# 2D Fourier Transforms using Java, Part 2

*Examine the code for a Java class that can be used to perform forward and inverse 2D Fourier transforms on 3D surfaces in the space domain. Learn how the 2D Fourier transform behaves for a variety of different sample surfaces in the space domain.*

**Published:** August 9, 2005
**By Richard G. Baldwin**

Java Programming, Notes # 1491



---

## Preface

This is the second part of a two-part lesson. The first part published earlier was entitled 2D Fourier transforms using Java. 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 entitled 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

In addition, I strongly recommend that you study Part 1 of this lesson before embarking on this part. 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

**The space domain**

In Part 1 of this lesson, I extended the concept of the Fourier transform from the time domain into the *space domain.* I pointed out that while the time domain is one-dimensional, the space domain is thee-dimensional. However, in order to keep the complexity of this lesson in check, we will assume that space is only two-dimensional. 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.)*

**A purely real space domain**

Although the space domain can be *(and often is)* complex, many interesting problems, *(such as photographic image processing)* can be dealt with under the assumption that the space domain is

purely real. We will make that assumption in this lesson. This assumption 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.

# Preview

I will present and explain two complete Java programs in this lesson. The first program is a single class named **ImgMod30**, which provides the capability to perform forward and inverse Fourier transforms on three-dimensional surfaces. In addition, the class provides a method that can be used to reformat the wavenumber spectrum to make it more suitable for display.

The second program is named **ImgMod31**. This is an executable program whose purpose is to exercise and to test the **ImgMod30** class using several examples for which the results should already be known.

# Sample Programs

### The class named ImgMod30

This class provides 2D Fourier transform capability that can be used for image processing and other purposes. The class provides three static methods:

- **xform2D**: Performs a forward 2D Fourier transform on a purely real 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 as a 2D array of double values. Assumes that the real and imaginary parts in the wavenumber domain are consistent with a purely real surface in the space domain, and does not return an imaginary surface for the space domain
- **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.

The class was tested using J2SE 5.0 and WinXP. The class uses the *static import* capability that was introduced in J2SE 5.0. Therefore, it should not compile using earlier versions of the compiler.

### The xform2D method

The beginning of the class and the beginning of the static method named **xform2D** is shown in Listing 1.

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, this is not a general purpose 2D complex-to-complex transform.

```
class ImgMod30{
  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);
```

**Listing 1**

## Parameters

The first parameter is a reference to a 2D **double** array object containing the data to be transformed. The remaining three parameters are references to 2D **double** array objects of the same size that will be populated with the following transform results:

- The real part
- The imaginary part
- The amplitude *(square root of sum of squares of the real and imaginary parts)*

Listing 1 also determines and displays the dimensions of the incoming 2D array of data to be transformed.

I won't bore you with the details as to how and why the 2D Fourier transform does what it does. Neither will I bore you with the details of the code that implements the 2D Fourier transform. If you understand the material that I have previously published on Fourier transforms in one dimension, this code and these concepts should be a straightforward extension from one dimension to two dimensions.

## The remainder of the xform2D method

The remainder of the xform2D method is shown in Listing 2. Note that it was necessary to sacrifice indentation in order to force these very long equations to be compatible with this narrow publication format and still be somewhat readable.

```
    //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.
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
```

**Listing 2**

## The inverseXform2D method

The **inverseXform2d** method is shown in its entirety in Listing 3. 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 wavenumber domain in both dimensions.

This method assumes that the inverse transform will produce purely real values in the space domain. Therefore, in the interest of computational efficiency, the method does not compute the imaginary output values. Therefore, this 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.

Once again it was necessary to sacrifice indentation to force this very long equation to be compatible with this narrow publication format and still be readable.

```
  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.
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
```

**Listing 3**

## Parameters

This method requires three parameters. The first two parameters are references to 2D arrays containing the real and imaginary parts of the complex wavenumber spectrum that is to be transformed into a surface in the space domain.

The third parameter is a reference to a 2D array of the same size that will be populated with the transform results.

## The shiftOrigin method

This method deserves some explanation. The reason that this method is needed is illustrated by Figure 1.

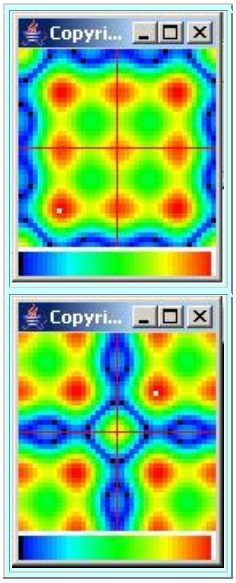



Figure 1

## Two views of the same wavenumber spectrum

Both of the images in Figure 1 represent the same wavenumber spectrum, but they are plotted against different coordinate systems.

## How the wavenumber spectrum is actually computed

The top image shows how the wavenumber spectrum is actually computed.

The wavenumber spectrum is computed covering an area of wavenumber space with the 0,0 origin in the upper left corner. The computation extends to twice the Nyquist folding wave number along each axis.

## Computationally sound but not visually pleasing

While this format is computationally sound, it isn't what most of us are accustomed to seeing in plots wavenumber space. Rather, we are accustomed to seeing wavenumber spectra plotted with the 0,0 origin at the center.

## The wavenumber spectrum is periodic

Knowing that the area of wavenumber space shown in the top image of Figure 1 covers one complete period of a periodic surface, the **shiftOrigin** method rearranges the results *(for display purposes only)* to that shown in the bottom image in Figure 1. The origin is at the center in the bottom image of Figure 1. The edges of the lower image in Figure 1 are the Nyquist folding wave numbers.

## The shiftOrigin method code

The **shiftOrigin** method is shown in its entirety in Listing 4. Although this method is rather long, it is also completely straightforward. Therefore, it shouldn't require a further explanation. You may be able to develop a much shorter algorithm for accomplishing the same task.

```
  //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
  // wavenumber 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 4**

## End of the ImgMod30 class

Listing 4 also signals the end of the class definition for the class named **ImgMod30**.

## The program named ImgMod31

The purpose of this program is to exercise and to test the 2D Fourier transform methods and the axis shifting method provided by the class named **ImgMod30**.

## Command line parameters

The **main** method in this class reads two command line parameters and uses them to select:

- A specific case involving a particular 3D input surface in the space domain.
- A specific display format.

## Forward and inverse Fourier transforms

The program performs a 2D Fourier transform on that surface followed by an inverse 2D Fourier transform. Six different plots are produced in this process showing different aspects of the transform and the inverse transform.

## Fourteen cases

There are 14 different cases built into the program with case numbers ranging from 0 to 13 inclusive. Each of the cases is designed such that the results of the analysis should be known in advance by a person familiar with 2D Fourier analysis and the wavenumber domain. Thus, these cases can be used to confirm that the transform code was properly written.

The cases are also designed to illustrate the impact of various space domain characteristics on the wavenumber spectrum. This information will be useful later when analyzing the results of performing 2D transforms on photographic images.

## A stack of output images

Each time the program is run, it produces a stack of six output images in the upper left corner of the screen. A brief description of each of the output images is provided in the following list. The top-to-bottom order of the stack is:

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

To view the images near the bottom of the stack, you must physically move those on top to get them out of the way.

## Numeric output

In addition, the program produces some numeric output on the command line screen that may be useful in confirming the validity of the forward and inverse transformation processes. Figure 2 shows an example of the numeric output.

```
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
Figure 2
```

*(Note that I manually inserted some spaces line breaks in Figure 2 to cause the numeric values to line up in columns so as to be more readable.)*

## The size of the surfaces

The first two lines of numeric output in Figure 2 show the size of the spatial surface for the forward transform. The second two lines show the size of the wavenumber surface for the inverse transform.

## The quality of the transformation process

The remaining nine lines indicate something about the quality of the forward and inverse transforms in terms of the ability of the inverse transform to replicate the original spatial surface. These lines also indicate something about the correctness of the overall scaling from original input to final output.

## Matching pairs of values

Each of the last nine lines contains a pair of values. The first value is a sample from the original spatial surface. The second value is a sample from the corresponding location on the spatial surface produced by performing an inverse transform on the wavenumber 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.

## How to use the program named ImgMod31

Usage information for the program is shown in Figure 3.

```
Usage:
java ImgMod31 CaseNumber DisplayType

CaseNumber ranges from 0 to 13 inclusive.

DisplayType ranges from 0 to 2 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.

**Figure 3**

A description of each case is provided by the comments in this program. In addition, each case will be discussed in detail in this lesson.

See **ImgMod29** in the earlier lesson entitled Plotting 3D Surfaces using Java for a definition of **DisplayType**.

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

## Let's see some code

The beginning of the class and the beginning of the **main** method is shown in Listing 5.

```
class ImgMod31{

  public static void main(String[]
args){
    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

Listing 5
```

The code in Listing 5 gets the input parameters and uses them to set the case and the display format. A default case and a default display format are used if this information is not provided by the user.

## Create and save the test surface

Listing 6 invokes the method named **getSpatialData** to create a test surface that matches the specified case. This surface will be used for testing the transformation process.

```
    int rows = 41;
    int cols = 41;

    double[][] spatialData =

getSpatialData(switchCase,rows,cols);

Listing 6
```

I will discuss the method named **getSpatialData** in detail later. For now, just assume that the 2D array object referred to by **spatialData** contains the test surface when this method returns.

## Display the test surface

Listing 7 instantiates an object of the class named **ImgMod29** to display the test surface in the display format indicated by the value of **displayType**.

```
    new ImgMod29(spatialData,3,false,

displayType);

Listing 7
```

The value of **false** in the third parameter indicates that the axes should not be displayed.

> *(See the lesson entitled* Plotting 3D Surfaces using Java *for an explanation of the second parameter. Basically, this parameter is used to control the overall size of the plot.)*

## An example test surface plot

The upper left image in Figure 4 is an example of the output produced by the code in Listing 7 for a **displayType** value of 0**.**

*(Figure 4 shows the grayscale format. See the lesson entitled* Plotting 3D Surfaces using Java *for an explanation of the three available display formats.)*

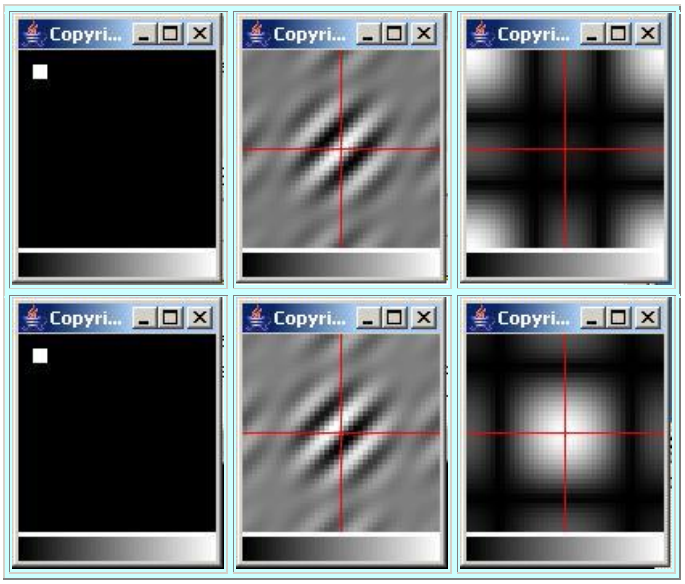



Figure 4

Figure 4 shows the results for a test surface **switchCase** value of 2. I will discuss the particulars of this case in detail later.

## Perform the forward Fourier transform

Listing 8 performs the forward Fourier transform to transform the test surface into the wavenumber domain.

```
    double[][] realSpect = //Real part
                           new
double[rows][cols];
    double[][] imagSpect = //Imaginary
part
                           new
double[rows][cols];
    double[][] amplitudeSpect =
//Amplitude
                           new
double[rows][cols];


ImgMod30.xform2D(spatialData,realSpect,

imagSpect,amplitudeSpect);

Listing 8
```

## Prepare array objects to receive the transform results

Listing 8 begins by preparing some array objects to receive the transform results. The forward transform receives an incoming surface array and returns the real and imaginary parts of the complex wavenumber spectrum along with the amplitude spectrum by populating three array objects passed as parameters to the method.

## Perform the forward transform

Then Listing 8 invokes the static **xform2D** method of the **ImgMod30** class to perform the forward transform, returning the results by way of the parameters to the method.

## Display unshifted amplitude spectrum

The upper right image in Figure 4 is an example of the type of display produced by the code in Listing 9. This is a plot of the amplitude spectrum without the wavenumber origin being shifted to place it at the center.

> *(The wavenumber origin is in the upper left corner of the upper right image in Figure 4.)*

This image also shows the result of passing **true** as the third parameter causing the red axes to be plotted on top of the spectral data.

```
    new
ImgMod29(amplitudeSpect,3,true,

displayType);
```

**Listing 9**

## Need to shift the origin for display

The upper right image in Figure 4 is in a format that is not particularly good for viewing. In particular, the origin is at the upper left corner. The horizontal Nyquist folding wavenumber 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 the plot when the wavenumber 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 shift the origin to the center of the plot.

## Shift the origin and display the results

Listing 10 shifts the origin to the center of the plot and displays:

- The real part of the shifted spectrum
- The imaginary part of the shifted spectrum
- The amplitude of the shifted spectrum

The axes are displayed 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);
```

**Listing 10**

## Example displays

Examples of the displays produced by the code in Listing 10 are shown in Figure 4. The real part of the shifted wavenumber spectrum is shown in the image in the top center of Figure 4. The imaginary part of the shifted wavenumber spectrum is shown in the bottom center of Figure 4. The shifted amplitude spectrum is shown in the bottom right image in Figure 4. The origin has been shifted to the center in all three cases.

> *(It would probably be constructive for you to compare the two rightmost images in Figure 4 in order to appreciate the result of shifting the origin to the center.)*

## Perform an inverse transform

Listing 11 performs an inverse Fourier transform to transform the complex wavenumber surface into a real surface in the space domain. Ideally, the result should exactly match the space domain surface that was transformed into the wavenumber domain in Listing 8. However, because of small arithmetic errors that accumulate in the forward and inverse transform computations, it is unusual for an exact match to be achieved.

```
    double[][] recoveredSpatialData =
                              new
double[rows][cols];


ImgMod30.inverseXform2D(realSpect,imagSpect,

recoveredSpatialData);

Listing 11
```

**Prepare an array object to store the results**

Listing 11 begins by preparing an array object to store the results of the inverse transformation process.

**Invoke the inverseXform2D method**

Then Listing 11 invokes the **inverseXform2D** method to transform the complex wavenumber spectrum into a real space function. The **inverseXform2D** method requires the real and imaginary parts of the complex wavenumber spectrum as input parameters.

> *(Note that these are the original real and imaginary parts of the complex wavenumber spectrum. They are not the versions for which the origin has been shifted for display purposes.)*

The **inverseXform2D** method also receives an incoming array object in which to store the real result of the transformation process.

**Display the result of the inverse transform**

Finally, Listing 12 displays the result of the inverse transformation process as a surface in the space domain. This surface should compare favorably with the original surface that was transformed into the wavenumber domain in Listing 8

```
    new
ImgMod29(recoveredSpatialData,3,false,

displayType);

Listing 12
```

The output produced by Listing 12 is shown in the lower left image in Figure 4. Compare this with the input surface shown in the upper left image in Figure 4. As you can see, they do compare favorably. In fact, they appear to be identical in this grayscale plotting format. We will see later that when a more sensitive plotting format is used, small differences in the two may become apparent.

## Display some numeric results

As discussed earlier, the code in Listing 13 samples and displays a few corresponding points on the original surface and the surface produced by the inverse transformation process. The results can be used to evaluate the overall quality of the process as well as the correctness of the overall scaling.

```
    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

Listing 13
```

Each line of output text contains two values, and ideally the two values should be exactly the same. Realistically, because of small computational errors in the transform and inverse transform process, it is unlikely that the two values will be exactly the same except in computationally trivial cases. Unless the two values are very close, however, something probably went wrong in the transformation process and the results should not be trusted.

Listing 13 also signals the end of the **main** method.

## What we know so far ...

Now we know how to use the **ImgMod30** class and the **ImgMod29** class to:

- Transform a purely real 3D surface from the space domain into the wavenumber domain
- Transform a complex wavenumber spectrum into a purely real surface in the space domain
- Shift the origin of the real, imaginary, and amplitude wavenumber spectral parts to convert the data into a format that is more suitable for plotting
- Plot 3D surfaces in both domains

It is time to for us to take a look at the method named **getSpatialData** that can be used to create any of fourteen standard surfaces in the space domain.

## The getSpatialData method

This method constructs and returns a specific 3D surface in a 2D array of type **double**. The surface is identified by the value of an incoming parameter named **switchCase**. There are 14 possible cases. The allowable values for **switchCase** range from 0 through 13 inclusive.

The other two input parameters specify the size of the surface that will be produced in units of rows and columns.

## The code for the getSpatialData method

The **getSpatialData** method begins in Listing 14.

```
  private static double[][]
getSpatialData(
               int switchCase,int
rows,int cols){

    double[][] spatialData =
                           new
double[rows][cols];

    switch(switchCase){
```

**Listing 14**

Listing 14 begins by creating a 2D array object of type **double** in which to store the surface.

Then Listing 14 shows the beginning of a **switch** statement that will be used to select the code to create a surface that matches the value of the incoming parameter named **switchCase**.

## For switchCase = 0

Listing 15 shows the code that is executed for a value of **switchCase** equal to 0.

```
      case 0:
        spatialData[0][0] = 1;
      break;
```

**Listing 15**

This case places a single non-zero point at the origin in the space domain. The origin is at the upper left corner. The surface produced by this case is shown in the leftmost image in Figure 5, and the non-zero value can be seen as the small white square in the upper left corner. In signal processing terminology, this point can be viewed as an impulse in space. It is well known that such an impulse produces a flat spectrum in wavenumber space.

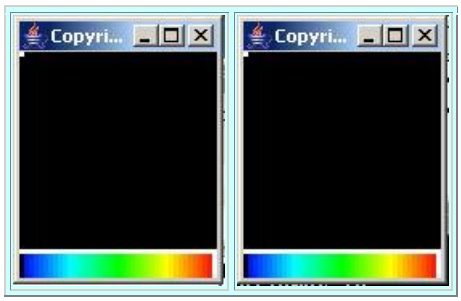



Figure 5

## The output surface

The rightmost image in Figure 5 shows the result of:

- Performing a forward Fourier transform on the surface in the leftmost image
- Performing an inverse Fourier transform on the complex wavenumber spectrum produced by the forward transform.

You can see the impulse as the small white square in the upper left corner of both images.

## The wavenumber spectrum is flat

Because the wavenumber spectrum is flat, plots of the spectrum are completely featureless. Therefore, I did not include them in Figure 5.

## A very small error

The numeric output shows that the final output surface matches the input surface to within an error that is less than about one part in ten to the fourteenth power. The program produces the expected results for this test case.

If you were to go back to the equations in Listing 2 and Listing 3 and work this case out by hand, you would soon discover that the computational requirements are almost trivial. Most of the computation involves doing arithmetic using values of 1 and 0. Thus, there isn't a lot of opportunity for computational errors in this case.

## For switchCase = 1

Now we are going to a case that is more significant from a computational viewpoint. The input surface in this case will consist of a single impulse that is not located at the origin in the space domain. Rather, it is displaced from the origin.

The wavenumber amplitude spectrum of a single impulse in the space domain should be flat regardless of the location of the impulse in the space domain. However, the real and imaginary

parts of the wavenumber spectrum are flat only when the impulse is located at the origin in space. This case does not satisfy that requirement.

Regardless of the fact that the real and imaginary parts are not flat, the square root of the sum of the squares of the real and imaginary parts *(the amplitude)* should be the same for every point in wavenumber space for this case. Thus, the real and imaginary parts are related in a very special way.

## The code for switchCase = 1

The code that is executed when **switchCase** equals 1 is shown in Listing 16.

```
      case 1:
        spatialData[2][2] = 1;
      break;

Listing 16
```

This case places a single impulse close to but not at the origin in space. This produces a flat amplitude spectrum in wavenumber space just like in case 0. However, the real and imaginary parts of the spectrum are different from case 0. They are not flat. The computations are probably more subject to errors in this case than for case 0.

## The visual output

Figure 6 shows the six output images produced by the program for a **switchCase** value of 1.

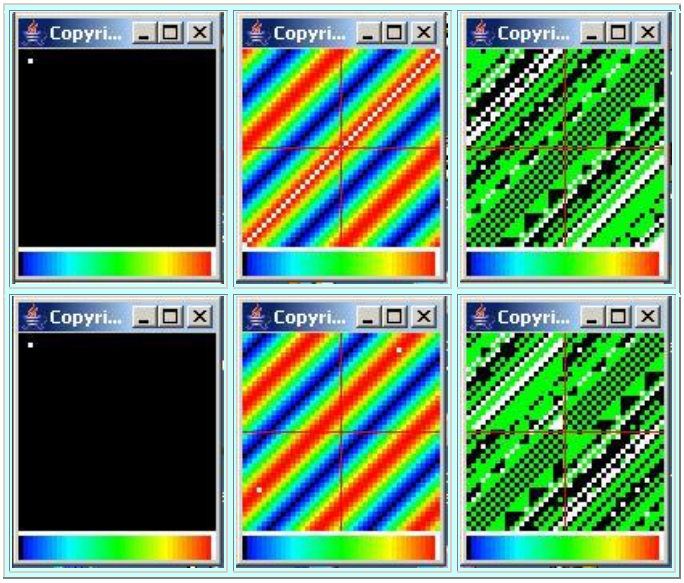



Figure 6

## The input and output surfaces match visually

The input and output surfaces showing the single impulse are in the two leftmost images in Figure 6. From a visual viewpoint, the output at the bottom appears to be an exact match for the input at the top.

## The real and imaginary parts

The real and imaginary parts of the wavenumber spectrum are shown in the two center images. The real part is at the top and the imaginary part is at the bottom.

For a single impulse in the space domain, we would expect each of these surfaces to be a 3D sinusoidal wave *(similar to a piece of corrugated sheet metal).* That appears to be what we are seeing, with almost two full cycles of the sinusoidal wave between the origin and the bottom right corner of the image.

> *(The distance between the peaks in the sinusoidal wave in wavenumber space is inversely proportional to the distance of the impulse from the origin in space. Hence, as the impulse approaches the origin in space, the peaks in wavenumber space become further and further apart. When the impulse is located at the origin in space, the distance between the peaks in wavenumber space becomes infinite, leading to flat real and imaginary parts.)*

## Symmetry

We know that the real part of a wavenumber spectrum resulting from the Fourier transform of a real space function is symmetric about the origin. We also know that the imaginary part is asymmetric about the origin.

The symmetry/asymmetry requirements appear to be satisfied by this case. The color bands in the real part at the top are symmetric on either side of the origin.

The imaginary part is asymmetric about the origin *(the centers of the red/white and the blue/black bands appear to be equidistant from and on opposite sides of the origin).*

## The amplitude spectrum is ugly

The amplitude spectrum is shown in the two rightmost images in Figure 6. The unshifted amplitude spectrum is shown at the top. The amplitude spectrum with the origin shifted to the center is shown at the bottom.

The ugliness of these two plots is an artifact of the 3D plotting scheme implemented by the class named **ImgMod29**. In order to maximize the use of the available dynamic range in the plot, each surface that is plotted is normalized such that:

- The highest elevation is colored white
- The lowest elevation is colored black

- Elevations between the highest and lowest values are colored according to the calibration scale below the image

This normalization is applied even when the distance between the highest and lowest elevation is very small. As a result of computational errors, the amplitude spectrum is not perfectly flat. Rather there are very small variations from one point to the next. As a result, the colors used to plot the surface switch among the full range of available colors even for tiny deviations from perfect flatness.

**A very small error**

The total error for this case is very small. The numeric output shows that the final output surface matches the input surface to within an error that is less than about one part in ten to the thirteenth power. The program produces the expected results for this test case.

**For switchCase = 2**

Now we are going to take a look at another case for which we know in advance generally what the outcome should be. This will allow us to compare the outcome with our expectations to confirm proper operation of the program.

**A box on the diagonal in space**

This case places a box that is one unit tall on the diagonal near the origin in the space domain as shown in the upper left image in Figure 7.

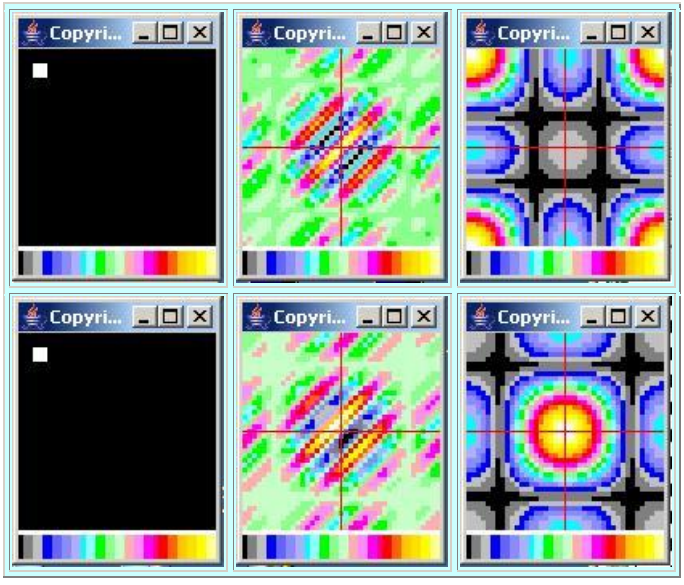



Figure 7

**What do we know?**

On the basis of prior experience, we know that the amplitude spectrum of this surface along the horizontal and vertical axes of the wavenumber spectrum should have a rectified sin(x)/x shape *(all negative values are converted to positive values).* We know that the peak in this amplitude spectrum should appear at the origin in wavenumber space, and that the width of the peak should be inversely proportional to the size of the box.

## The code for switchCase = 2

The code that constructs the space domain surface for this case is shown in Listing 17.

```
      case 2:
        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;

Listing 17
```

This code is completely straightforward. It sets the value of each of nine adjacent points on the surface to a value of 1, while the values of all other points on the surface remain at zero. The arrangement of those nine points forms a square whose sides are parallel to the horizontal and vertical axes.

## The real and imaginary parts of the spectrum

There isn't a lot that I can tell you about what to expect regarding the real and imaginary parts of this spectrum, other than that they should exhibit the same symmetry and asymmetry conditions that I described earlier for the real and imaginary parts in general. These requirements appear to be satisfied by the real part at the top center of Figure 7 and the imaginary part at the bottom center of Figure 7.

Otherwise, the shape of the real and imaginary wavenumber spectra will depend on the location of the box in space and the size and orientation of the box.

## A different plotting color scheme

Note that the plotting color scheme that I used for Figure 7 is different from any of the plots previously shown in this lesson.
This color scheme is what I refer to as the *Color Contour* scheme in the lesson entitled Plotting 3D Surfaces using Java.

This scheme quantizes the range from the lowest to the highest elevation into 23 levels, coloring the lowest elevation black, the highest elevation white, and assigning very specific colors to the 21 levels in between. The colors and the levels that they represent are shown in the calibration scales under the plots in Figure 7. The lowest elevation is on the left end of the calibration scale. The highest elevation is on the right end of the calibration scale.

**The amplitude spectrum**

As before, the wavenumber amplitude spectrum with the origin in the upper left corner is shown in the upper right image in Figure 7. The amplitude spectrum with the origin shifted to the center is shown in the lower right image in Figure 7.

If you were to use the calibration scale to convert the colors along the horizontal and vertical axes in the lower right image into numeric values, you would find that they approximate a rectified sin(x)/x shape as expected.

**The output surface**

The output surface produced by performing an inverse Fourier transform on the complex wavenumber spectrum is shown in the lower left image in Figure 7. This surface appears to match the input surface shown in the upper left image in Figure 7.

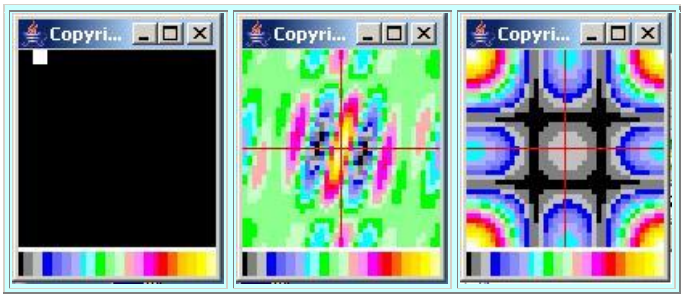
**The overall results**

The numeric output for this case isn't very useful because none of the samples for which numeric data is provided fall within the square. However, because the real and imaginary parts exhibit the correct symmetry, the shape of the amplitude spectrum is generally what we expect, and the output from the inverse Fourier transform appears to match the original input causes us to conclude that the program is working properly in this case.

**For switchCase = 3**

This case places a raised box at the top near the origin in the space domain, but the box is not on the diagonal as it was in case 2.

*(See the upper left image in Figure 8 for the new location of the box.)*



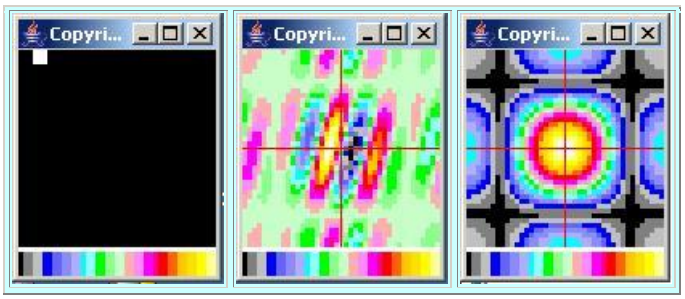



Figure 8

**Amplitude spectrum should not change**

As long as the size and the orientation of the box doesn't change, the wavenumber amplitude spectrum should be the same as case 2 regardless of the location of the box in space. Since the size and orientation of this box is the same as in case 2, the amplitude spectrum for this case should be the same as for case 2.

**The real and imaginary parts of the spectrum may change**

However, the real and imaginary parts *(or the phase)* change as the location of the box changes relative to the origin in space.

**A hypothetical example**

The purpose of this case is to illustrate a hypothetical example. If two different photographic images contain a picture of the same object in the same size and the same orientation in space, that object will contribute the same values to the amplitude spectrum of both images regardless of where the object is located in the different images.

For example, assume that a photographic image includes a picture of a vase. Assume that the original image is cropped twice along two different borders producing two new images. Assume that both of the new images contain the picture of the vase, but in different locations. That vase will contribute the same values to the amplitude spectra of the two images regardless of the location of the vase in each of the images. This knowledge will be useful to us in future lessons when we begin using 2D Fourier transforms to process photographic images.

**Amplitude spectrum is the same**

If you compare Figure 8 with Figure 7, you will see that the amplitude spectrum is the same for both surfaces despite the fact that the box is in a different location in each of the two surfaces. However, the real and imaginary parts of the spectrum in Figure 8 are considerably different from the real and imaginary parts of the spectrum in Figure 7.

The code that was used to create the surface for this case is straightforward. You can view that code in Listing 22 near the end of the lesson.

## For switchCase = 4

This case draws a short line containing eight points along the diagonal from upper left to lower right in the space domain. You can view this surface in the upper left image in Figure 9. You can view the code that generated this surface in Listing 22 near the end of the lesson.

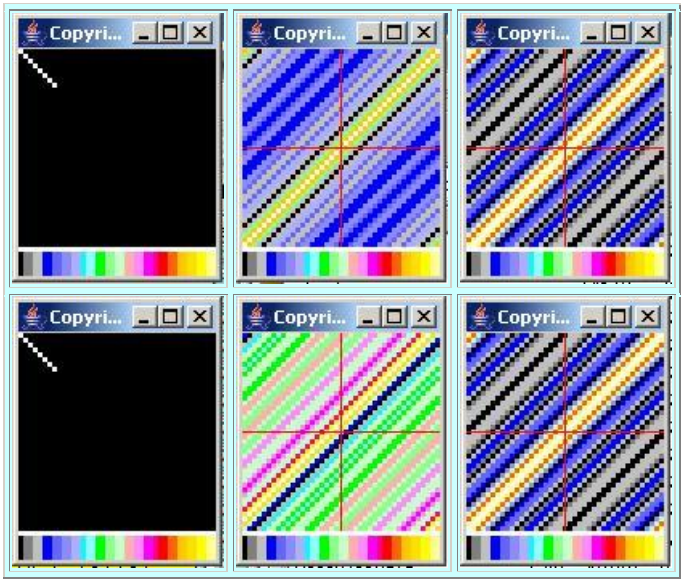



Figure 9

## Another example of sin(x)/x

On the basis of prior experience, we would expect the wavenumber amplitude spectrum, *(when viewed along any line in wavenumber space parallel to the line in space),* to have a rectified sin(x)/x shape. We would expect the peak of that shape to be centered on the origin in wavenumber space. We would expect the width of the peak in wavenumber space to be inversely proportional to the length of the line in space.

We would expect the amplitude spectrum when viewed along any line in wavenumber space perpendicular to the line in space to have a constant value.

## Our expectations are borne out

The shape of the amplitude spectrum shown in the lower right image in Figure 9 agrees with our expectations. Although not shown here, if we were to make the line of points longer, the width of the peak in the rectified sin(x)/x would become narrower. If we were to make the line of points shorter, the peak would become wider, as we will demonstrate in case 5.

Our expectations regarding symmetry and asymmetry for the real and imaginary parts shown in the center images of Figure 9 are borne out. The real part is at the top center and the imaginary part is at the bottom center.

The output from the inverse Fourier transform shown in the bottom left of Figure 9 matches the original space domain surface in the top left of Figure 9.

**For switchCase = 5**

This case draws a short line consisting of only four points perpendicular to the diagonal from upper left to lower right. This line of points is perpendicular to the direction of the line of points in case 4.

You can view the surface for this case in the upper left image of Figure 10. You can view the code that generated this surface in Listing 22 near the end of the lesson.

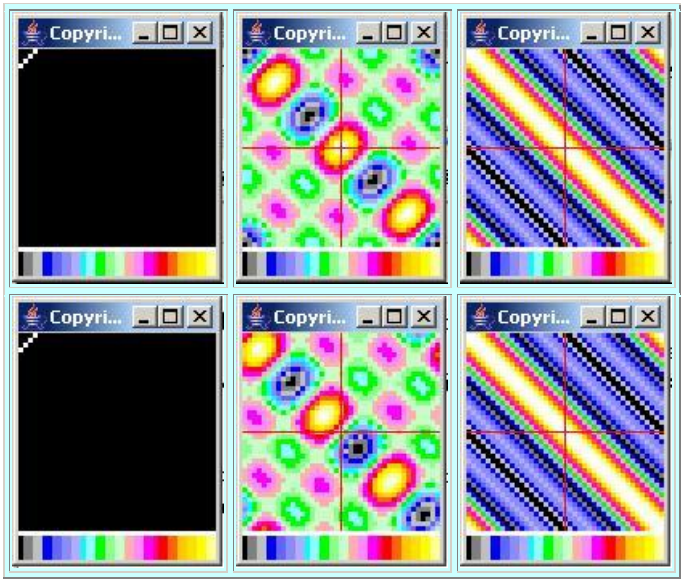



Figure 10

**Rotated by ninety degrees**

If you compare Figure 10 with Figure 9, you will see that the spectral result is rotated ninety degrees relative to that shown for case 4 where the line was along the diagonal. In other words, rotating the line of points by ninety degrees also rotated the structure in the wavenumber spectrum by ninety degrees.

**A wider peak**

In addition, the line of points for case 5 is shorter than the line of points for case 4 resulting in a wider peak in the rectified sin(x)/x shape for case 5.

**The real and imaginary parts**

While the real and imaginary parts of the spectrum shown in the center of Figure 10 are considerably different from anything that we have seen prior to this, they still satisfy the symmetry and asymmetry conditions that we expect for the real and imaginary parts.

**The final output matches the input**

The output from the inverse Fourier transform in the bottom left image in Figure 10 matches the input surface in the top left image in Figure 10.

All of this matches our expectations for this case.

**For switchCase = 6**

This case is considerably more complicated than the previous cases. You can view the surface for this case in the upper left image in Figure 11. You can view the code that generated this surface in Listing 22 near the end of the lesson.

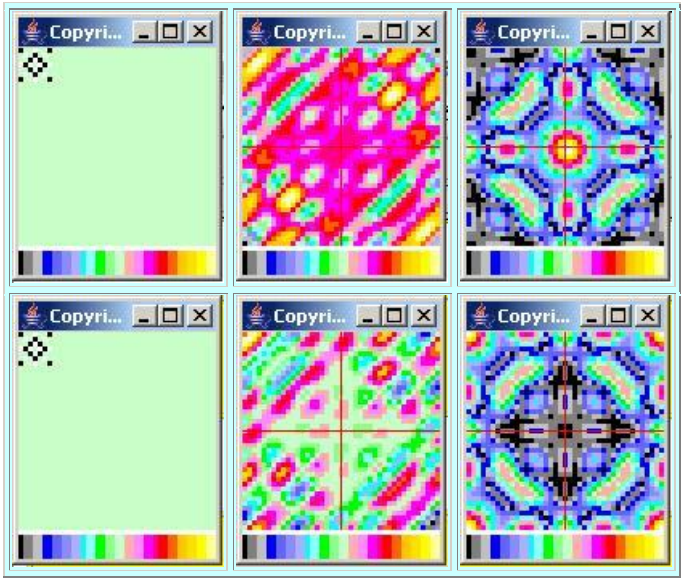



Figure 11

**Many weighted lines of points**

This case draws horizontal lines, vertical lines, and lines on both diagonals. Each individual point on each line is given a value of either +1 or -1. The weights of the individual points are adjusted so that the sum of all the weights is 0. The weight at the point where the lines intersect is also 0.

**Black is -1, white is +1**

The small black squares in the upper left image in Figure 11 represent points with a weight of -1. The small white squares represent points with a weight of +1. The green background color represents a value of 0.

**Symmetries on four different axes**

The wavenumber amplitude spectrum is shown in the bottom right image in Figure 11. As you can see from that image, performing a 2D Fourier transform on this surface produces a wavenumber amplitude spectrum that is symmetrical along lines drawn at 0, 45, 90, and 135 degrees to the horizontal. There is a line of symmetry in the amplitude spectrum for every line of points on the space domain surface.

**Must be zero at the wavenumber origin**

Because the sum of all the points is 0, the value of the wavenumber spectrum at the origin must also be zero. This is indicated by the black square at the origin in the lower right image.

**Peaks at the folding wave numbers**

This amplitude spectrum has major peaks at the folding wave number on each of the 45-degree axes. In addition, there are minor peaks at various other points in the spectrum.

**The real and imaginary parts**

As expected, the real and imaginary parts of the spectrum, shown in the center of Figure 11 exhibit the required symmetry and asymmetry that I discussed earlier.

**The final output**

The output produced by performing an inverse Fourier transform on the complex wavenumber spectrum is shown in the lower left image in Figure 11. This image matches the input surface shown in the top left image in Figure 11.

**For switchCase = 7**

Now we are going to make a major change in direction. All of the surfaces from cases 0 through 6 consisted of a few individual points located in specific geometries in the space domain. All of the remaining points on the surface had a value of zero. This resulted in continuous *(but sampled)* surfaces in the wavenumber domain.

Now we are going to generate continuous *(but sampled)* surfaces in the space domain. We will generate these surfaces as sinusoidal surfaces *(similar to a sheet of corrugated sheet metal)* or the sums of sinusoidal surfaces.

Performing Fourier transforms on these surfaces will produce amplitude spectra consisting of a few non-zero points in wavenumber space with the remaining points in the spectrum having values near zero.

## Need to change the surface plotting scale

In order to make these amplitude spectra easier to view, I have modified the program to cause the square representing each point in the amplitude spectrum to be five pixels on each side instead of three pixels on each side. To keep the overall size of the images under control, I reduced the width and the height of the surfaces from 41 points to 23 points.

## Display fewer results

I suspect that you have seen all the real parts, imaginary parts, and unshifted amplitude spectra that you want to see. Therefore, at this point, I will begin displaying only the input surface, the amplitude spectrum, and the output surface that results from performing an inverse Fourier transform on the complex spectrum.

## A zero frequency sine wave

The first example in this category is shown in Figure 12. The input surface for this example is a sinusoidal wave with a frequency of zero. This results in a perfectly flat surface in the space domain as shown in the leftmost image in Figure 12. This surface is perfectly flat and featureless.

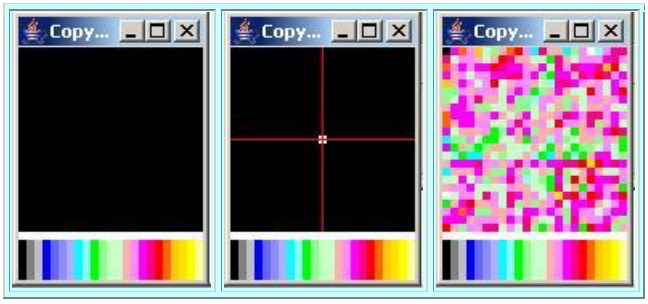



Figure 12

## The code for this case

The code that was used to generate this surface is shown in Listing 18. For the case of a sinusoidal wave with zero frequency, every point on the surface has a value of 1.0.

```
      case 7:
        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;

Listing 18
```

## A single point at the origin

As shown by the center image in Figure 12, the Fourier transform of this surface produces a single point at the origin in wavenumber space. This is exactly what we would expect.

## The inverse transform output is ugly

The result of performing an inverse Fourier transform on the complex spectrum is shown in the rightmost image in Figure 12. As was the case earlier in Figure 6, the ugliness of this plot is an artifact of the 3D plotting scheme implemented by the class named **ImgMod29**. The explanation that I gave there applies here also.

## A very small error

Once again, the total error is very small. The numeric output shows that the final output surface matches the input surface to within an error that is less than about one part in ten to the thirteenth power. Thus, the program produces the expected results for this test case.

## For switchCase = 8

This case draws a sinusoidal surface along the horizontal axis with one sample per cycle.

> *(This surface is under sampled by a factor of two under the commonly held belief that there should be at least two samples per cycle of the highest frequency component in the surface.)*

Thus, it is impossible to distinguish this surface from a surface consisting of a sinusoid with a frequency of zero.

The code that was used to produce this surface is shown in Listing 19. This code is typical of the code that I will be using to produce the remaining surfaces in this lesson. This code is straightforward and shouldn't require further explanation.

```
      case 8:
        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;

Listing 19
```

## The graphic output

The Fourier transform of this surface produces a single peak at the origin in the wavenumber spectrum just like in Figure 12. I didn't provide a display of the graphic output for this case because it looks just like the graphic output shown for the zero frequency sinusoid in Figure 12.

## For switchCase = 9

This case draws a sinusoidal surface along the horizontal axis with two samples per cycle as shown in the leftmost image in Figure 13. This is the Nyquist folding wavenumber.

## The wavenumber spectrum

The center image in Figure 13 shows the wavenumber amplitude spectrum for this surface. The wavenumber spectrum has white peak values at the positive and negative folding wave numbers. The colors in between these two peaks are green, blue, and gray indicating very low values.

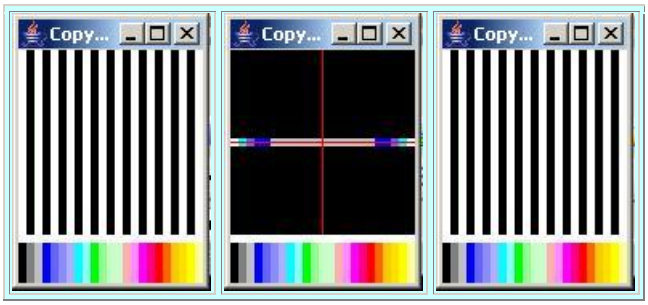



Figure 13

## The inverse Fourier transform output

The output from the inverse Fourier transform performed on the complex wavenumber spectrum for this case is shown in the rightmost image in Figure 13. The output is a good match for the input.

You can view the code that was used to create this surface in Listing 22 near the end of the lesson.

## For switchCase = 10

This case draws a sinusoidal surface along the vertical axis with two samples per cycle. Again, this is the Nyquist folding wave number but the sinusoid appears along the vertical axis instead of appearing along the horizontal axis. If you run this case and view the results, you will see that it replicates the results from case 9 except that everything is rotated by ninety degrees in both the space domain and the wavenumber domain.

**For switchCase = 11**

This case draws a sinusoidal surface along the horizontal axis with eight samples per cycle as shown in the leftmost image of Figure 14.

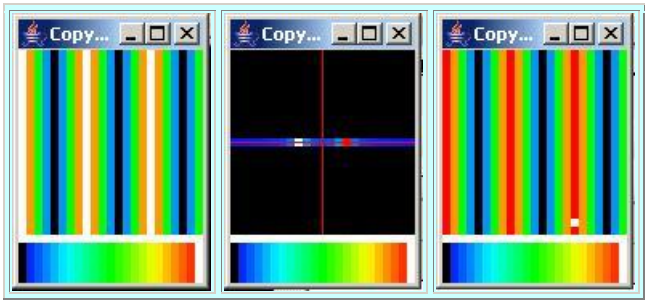



Figure 14

**The wavenumber spectrum**

Performing a forward Fourier transform on this surface produces symmetrical peaks on the horizontal axis on either side of the wavenumber origin. The two peaks are indicated by the small white and red squares on the horizontal axis in the center image in Figure 14.

> *(Recall that for the plotting format used in Figure 14, the color white is reserved for the single point with the highest elevation. The difference in an elevation colored white and an elevation colored red for this plotting format might be as small as one part in ten to the fourteenth or fifteenth power. As a practical matter, red and white indicate the same elevation for this plotting format.)*

For a sinusoidal surface with eight samples per cycle, we would expect the peaks to occur in the wavenumber spectrum about one-fourth of the distance from the origin to the folding wavenumber. Figure 14 meets that expectation.

The peaks are surrounded on both sides by blue and aqua colors, indicating very low values.

**The inverse Fourier transform output**

The output from the inverse Fourier transformed performed on the complex spectrum is shown in the rightmost image in Figure 14. This output compares very favorably with the input surface shown in the leftmost image. The difference between the two is that the input has white vertical

bands whereas the output has red vertical bands *(with a single white spot).* The above explanation of white versus red applies here also.

You can view the code that created this surface in Listing 22 near the end of the lesson.

**For switchCase = 12**

This case draws a sinusoidal surface on the horizontal axis with three samples per cycle plus a sinusoidal surface on the vertical axis with eight samples per cycle as shown by the leftmost image in Figure 15.

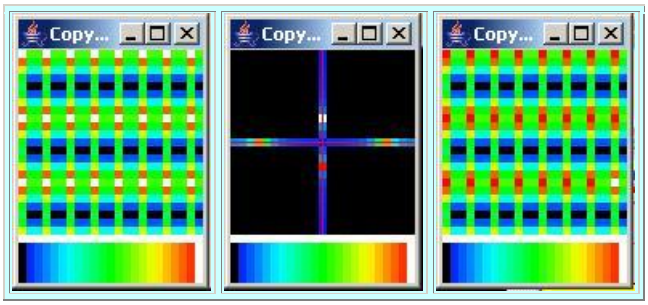



Figure 15

**The wavenumber spectrum**

Performing a forward Fourier transform produces symmetrical peaks on the horizontal and vertical axes on all four sides of the wave number origin. These peaks are indicated by the red and white squares in the center image in Figure 15.

> *(See the earlier discussion regarding the difference in elevation indicated by red and white for this plotting format.)*

The peaks on the vertical axis should be about one-fourth of the way between the origin and the folding wavenumber. This appears to be the case. The peaks on the horizontal axis should be about two-thirds of the way between the origin and the folding wavenumber, which they also appear to be.

**Inverse Fourier transform output**

The output produced by performing an inverse Fourier transform on the complex spectrum is shown in the rightmost image in Figure 15. Taking the red versus white issue into account, this output compares favorably with the input surface shown in the leftmost image in Figure 15.

You can view the code that created this surface in Listing 22 near the end of the lesson.

**For switchCase = 13**

This case draws a sinusoidal surface at an angle of approximately 45 degrees relative to the horizontal as shown in the leftmost image in Figure 16. This sinusoid has approximately eight samples per cycle.

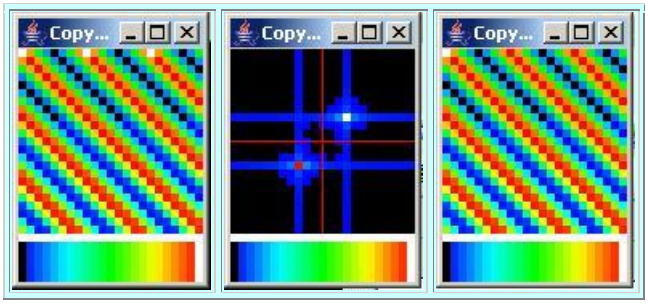



Figure 16

## The wavenumber spectrum

Performing a forward Fourier transform on this surface produces a pair of peaks in the wavenumber spectrum that are symmetrical about the origin at approximately 45 degrees relative to the horizontal axis. These peaks are indicated by the red and white squares in the center image in Figure 16.

## The inverse Fourier transform output

The output produced by performing an inverse Fourier transform on the complex wavenumber spectrum is shown in the rightmost image in Figure 16. This output compares favorably with the input surface shown in the leftmost image in Figure 16.

You can view the code that created this surface in Listing 22 near the end of the lesson.

## The end of the getSpatialData method

Listing 20 shows the end of the method named **getSpatialData** and the end of the class named **ImgMod31**.

```
      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 20
```

A default case is provided in the **switch** statement to deal with the possibility that the user may specify a case that is not included in the allowable limits of 0 through 13 inclusive.

The method ends by returning a reference to the array object containing the 3D surface that was created by the selected case in the **switch** statement.

# Run the Program

I encourage you to copy, compile, and run the programs that you will find in Listing 21 and Listing 22 near the end of the lesson.

> *(You will also need to go to the lesson entitled* Plotting 3D Surfaces using Java *and get a copy of the source code for the program named ImgMod29.)*

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 Part 1 of this lesson 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.

In Part 2, I provided and explained a 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 spectral data.

Finally, I provided and explained 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 relationships that exist between the space domain and the wavenumber domain

# What's Next?

I will explain general purpose 2D convolution and will explain and demonstrate the relationships that exist between 2D Fourier transforms and 2D convolution in the next lesson in this series.

# Complete Program Listings

Complete listings of the classes presented in this lesson are provided in Listing 21 and Listing 22 below.

```
/*File ImgMod30.java.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 21**

```
/*File ImgMod31.java.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 wavenumber 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 wavenumber domain with shifted origin. Compare with 5 below.
3. Imaginary wavenumber spectrum with shifted origin.
4. Real wavenumber spectrum with shifted origin.
5. Amplitude spectrum in wavenumber 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
0.20710678118654778 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 wavenumber 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 wavenumber 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 wavenumber 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 wavenumber 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  wavenumber 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
    // wavenumber 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 wavenumber 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 wavenumber 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 wavenumber
      // 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
          // wavenumbers 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 wavenumber
        // 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
        // wavenumber 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 22**

---



**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 forward inverse 2D 3D Fourier transform spectra antenna array complex spectral DSP surface Nyquist folding periodic

-end-