attack.

(There is now a difference between the actual wavelength or wavenumber and the apparent wavelength or wavenumber as seen by the array.)

Infinite wavelength

For example, if the wave impinges on the array from a broadside direction, all of the sensors will move up and down in unison regardless of their separation and regardless of the actual wavelength of the wave. For this case, the wave will *appear* to the array to have infinite wavelength or zero wavenumber.

(A linear array has no ability to filter on the basis of wavenumber for waves that impinge on the array from the broadside direction. All waves from that direction appear to have zero wavenumber. This will lead us later to consider the use of a two-dimensional array.)

The 2D wavenumber response of a linear array

Figure 4 shows the two-dimensional wavenumber response for a five element linear array with equal weighting for all of the elements. The array is shown at the top. The wavenumber response of the array is shown at the bottom.

(In this case, I placed all five elements on adjacent points on the space sampling grid with no spaces in between. This places them so close together that you can't visually separate them in the image.)

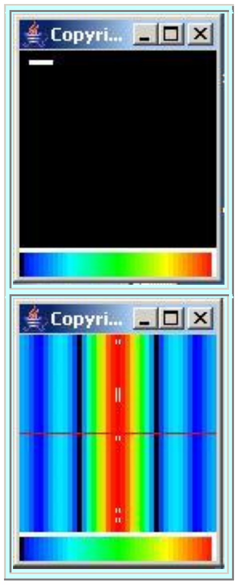


Figure 4

The response for a constant wavenumber

If you were to draw a circle centered on the crosshairs *(axes)* in the center of the wavenumber response, the points on that circle would represent a fixed wavenumber for a wave arriving from any direction. The value of the response at any particular point on the circle would indicate the response of the array to a wave having that wavenumber from that direction.

Response versus direction

If the diameter of the circle is larger than the width of the red vertical band, and if you were to plot that response versus direction, you would see that the response is maximum for the two directions that are broadside to the array and the response tends to drop off as the direction approaches the end fire direction of the array.

Symmetry

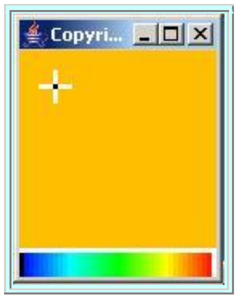
You would also notice quite a lot of symmetry. For example, the maximum response occurs in two directions that are 180 degrees apart. In fact, if you pick any direction and a given wavenumber, the response is the same for that direction and for the direction that is 180 degrees around from that direction.

Not good for the radio transmitter

This wouldn't be a very good design for your radio station. If one of the broadside directions of the array faces northeast and the other faces southwest, then the people who live in the northwest and southeast directions wouldn't receive a very good signal from your transmitter. You need a design that maximizes the power in the four directions where the people live, and that minimizes the power in the other directions. To accomplish that, we will need a two-dimensional array in place of our one-dimensional linear array.

A two-dimensional array

We will achieve the desired array response by using an array having thirteen elements in the form of a cross with very specific weighting applied to each element prior to summation. The array and the wavenumber response of the array are shown in Figure 5.



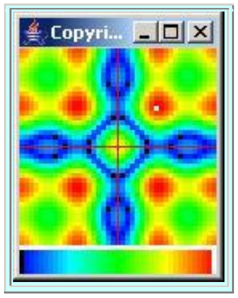


Figure 5

Computing the wavenumber response

The wavenumber response of the array *(shown at the bottom in Figure 5)* was produced by performing a 2D Fourier transform on the weighted array *(shown at the top in Figure 5).*

The center element in the array was weighted by -4.5 before summation. *(The signal from this element was amplified by a factor of 4.5 and the sign of the signal was inverted prior to summation.)*

The twelve remaining elements were weighted by a factor of 1.0 before summation.

A constant wave number

Regardless of direction, the wavelength or wavenumber of the RF energy transmitted by a commercial radio station is the same and it doesn't change over time, *(unless the frequency on which the station broadcasts changes)*.

Once again, we can determine the wavenumber response of the array for a given wavenumber from any direction by drawing a circle on the response plot, centered at the origin, with the radius of the circle equal to the specific wavenumber of interest.

Assume a wavenumber

Assume that the wavenumber of interest in our case is exactly equal to the distance from the origin to the white spot in the upper right quadrant of the wavenumber response plot. This white spot represents the maximum response of the array.

> *(If our computation had perfect accuracy, there would be a white spot at the same location in all four quadrants.)*

Draw a circle centered on the origin having a radius that causes the circle to go through the white spot.

Now determine the response versus direction

The color corresponding to the response at any point on the circle represents the response of the array to that wavenumber for waves arriving from that direction *(or in the case of a transmitter, for waves being transmitted in that direction).*

The circle passes through yellow, red, and white *(indicating a high response)* in the general directions of northeast, southeast, southwest, and northwest. This means that a strong RF signal will be transmitted to the people living along the highways in those four directions.

The circle passes through green, aqua, and blue *(indicating a low response)* in the general directions of north, south, east, and west. This means that very little of the precious RF energy will be transmitted to the cotton plants and the cows that live in those directions.

A possible solution to the problem

Thus, an array of active transmitter elements arranged and weighted as shown at the top of Figure 5 might be a reasonable design solution for your radio station. *(However, I suspect that an experienced RF engineer would have a much more sophisticated solution.)*

In any event, you have now seen one possible practical example of the use of a 2D Fourier transform.

Arrays are used in various applications

Although this example was admittedly somewhat contrived, it is not far fetched. Arrays similar to those that I have been discussing are widely used in the technology area of spatial signal processing.

Radio astronomy

Perhaps the application that is most familiar to the general public *(due to widespread publicity and a very popular movie)* is the Paul Allen radio telescope used in the Search for Extraterrestrial Intelligence *(SETI)*.

In the past, much of this work has been done using a very large dish antenna known as the Arecibo Radio Telescope in Puerto Rico. Efforts are now underway involving an alternative approach that uses a large array of small dishes instead of one large dish.

By properly processing and then summing the outputs produced by the dishes in the array, the users will be able to steer the telescope and possibly to also eliminate strong sources of interference.

Seismology

Arrays of seismometers are used by U.S. government agencies to monitor for seismic signals produced by earthquakes in locations nearly halfway around the earth.

By applying complex, frequency dependent weighting factors to the seismometer outputs before summing them, the arrays can be tuned to provide a complex response in wavenumber space. For example, the arrays can be processed to form response beams looking in different directions with a beam width that is relatively constant across a wide band of interesting frequencies. In addition, null points in the wavenumber response can be created to suppress seismic noise that originates from specific points on the earth such as mines, rock quarries, and cities.

The design and analysis of such array systems use 2D *(and sometimes 3D)* Fourier transforms. Because the weights that are applied are produced by complex frequency filters, the transform programs that are used must treat both the space domain data and the wavenumber data as complex *(instead of being purely real as in the examples in this lesson).*

Sonar

Probably ninety percent of all sonar systems currently installed on surface ships and submarines use arrays for steering and processing both active and passive sonar. In almost all cases, these are 3D arrays. Some of the arrays contain multiple sensors on the surface of a portion of a sphere. Some contain multiple sensors located along slats that are mounted on a frame much like the staves on a barrel. Some are located on the sides of the vessel. There are probably numerous other geometries in use as well.

A Fourier transform program used with these arrays would normally have to be a 3D Fourier transform program capable of transforming from complex space functions to complex wavenumber functions.

Radar

One of the reasons that sonar is typically processed using arrays has to do with the wavelength of the signals and the operating environment. It is usually not practical to physically move a sonar sensor large enough to do the job in order to cause it to look in different directions. Thus arrays of small sensors are used with the ability to steer beams electronically in order to look in different directions.

Because of the shorter wavelengths involved, typical radar sensors are usually small enough that they can be physically turned and tilted. Thus, it is not unusual to see radar sensors turning around and tilting up and down. Although I'm not personally aware of any applications that use arrays of radar sensors, my suspicion is that there probably are some being used in fixed air surveillance operations.

Petroleum exploration

A large percentage of petroleum exploration involves the insertion of a powerful surge of acoustic energy into the ground *(or into the ocean)* and listening for and recording the echo signals returned by the various layers of the earth. By moving across the earth and repeating this process, a profile of the earth's layering can be produced. An experienced exploration geophysicist can examine the profiles and reach conclusions as to the likelihood that a particular stratum contains petroleum.

Exploration geophysicists have been using arrays of sensors for this purpose for at least the past 45 years according to my personal knowledge, and probably for many years before that.

Image processing in the wavenumber domain

While the examples described above are interesting, they are beyond the scope of anything that I can demonstrate online. However, there are several interesting applications using 2D Fourier transforms that I can demonstrate online. One of those applications is image processing.

Future lessons will show how to use 2D Fourier transforms for such purposes as softening images, sharpening images, doing edge detection on images, etc. For this application, it is satisfactory to use a 2D Fourier transform program that assumes that the space domain data is purely real. Therefore, the program that I will present and explain in Part 2 of this lesson will make that assumption.

Image processing in the space domain

The 2D Fourier transform will be used in future lessons to help explain how and why 2D image convolution behaves the way it does. A preview of that material is shown in Figure 6.

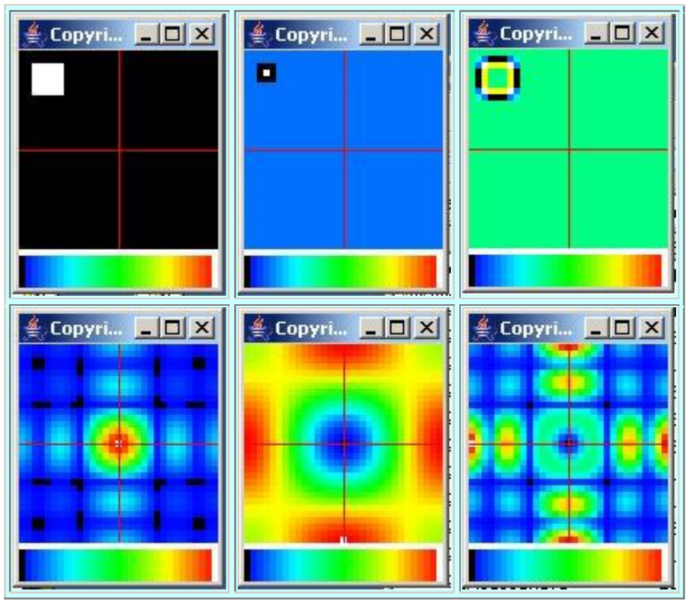


Figure 6

Convolution versus multiplication

Recall that convolution in the space domain is equivalent to multiplication in the wavenumber domain, and vice versa.

A simple space function and its wavenumber spectrum

The top left image in Figure 6 shows a simple 3D surface in space consisting of a raised square. The wavenumber spectrum of that surface is shown in the lower left image in Figure 6. Note that the spectrum has a peak at a wavenumber value of zero with low values at the higher wave numbers near the edges. The peak in the center is relatively narrow with respect to the folding wave number at the edges.

A 2D convolution operator and its spectral response

The image in the upper center of Figure 6 shows a typical 2D convolution operator consisting of a value of +8 in the center surrounded by eight coefficients each having a value of -1.

The wavenumber spectral response of that convolution operator is shown in the lower center. Note that it has peaks at the folding wavenumbers on all four sides with a low value in the center.

The deepest part of the trough in the center is relatively narrow with respect to the folding wave numbers at the edges. However, it is somewhat broader than the peak in the spectrum at the lower left.

The result of convolution

The image in the upper right shows the result of convolving the space domain surface in the upper left with the convolution operator in the upper center. This output space domain surface has a green square area in the center that is at the same level as the green background. In this case, green represents an elevation of 0, which is about midway between the lowest elevation *(black)* and the highest elevation *(white).*

Positive and negative fences

Surrounding the green square is a yellow and white fence representing very high elevations. Surrounding that fence is a black and blue fence, representing very low elevations consisting of large negative values.

Thus, as you move from the outside to the inside of the square in the output surface, the elevation goes from a background level of zero, to a large negative value, followed immediately by a large positive value, followed by zero.

Edge detection

As you can see, this is one form of edge detection. The edges of the square in the input surface have been emphasized and the flat portion of the input surface has been deemphasized in the convolution output.

Wavenumber spectrum of the convolution output

The wavenumber spectrum of the output from the convolution operation is shown in the lower right. The spectrum indicates that this surface is made up mostly of wavenumber components having mid range to high values.

If you are familiar with digital signal processing, you will know that in order for a space *(or time)* function to contain very rapid changes in value *(such as the elevation changes at the fences described above)* the function must contain significant high wavenumber *(or frequency)* components. That appears to be the case here.

Although this spectrum was produced by convolution in the space domain followed by a 2D Fourier transform on the convolution output, you should be able to see that the shape of the spectrum on the bottom right approximates the product of the spectrum of the original surface on the bottom left and the spectral response of the convolution operator in the bottom center.

Thus, the same results could have been produced using multiplication in the wavenumber domain followed by an inverse Fourier transform to produce the space domain result. Convolution in the space domain is equivalent to multiplication in the wavenumber domain and vice versa.

Hidden watermarks and trademarks

Another interesting application that I can demonstrate online is using 2D Fourier transforms to hide secret trademarks and watermarks in images. The purpose of a hidden trademark or watermark is for the owner of the image to be able to demonstrate that the image may have been used inappropriately by someone else. Once again, this application can be satisfied by treating the space domain data as purely real. I plan to demonstrate how this is done in a future lesson.

Run the Programs

Although I did not discuss any Java programs in Part 1 of this lesson, I will provide and explain two different programs in Part 2 of this lesson. For those of you who may want to get a jump on that lesson, I have included those two programs in Listing 1 and Listing 2 below. I encourage you to copy, compile, and run those programs. Modify the programs and experiment with them in order to learn as much as you can about 2D Fourier transforms.

Create some different test cases and work with them until you understand why they produce the results that they do.

Summary

I began by explaining how the space domain and the wavenumber domain in two-dimensional analysis are analogous to the time domain and the frequency domain in one-dimensional analysis.

Then I introduced you to some practical examples showing how 2D Fourier transforms and wavenumber spectra can be useful in solving engineering problems involving antenna arrays.

What's Next?

In Part 2 of this lesson, I will provide and explain a Java class that can be used to perform forward and inverse 2D Fourier transforms, and can also be used to shift the wavenumber origin from the upper left to the center for a more pleasing plot of the wavenumber spectrum.

In addition, I will provide and explain a program that is used to:

- Test the forward and inverse 2D Fourier transforms to confirm that the code is correct and that the transformations behave as they should
- Produce wavenumber spectra for simple surfaces to help the student gain a feel for the relationship that exists between the space domain and the wavenumber domain

Complete Program Listings

Preview listings of the classes that I will present and explain in Part 2 of this lesson are provided in Listing 1 and Listing 2 below.

```
/*File ImgMod30.java
Copyright 2005, R.G.Baldwin

The purpose of this program is to provide 2D
Fourier Transform capability to be used for image
processing and other purposes.  The class
provides three static methods:

xform2D: Performs a forward 2D Fourier transform
 on a surface described by a 2D array of double
 values in the space domain to produce a spectrum
 in the wavenumber domain.  The method returns
 the real part, the imaginary part, and the
 amplitude spectrum, each in its own 2D array of
 double values.

inverseXform2D:  Performs an inverse 2D Fourier
 transform from the wavenumber domain into the
 space domain using the real and imaginary parts
 of the wavenumber spectrum as input.  Returns
 the surface in the space domain in a 2D array of
 double values.

shiftOrigin:  The wavenumber spectrum produced
 by xform2D has its origin in the upper left
 corner with the Nyquist folding wave numbers
 near the center.  This is not a very suitable
 format for visual analysis.  This method
 rearranges the data to place the origin at the
 center with the Nyquist folding wave numbers
 along the edges.

Tested using J2SE 5.0 and WinXP
************************************************/
import static java.lang.Math.*;

class ImgMod30{

  //This method computes a forward 2D Fourier
  // transform from the space domain into the
  // wavenumber domain.  The number of points
  // produced for the wavenumber domain matches
  // the number of points received for the space
  // domain in both dimensions.  Note that the
  // input data must be purely real.  In other
  // words, the program assumes that there are
  // no imaginary values in the space domain.
  // Therefore, it is not a general-purpose 2D
  // complex-to-complex transform.
  static void xform2D(double[][] inputData,
                      double[][] realOut,
                      double[][] imagOut,
                      double[][] amplitudeOut){

    int height = inputData.length;
    int width = inputData[0].length;

    System.out.println("height = " + height);
    System.out.println("width = " + width);

    //Two outer loops iterate on output data.
    for(int yWave = 0;yWave < height;yWave++){
      for(int xWave = 0;xWave < width;xWave++){
        //Two inner loops iterate on input data.
        for(int ySpace = 0;ySpace < height;
                                       ySpace++){
          for(int xSpace = 0;xSpace < width;
                                       xSpace++){
//Compute real, imag, and ampltude. Note that it
// was necessary to sacrifice indentation to
// force  these very long equations to be
// compatible with this narrow publication format
// and still be somewhat readable.
realOut[yWave][xWave] +=
 (inputData[ySpace][xSpace]*cos(2*PI*((1.0*
 xWave*xSpace/width)+(1.0*yWave*ySpace/height))))
 /sqrt(width*height);

imagOut[yWave][xWave ] -=
 (inputData[ySpace][xSpace]*sin(2*PI*((1.0*xWave*
  xSpace/width) + (1.0*yWave*ySpace/height))))
  /sqrt(width*height);

amplitudeOut[yWave][xWave] =
 sqrt(
  realOut[yWave][xWave] * realOut[yWave][xWave] +
  imagOut[yWave][xWave] * imagOut[yWave][xWave]);
          }//end xSpace loop
        }//end ySpace loop
      }//end xWave loop
    }//end yWave loop
  }//end xform2D method
  //-------------------------------------------//

  //This method computes an inverse 2D Fourier
  // transform from the wavenumber domain into
  // the space domain.  The number of points
  // produced for the space domain matches
  // the number of points received for the wave-
  // number domain in both dimensions.  Note that
  // this method assumes that the inverse
  // transform will produce purely real values in
  // the space domain.  Therefore, in the
  // interest of computational efficiency, it
  // does not compute the imaginary output
  // values.  Therefore, it is not a general
  // purpose 2D complex-to-complex transform. For
  // correct results, the input complex data must
  // match that obtained by performing a forward
  // transform on purely real data in the space
  // domain.

  static void inverseXform2D(double[][] real,
                             double[][] imag,
                             double[][] dataOut){

    int height = real.length;
    int width = real[0].length;

    System.out.println("height = " + height);
    System.out.println("width = " + width);

    //Two outer loops iterate on output data.
    for(int ySpace = 0;ySpace < height;ySpace++){
      for(int xSpace = 0;xSpace < width;
                                        xSpace++){
        //Two inner loops iterate on input data.
        for(int yWave = 0;yWave < height;
                                         yWave++){

          for(int xWave = 0;xWave < width;
                                         xWave++){
//Compute real output data. Note that it was
// necessary to sacrifice indentation to force
// this very long equation to be compatible with
// this narrow publication format and still be
// somewhat readable.
dataOut[ySpace][xSpace] +=
 (real[yWave][xWave]*cos(2*PI*((1.0 * xSpace*
 xWave/width) + (1.0*ySpace*yWave/height))) -
 imag[yWave][xWave]*sin(2*PI*((1.0 * xSpace*
 xWave/width) + (1.0*ySpace*yWave/height))))
 /sqrt(width*height);
          }//end xWave loop
        }//end yWave loop
      }//end xSpace loop
    }//end ySpace loop
  }//end inverseXform2D method
  //-------------------------------------------//

  //Method to shift the wavenumber origin and
  // place it at the center for a more visually
  // pleasing display.  Must be applied
  // separately to the real part, the imaginary
  // part, and the amplitude spectrum for a wave-
  // number spectrum.
  static double[][] shiftOrigin(double[][] data){
    int numberOfRows = data.length;
    int numberOfCols = data[0].length;
    int newRows;
    int newCols;

    double[][] output =
          new double[numberOfRows][numberOfCols];

    //Must treat the data differently when the
    // dimension is odd than when it is even.

    if(numberOfRows%2 != 0){//odd
      newRows = numberOfRows +
                            (numberOfRows + 1)/2;
    }else{//even
      newRows = numberOfRows + numberOfRows/2;
    }//end else

    if(numberOfCols%2 != 0){//odd
      newCols = numberOfCols +
                            (numberOfCols + 1)/2;
    }else{//even
      newCols = numberOfCols + numberOfCols/2;
    }//end else

    //Create a temporary working array.
    double[][] temp =
                    new double[newRows][newCols];

    //Copy input data into the working array.
    for(int row = 0;row < numberOfRows;row++){
      for(int col = 0;col < numberOfCols;col++){
        temp[row][col] = data[row][col];
      }//col loop
    }//row loop

    //Do the horizontal shift first
    if(numberOfCols%2 != 0){//shift for odd
```

```
      //Slide leftmost (numberOfCols+1)/2 columns
      // to the right by numberOfCols columns
      for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
                 col < (numberOfCols+1)/2;col++){
          temp[row][col + numberOfCols] =
                                   temp[row][col];
        }//col loop
      }//row loop

      //Now slide everything back to the left by
      // (numberOfCols+1)/2 columns
      for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
                       col < numberOfCols;col++){
          temp[row][col] =
             temp[row][col+(numberOfCols + 1)/2];
        }//col loop
      }//row loop

    }else{//shift for even
      //Slide leftmost (numberOfCols/2) columns
      // to the right by numberOfCols columns.
      for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
                     col < numberOfCols/2;col++){
          temp[row][col + numberOfCols] =
                                   temp[row][col];
        }//col loop
      }//row loop

      //Now slide everything back to the left by
      // numberOfCols/2 columns
      for(int row = 0;row < numberOfRows;row++){
        for(int col = 0;
                       col < numberOfCols;col++){
          temp[row][col] =
                 temp[row][col + numberOfCols/2];
        }//col loop
      }//row loop
    }//end else

    //Now do the vertical shift
    if(numberOfRows%2 != 0){//shift for odd
      //Slide topmost (numberOfRows+1)/2 rows
      // down by numberOfRows rows.
      for(int col = 0;col < numberOfCols;col++){
        for(int row = 0;
                 row < (numberOfRows+1)/2;row++){
          temp[row + numberOfRows][col] =
                                   temp[row][col];
        }//row loop
      }//col loop

      //Now slide everything back up by
      // (numberOfRows+1)/2 rows.
      for(int col = 0;col < numberOfCols;col++){
        for(int row = 0;
                       row < numberOfRows;row++){
          temp[row][col] =
             temp[row+(numberOfRows + 1)/2][col];
        }//row loop
      }//col loop

    }else{//shift for even
      //Slide topmost (numberOfRows/2) rows down
      // by numberOfRows rows
      for(int col = 0;col < numberOfCols;col++){
        for(int row = 0;
                     row < numberOfRows/2;row++){
          temp[row + numberOfRows][col] =
                                  temp[row][col];
        }//row loop
      }//col loop

      //Now slide everything back up by
      // numberOfRows/2 rows.
      for(int col = 0;col < numberOfCols;col++){
        for(int row = 0;
                       row < numberOfRows;row++){
          temp[row][col] =
                 temp[row + numberOfRows/2][col];
        }//row loop
      }//col loop
    }//end else

    //Shifting of the origin is complete.  Copy
    // the rearranged data from temp to output
    // array.
    for(int row = 0;row < numberOfRows;row++){
      for(int col = 0;col < numberOfCols;col++){
        output[row][col] = temp[row][col];
      }//col loop
    }//row loop

    return output;
  }//end shiftOrigin method

}//end class ImgMod30
```

Listing 1

```
/*File ImgMod31.java
Copyright 2005, R.G.Baldwin

The purpose of this program is to exercise and
test the 2D Fourier Transform methods and the
axis shifting method provided by the class named
```

```
ImgMod30.

The main method in this class reads a command-
line parameter and uses it to select a specific
case involving a particular kind of input data
in the space domain.  The program then performs
a 2D Fourier transform on that data followed by
an inverse 2D Fourier transform.

There are 14 cases built into the program with
case numbers ranging from 0 to 13 inclusive.
Each of the cases is designed such that the
results should be known in advance by a person
familiar with 2D Fourier analysis and the wave-
number domain.  The cases are also designed to
illustrate the impact of various space-domain
characteristics on the wave-number spectrum.
This information will be useful later when
analyzing the results of performing 2D
transforms on photographic images and other
images as well.

Each time the program is run, it produces a stack
of six output images in the upper left corner of
the screen.  The type of each image is listed
below.  This list is in top-to-bottom order.  To
view the images further down in the stack, you
must  physically move those on top to get them
out of the way.

The top-to-bottom order of the output images is
as follows:

1. Space-domain output of inverse Fourier
transform.  Compare with original input in 6
below.
2. Amplitude spectrum in wave-number domain with
shifted origin.  Compare with 5 below.
3. Imaginary wave-number spectrum with shifted
origin.
4. Real wave-number spectrum with shifted
origin.
5. Amplitude spectrum in wave-number domain
without shifted origin.  Compare with 2 above.
6. Space-domain input data.  Compare with 1
above.

In addition, the program produces some numeric
output on the command-line screen that may be
useful in confirming the validity of the inverse
transform.  The following is an example:

height = 41
width = 41
height = 41
width = 41
```

```
2.0 1.9999999999999916
0.5000000000000002 0.49999999999999845
0.49999999999999956 0.4999999999999923
1.7071067811865475 1.7071067811865526
0.2071067811865478 0.20710678118654233
0.20710678118654713 0.20710678118655435
1.0 1.0000000000000064
-0.4999999999999997 -0.49999999999999484
-0.5000000000000003 -0.4999999999999965

The first two lines above indicate the size of
the spatial surface for the forward transform.
The second two lines indicate the size of the
wave-number surface for the inverse transform.

The remaining nine lines indicate something
about the quality of the inverse transform in
terms of its ability to replicate the original
spatial surface.  These lines also indicate
something about the correctness or lack thereof
of the overall scaling from original input to
final output.  Each line contains a pair of
values.  The first value is from the original
spatial surface.  The second value is from the
spatial surface produced by performing an inverse
transform on the wave-number spectrum.  The two
values in each pair of values should match.  If
they match, this indicates the probability of a
valid result.  Note however that this is
a very small sampling of the values that make
up the original and replicated spatial data and
problems could arise in areas that are not
included in this small sample.  The match is very
good in the example shown above.  This example
is from Case #12.

Usage: java ImgMod31 CaseNumber DisplayType
CaseNumber from 0 to 13 inclusive.

If a case number is not provided, Case #2 will be
run by default.  If a display type is not
provided, display type 1 will be used by default.

A description of each case is provided by the
comments in this program.

See ImgMod29 for a definition of DisplayType,
which can have a value of 0, 1, or 2.

You can terminate the program by clicking on the
close button on any of the display frames
produced by the program.

Tested using J2SE 5.0 and WinXP
************************************************/
import static java.lang.Math.*;
```

```
class ImgMod31{

  public static void main(String[] args){
    //Get input parameters to select the case to
    // be run and the displayType.  See ImgMod29
    // for a description of displayType.  Use
    // default case and displayType if the user
    // fails to provide that information.
    // If the user provides a non-numeric input
    // parameter, an exception will be thrown.
    int switchCase = 2;//default
    int displayType = 1;//default
    if(args.length == 1){
      switchCase = Integer.parseInt(args[0]);
    }else if(args.length == 2){
      switchCase = Integer.parseInt(args[0]);
      displayType = Integer.parseInt(args[1]);
    }else{
      System.out.println("Usage: java ImgMod31 "
                     + "CaseNumber DisplayType");
      System.out.println(
           "CaseNumber from 0 to 13 inclusive.");
      System.out.println(
           "DisplayType from 0 to 2 inclusive.");
      System.out.println("Running case "
                  + switchCase + " by default.");
      System.out.println("Running DisplayType "
                 + displayType + " by default.");
    }//end else

    //Create the array of test data.
    int rows = 41;
    int cols = 41;

    //Get a test surface in the space domain.
    double[][] spatialData =
            getSpatialData(switchCase,rows,cols);

    //Display the spatial data.  Don't display
    // the axes.
    new ImgMod29(spatialData,3,false,
                                    displayType);

    //Perform the forward transform from the
    // space domain into the wave-number domain.
    // First prepare some array objects to
    // store the results.
    double[][] realSpect = //Real part
                          new double[rows][cols];
    double[][] imagSpect = //Imaginary part
                          new double[rows][cols];
    double[][] amplitudeSpect = //Amplitude
                          new double[rows][cols];
    //Now perform the transform
    ImgMod30.xform2D(spatialData,realSpect,
                      imagSpect,amplitudeSpect);

    //Display the raw amplitude spectrum without
    // shifting the origin first.  Display the
    // axes.
    new ImgMod29(amplitudeSpect,3,true,
                                    displayType);

    //At this point, the wave-number spectrum is
    // not in a format that is good for viewing.
    // In particular, the origin is at the upper
    // left corner.  The horizontal Nyquist
    // folding  wave-number is near the
    // horizontal center of the plot.  The
    // vertical Nyquist folding wave number is
    // near the vertical center of the plot.  It
    // is much easier for most people to
    // understand what is going on when the
    // wave-number origin is shifted to the
    // center of the plot with the Nyquist
    // folding wave numbers at the edges of the
    // plot.  The method named shiftOrigin can be
    // used to rearrange the data and to shift
    // the orgin in that manner.

    //Shift the origin and display the real part
    // of the spectrum, the imaginary part of the
    // spectrum, and the amplitude of the
    // spectrum.  Display the axes in all three
    // cases.
    double[][] shiftedRealSpect =
                 ImgMod30.shiftOrigin(realSpect);
    new ImgMod29(shiftedRealSpect,3,true,
                                    displayType);

    double[][] shiftedImagSpect =
                 ImgMod30.shiftOrigin(imagSpect);
    new ImgMod29(shiftedImagSpect,3,true,
                                    displayType);

    double[][] shiftedAmplitudeSpect =
            ImgMod30.shiftOrigin(amplitudeSpect);
    new ImgMod29(shiftedAmplitudeSpect,3,true,
                                    displayType);

    //Now test the inverse transform by
    // performing an inverse transform on the
    // real and imaginary parts produced earlier
    // by the forward transform.
    //Begin by preparing an array object to store
    // the results.
    double[][] recoveredSpatialData =
                           new double[rows][cols];
    //Now perform the inverse transform.
    ImgMod30.inverseXform2D(realSpect,imagSpect,
                            recoveredSpatialData);
    //Display the output from the inverse
    // transform.  It should compare favorably
    // with the original spatial surface.
    new ImgMod29(recoveredSpatialData,3,false,
                                displayType);

    //Use the following code to confirm correct
    // scaling. If the scaling is correct, the
    // two values in each pair of values should
    // match.  Note that this is a very small
    // subset of the total set of values that
    // make up the original and recovered
    // spatial data.
    for(int row = 0;row < 3;row++){
      for(int col = 0;col < 3;col++){
        System.out.println(
          spatialData[row][col] + " " +
           recoveredSpatialData[row][col] + " ");
      }//col
    }//row
  }//end main
  //===========================================//

  //This method constructs and returns a 3D
  // surface in a 2D array of type double
  // according to the identification of a
  // specific case received as an input
  // parameter.  There are 14 possible cases.  A
  // description of each case is provided in the
  // comments.  The other two input parameters
  // specify the size of the surface in units of
  // rows and columns.
  private static double[][] getSpatialData(
               int switchCase,int rows,int cols){

    //Create an array to hold the data.  All
    // elements are initialized to a value of
    // zero.
    double[][] spatialData =
                         new double[rows][cols];

    //Use a switch statement to select and
    // create a specified case.
    switch(switchCase){
      case 0:
        //This case places a single non-zero
        // point at the origin in the space
        // domain.  The origin is at the upper
        // left corner.  In signal processing
        // terminology, this point can be viewed
        // as an impulse in space.  This produces
        // a flat spectrum in wave-number space.
        spatialData[0][0] = 1;
      break;
      case 1:
        //This case places a single non-zero
        // point near but not at the origin in
        // space.  This produces a flat spectrum
        // in wave-number space as in case 0.
        // However, the real and imaginary parts
        // of the transform are different from
        // case 0 and the result is subject to
        // arithmetic accuracy issues.  The
        // plotted flat spectrum doesn't look
        // very good because the color switches
        // back and forth between three values
        // that are very close to together.  This
        // is the result of the display program
        // normalizing the surface values based
        // on the maximum and minimum values,
        // which in this case are very close
        // together.
        spatialData[2][2] = 1;
      break;

      case 2:
        //This case places a box on the diagonal
        // near the origin. This produces a
        // sin(x)/x shape to the spectrum with
        // its peak at the origin in wave-number
        // space.
        spatialData[3][3] = 1;
        spatialData[3][4] = 1;
        spatialData[3][5] = 1;
        spatialData[4][3] = 1;
        spatialData[4][4] = 1;
        spatialData[4][5] = 1;
        spatialData[5][3] = 1;
        spatialData[5][4] = 1;
        spatialData[5][5] = 1;
      break;

      case 3:

        //This case places a box at the top near
        // the origin.  This produces the same
        // amplitude spectrum as case 2. However,
        // the real and imaginary parts, (or the
        // phase) is different from case 2 due to
        // the difference in location of the box
        // relative to the origin in space.
        spatialData[0][3] = 1;
        spatialData[0][4] = 1;
        spatialData[0][5] = 1;
        spatialData[1][3] = 1;
        spatialData[1][4] = 1;
        spatialData[1][5] = 1;
        spatialData[2][3] = 1;
        spatialData[2][4] = 1;
        spatialData[2][5] = 1;
      break;

      case 4:
        //This case draws a short line along the
        // diagonal from upper left to lower
        // right. This results in a spectrum with
        // a sin(x)/x shape along that axis and a
        // constant along the axis that is
        // perpendicular to that axis
        spatialData[0][0] = 1;
        spatialData[1][1] = 1;
        spatialData[2][2] = 1;
        spatialData[3][3] = 1;
        spatialData[4][4] = 1;
        spatialData[5][5] = 1;
        spatialData[6][6] = 1;
        spatialData[7][7] = 1;
      break;

      case 5:
        //This case draws a short line
        // perpendicular to the diagonal from
        // upper left to lower right.  The
        // spectral result is shifted 90 degrees
        // relative to that shown for case 4
        // where the line was along the diagonal.
        // In addition, the line is shorter
        // resulting in wider lobes in the
        // spectrum.
        spatialData[0][3] = 1;
        spatialData[1][2] = 1;
        spatialData[2][1] = 1;
        spatialData[3][0] = 1;
      break;

      case 6:
          //This case draws horizontal lines,
          // vertical lines, and lines on both
          // diagonals.  The weights of the
          // individual points is such that the
          // average of all the weights is 0.
          // The weight at the point where the
          // lines intersect is also 0.  This
          // produces a spectrum that is
          // symmetrical across the axes at 0,
          // 45, and 90 degrees.  The value of
          // the spectrum at the origin is zero
          // with major peaks at the folding
          // wave-numbers on the 45-degree axes.
          // In addition, there are minor peaks
          // at various other points as well.
          spatialData[0][0] = -1;
          spatialData[1][1] = 1;
          spatialData[2][2] = -1;
          spatialData[3][3] = 0;
          spatialData[4][4] = -1;
          spatialData[5][5] = 1;
          spatialData[6][6] = -1;

          spatialData[6][0] = -1;
          spatialData[5][1] = 1;
          spatialData[4][2] = -1;
          spatialData[3][3] = 0;
          spatialData[2][4] = -1;
          spatialData[1][5] = 1;
          spatialData[0][6] = -1;

          spatialData[3][0] = 1;
          spatialData[3][1] = -1;
          spatialData[3][2] = 1;
          spatialData[3][3] = 0;
          spatialData[3][4] = 1;
          spatialData[3][5] = -1;
          spatialData[3][6] = 1;

          spatialData[0][3] = 1;
          spatialData[1][3] = -1;
          spatialData[2][3] = 1;
          spatialData[3][3] = 0;
          spatialData[4][3] = 1;
          spatialData[5][3] = -1;
          spatialData[6][3] = 1;
      break;

      case 7:
        //This case draws a zero-frequency
        // sinusoid (DC) on the surface with an
        // infinite number of samples per cycle.
        // This causes a single peak to appear in
        // the spectrum at the wave-number
        // origin.  This origin is the upper left
        // corner for the raw spectrum, and is
        // at the center cross hairs after the
        // origin has been shifted to the
        // center for better viewing.
        for(int row = 0; row < rows; row++){
          for(int col = 0; col < cols; col++){
            spatialData[row][col] = 1.0;
          }//end inner loop
        }//end outer loop
      break;

      case 8:
        //This case draws a sinusoidal surface
        // along the horizontal axis with one
        // sample per cycle. This function is
        // under-sampled by a factor of 2.
        // This produces a single peak in the
        // spectrum at the wave number origin.
        // The result is the same as if the
        // sinusoidal surface had zero frequency
        // as in case 7..
        for(int row = 0; row < rows; row++){
          for(int col = 0; col < cols; col++){
            spatialData[row][col] =
                                     cos(2*PI*col/1);
          }//end inner loop
        }//end outer loop
      break;

      case 9:
        //This case draws a sinusoidal surface on
        // the horizontal axis with 2 samples per
        // cycle.  This is the Nyquist folding
        // wave number.  This causes a single
        // peak to appear in the spectrum at the
        // negative folding wave number on the
        // horizontal axis.  A peak would also
        // appear at the positive folding wave
        // number if it were visible, but it is
        // one unit outside the boundary of the
        // plot.
        for(int row = 0; row < rows; row++){
          for(int col = 0; col < cols; col++){
            spatialData[row][col] =
                                     cos(2*PI*col/2);
          }//end inner loop
        }//end outer loop
      break;

      case 10:
        //This case draws a sinusoidal surface on
        // the vertical axis with 2 samples per
        // cycle.  Again, this is the Nyquist
        // folding wave number but the sinusoid
        // appears along a different axis.  This
        // causes a single peak to appear in the
        // spectrum at the negative folding wave
        // number on the vertical axis.  A peak
        // would also appear at the positive
        // folding wave number if it were
        // visible, but it is one unit outside
        // the boundary of the plot.
        for(int row = 0; row < rows; row++){
          for(int col = 0; col < cols; col++){
            spatialData[row][col] =
                                     cos(2*PI*row/2);
          }//end inner loop
        }//end outer loop
      break;

      case 11:
        //This case draws a sinusoidal surface on
        // the horizontal axis with 8 samples per
        // cycle. You might think of this surface
        // as resembling a sheet of corrugated
        // roofing material.  This produces
        // symmetrical peaks on the horizontal
        // axis on either side of the wave-
        // number origin.
        for(int row = 0; row < rows; row++){
          for(int col = 0; col < cols; col++){
            spatialData[row][col] =
                                     cos(2*PI*col/8);
          }//end inner loop
        }//end outer loop
      break;

      case 12:
        //This case draws a sinusoidal surface on
        // the horizontal axis with 3 samples per
        // cycle plus a sinusoidal surface on the
        // vertical axis with 8 samples per
        // cycle. This produces symmetrical peaks
        // on the horizontal and vertical axes on
        // all four sides of the wave number
        // origin.
        for(int row = 0; row < rows; row++){
          for(int col = 0; col < cols; col++){
            spatialData[row][col] =
               cos(2*PI*row/8) + cos(2*PI*col/3);
          }//end inner loop
        }//end outer loop
      break;

      case 13:
        //This case draws a sinusoidal surface at
        // an angle of approximately 45 degrees
        // relative to the horizontal.  This
        // produces a pair of peaks in the
        // wave-number spectrum that are
        // symmetrical about the origin at
        // approximately 45 degrees relative to
        // the horizontal axis.
        double phase = 0;
        for(int row = 0; row < rows; row++){
          for(int col = 0; col < cols; col++){
            spatialData[row][col] =
                           cos(2.0*PI*col/8 - phase);
          }//end inner loop
          //Increase phase for next row
          phase += .8;
        }//end outer loop
      break;

      default:
        System.out.println("Case must be " +
                  "between 0 and 13 inclusive.");
        System.out.println(
                         "Terminating program.");
        System.exit(0);
    }//end  switch statement

    return spatialData;
```

```
  }//end getSpatialData
}//end class ImgMod31

Listing 2
```

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

Keywords
Java space time wavenumber frequency domain two-dimensional one-dimensional 2D Fourier transform spectra antenna array complex spectral DSP transmit wavelength surface Nyquist folding seismology radio astronomy sonar radar

-end-