2D Fourier Series and Image Analysis

Mitchell Wasson - mwasson@comm.utoronto.ca; David Ding - davidy.ding@mail.utoronto.ca

Abstract
We have previously seen some uses of the Fourier series. These uses were shown in a one-dimensional setting where the independent variable is time. This lab is an extension of Fourier series to two dimensions, and introduces its application to image compression.

Keywords
Discrete — Fourier — Series — Dimension — Image — Analysis

Contents

Introduction

The two-dimensional discrete-time Fourier series is very similar to the one-dimensional discrete-time Fourier series. Once we understand the Fourier series as a projection of a signal onto the Fourier basis, this will be apparent. **Prior to** this lab, please read sections **6.4.2 – 6.4.5 (p. 82-89)** of your ECE216 course notes to gain insight into this Fourier series extension into 2D and do refer to these pages throughout this lab.

We start by reviewing the one-dimensional discrete-time Fourier series as a projection onto an orthogonal basis. We then extend this basis interpretation to two dimensions, producing the 2D discrete-time Fourier series. Possible applications of this two-dimensional transformation are explored briefly in the context of image compression. In the process, you will gain a glimpse into the area of multimedia communications, an area of engineering concerned with the transmission of media content.

Background

Review

Discrete-Time Fourier Series (DTFS)

We start by recalling the one-dimensional discrete-time Fourier series. Consider one period of a discrete-time signal, $x : \{0,1,\ldots,N_0-1\} \to \mathbb{C}$. The Fourier series coefficients (a_k) are given by

$$a_k = \sum_{n=0}^{N_0-1} x[n] e^{-j\frac{2\pi}{N_0}kn} \tag{1}$$

Additionally, the original signal can be obtained from its Fourier series as

$$x[n] = \frac{1}{N_0} \sum_{k=0}^{N_0-1} a_k e^{j\frac{2\pi}{N}kn} \tag{2}$$

Note: In the lecture and the course notes, the factor of $1/N_0$ is used in **the analysis equation**, whereas here it is used in **the synthesis equation** in (2). This is because Matlab uses the convention described in this lab. Please keep this in mind as you do the labs and follow all equations as given in this lab.

Basis Projection Interpretation

We wish to interpret the discrete-time Fourier series as a projection onto a basis. A basis for all discrete-time signals is a set of linearly independent signals that span the entire space. In other words, any discrete-time signal can be written as a linear combination of basis vectors.

Perhaps the most familiar basis for discrete-time signals is the standard basis, given by the discrete-time signals:

$$\delta_k[n] = \begin{cases} 1 & \text{if } n = k \\ 0 & \text{if } n \neq k \end{cases} \quad k,n \in \{0,1,\ldots,N_0-1\} \tag{3}$$

Another useful basis is the Fourier basis, comprised of the signals $\phi_k[n]$ where

$$\phi_k[n] = e^{j\frac{2\pi}{N_0}kn} \quad k,n \in \{0,1,\ldots,N_0-1\} \tag{4}$$

The Fourier series coefficients of a signal are found by projecting the signal onto the Fourier basis vectors. With this perspective, equation (1) can be expressed as the inner product of x and ϕ_k. Recall that $z^* \in \mathbb{C}$ is the complex conjugate of $z \in \mathbb{C}$. The DTFS coefficient then becomes

$$a_k = < x, \phi_k > = \sum_{n=0}^{N-1} x[n]\phi_k[n]^* = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn} \tag{5}$$

Since the complex exponential signals ϕ_k form a basis (hence the name "Fourier basis"), we can reconstruct the original signal as a linear combination of the Fourier basis vectors, appropriately weighted by the respective DTFS coefficients.

$$x[n] = \frac{1}{N_0} \sum_{k=0}^{N-1} a_k \phi_k[n] = \frac{1}{N_0} \sum_{k=0}^{N-1} a_k e^{j\frac{2\pi}{N}kn} \tag{6}$$

With this basis interpretation, we can see the the reason for "$e^{-j\frac{2\pi}{N}kn}$" in equation (5) instead of "$e^{j\frac{2\pi}{N}kn}$" is that the definition of the inner product involves a conjugate.

2D Fourier Series

We will now use a similar basis projection to define the 2D DTFS. In order to do this, we first define the 2D Fourier basis. As you might expect, these basis functions are complex exponentials. The difference is that they are functions of two variables instead of one. Necessarily, they must also have two frequency variables (one for each dimension).

We consider 2D discrete-time signals of the form $x : \{0,1,\ldots,N-1\} \times \{0,1,\ldots,M-1\} \to \mathbb{C}$. The 2D Fourier basis vector $\phi_{k,l}$ is defined by:

$$\phi_{k,l}[n,m] = e^{2\pi j\left(\frac{kn}{N}+\frac{lm}{M}\right)} \quad k,n \in \{0,1,\ldots,N-1\}, \quad l,m \in \{0,1,\ldots,M-1\} \tag{7}$$

A physical example of such signals is an N pixels $\times$ M pixels image, which the computer stores as a series of pixels.

Similar to the 1D case, we can obtain the 2D DTFS via projection onto the Fourier basis.

$$a_{k,l} = < x, \phi_{k,l} > = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} x[n,m]\phi_{k,l}[n,m]^* = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} x[n,m]e^{-2\pi j\left(\frac{kn}{N}+\frac{lm}{M}\right)} \tag{8}$$

For each $l \in \{0,1,\ldots,M-1\}$ and $k \in \{0,1,\ldots,N-1\}$.

We can also re-obtain the original signal from the above projection as a linear combination of the Fourier basis.

$$x[n,m] = \frac{1}{NM}\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} a_{k,l}\phi_{k,l}[n,m] = \frac{1}{NM}\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} a_{k,l}e^{2\pi j\left(\frac{kn}{N}+\frac{lm}{M}\right)} \tag{9}$$

For each $m \in \{0,1,\ldots,M-1\}$ and $n \in \{0,1,\ldots,N-1\}$.

Again, please note that the factor of $1/MN$ in theory is to be present in the analysis equation and not the synthesis equation, but Matlab switches them around, just like the 1D case.

2D Discrete-Time Signals as Matrices

Recall that one-dimensional discrete-time periodic signals can be represented by vectors. i.e. $x : \{0,1,\ldots,N_0-1\} \to \mathbb{C}$ is represented by $[x[0]\,x[1]\cdots x[N_0-1]] \in \mathbb{C}^{N_0}$. Similarly, two-dimensional discrete-time periodic signals can be represented with matrices. Therefore the signal $x : \{0,1,\ldots,N-1\} \times \{0,1,\ldots,M-1\} \to \mathbb{C}$ is represented by a matrix of the form

$$\begin{bmatrix} x[0,0] & x[0,1] & \cdots & x[0,M-1] \\ x[1,0] & x[1,1] & \cdots & x[1,M-1] \\ \vdots & \vdots & \ddots & \vdots \\ x[N-1,0] & x[N-1,1] & \cdots & x[N-1,M-1] \end{bmatrix} \in \mathbb{C}^{N\times M}.$$

This is a another powerful shift in perspective. By using Matlab's vector environment, it will allow us to perform some of the Fourier series operations outlined above on entire signals instead of executing them value by value. For example, we can express equation (9) in terms of entire signals when using matrix representations.

$$x = \frac{1}{NM}\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} a_{k,l}\phi_{k,l} \tag{10}$$

Please note here that $a_{l,k}$ is a scalar and $\phi_{l,k}$ is a matrix. Also, x is a matrix as well.

1. 2D Fourier Basis Vectors

1.1 Generating the Basis Vectors

In this section of the experiment, you will complete the implementation of a Matlab function that generates 2D Fourier basis functions as matrices. Look at the provided Matlab function in `FourierBasisVector2D.m`. It has the following function header:

```
function [basisVector] = FourierBasisVector2D(k, l, N, M)
```

As you can see, the function inputs consist of the two frequency parameters (l and k) and two parameters specifying the signal size (N and M). The function should return an $N \times M$ matrix (N rows and M columns) that corresponds to the 2D Fourier basis vector with frequency parameters k and l as specified in equation (7). For convenience, we refer to these 2D Fourier basis vectors as **basis matrices**. It should be noted that Matlab's array indexing starts at 1 instead of 0. You will have to account for this when making your function.

Hint: Matlab constant for π is `pi`. Matlab constant for the imaginary number j is `1i`.

1.2 Evaluation and Experimentation

Once completed, run the script titled `Part1.m`. It will display the real part of the Fourier basis vector that is generated by a call to your function, as a surface plot and as an image. The following figures show a correct output with inputs:
`k=1; l=2; N=300; M=400;`

Figure 1. Example Surface Plot of 2D Complex Exponential

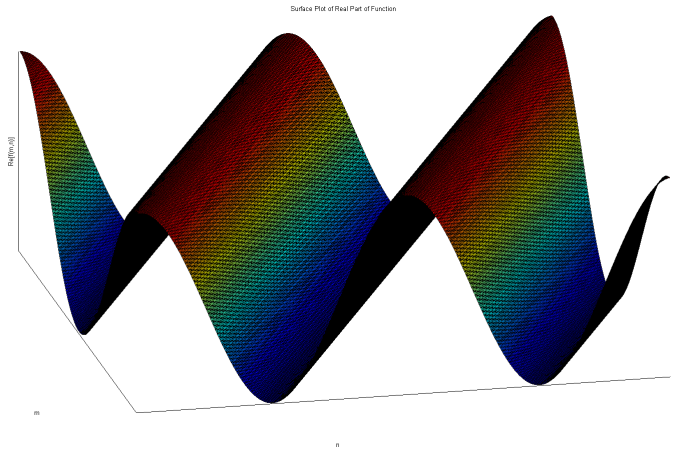
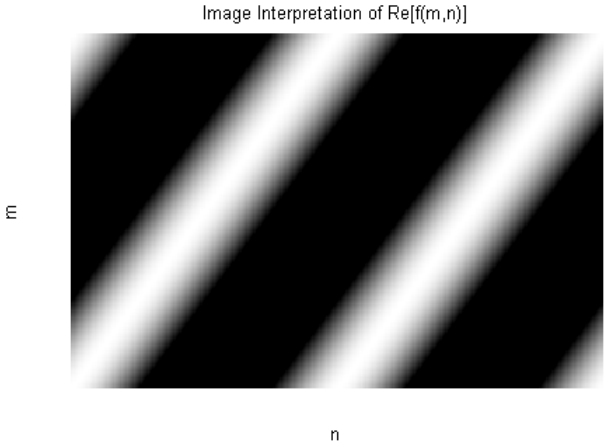


Figure 2. Example Image of 2D Complex Exponential



When you have the correct implementation, look at these images carefully. Figure 1 shows a 3D plot of the basis function (real part) with the two independent variables n and m, which interestingly looks like a sinusoid. This should not be surprising as the basis function, according to (7), is just a 2D extension of the 1D complex sinusoid. Figure 2 shows the basis in the context of an image: the higher the magnitude of the basis for a particular (n,m), the "whiter" the pixel at (n,m) becomes.

As you change k and l, you get a different basis function that corresponds to the frequency pair at (k,l). In `Part1.m` script, feel free to change the values of k and l as you wish to examine different basis matrices. When you plot the figures by running the script, notice changes in the sinusoid plotted as well as the image represented by the basis function. Does the sinusoid appear to have higher frequencies for different values of k and l?

When you show your TA the working implementation, answer these questions based on the plots that you produced and by playing around with k and l.

(a) Which values of k and l produce the lowest frequency complex exponentials?

(b) Which values of k and l produce the highest frequency complex exponentials?

Things to keep in mind:

- Just like the 1D complex sinuoid for forming the basis of 1D Fourier series, the complex sinusoid $e^{2\pi j \frac{kn}{N_0}}$ is periodic in k with period N_0 if the signal is periodic in n with period N_0. Thus, $k = N_0 + 1$ is the same frequency as $k = 1$. Please keep this in mind when you answer the questions about 2D Fourier series. The exact same concept applies, except you have to keep track of two dimensions!

- If you are stuck, try fixing, say the value of k and see how the basis functions change with l. Think about which values of k for the **1D** basis functions yield the highest frequencies and which yields the lowest frequencies.

- This might be common-sense but worth mentioning: if you don't see any changes in the surf plots and image plots, it might mean that you are looking at zero frequency signals.

At this point, show your TA that your implementation functions properly and answer the above questions.

2. 2D Fourier Series

2.1 DTFS Development

In this section you will build upon the function you created in Part 1 in order to create a two-dimensional DTFS function. The function has the following header:

```
function [frequencyMatrix] = DiscreteTimeFourierSeries2D(timeMatrix)
```

The input to the function, `timeMatrix`, is the matrix you wish to take the 2D DTFS. The output of the function, `frequencyMatrix`, is the matrix of DTFS coefficients, containing all the projections of the input onto the 2D Fourier basis. This is an $N \times M$ matrix as well. You should produce the output by referring to equation (8).

Hint: First, note that you need to make use of the `FourierBasisVector2D.m` function created in part 1. Next, if you are not familiar with vectorization in Matlab (the .* operator), you would need to have four levels of for-loops to build the 2D Fourier series coefficients matrix. The outer two levels are already set up for you in the skeleton code, which corresponds to the frequency coefficient at the $(k,l)^{th}$ entry. Remember that Matlab begins indexing at 1.

2.2 Evaluation and Experimentation

Once completed, run the script titled `Part2.m` to test your functions. Check the transform error value that is printed to the console. If it is zero (within floating point round-off error) then your functions work properly.

`Part2.m` tests your transforms on the 2D equivalent of the rectangle function. Visual representations of this function are shown in Figures 3 and 4.

Figure 3. Example Surface Plot of 2D Rectangle Function

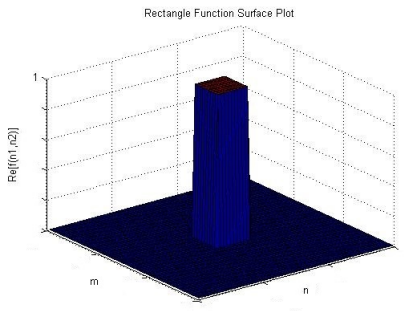
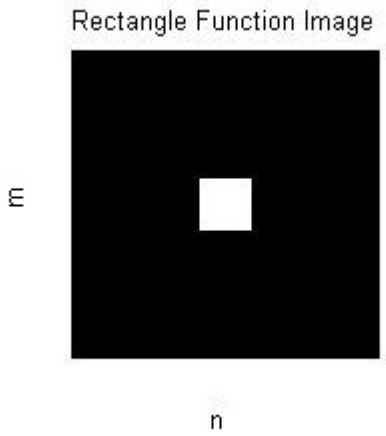


Figure 4. Example Image of 2D Rectangle Function



You need to answer the question: "How does the 2D square pulse compare to the 1D square pulse with regard to the Fourier series?". To help you on this, recall that the signal shown in figures 3 and 4 is the 2D analog of the 1D rectangular pulse studies in class. The 1D rectangular pulse would have something like:

$$rect(n) = \begin{cases} 1, & -N/2 \leq n \leq N/2 \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

Think about what the general shape of the Fourier transform of the 1D rectangular pulse looks like. Then look at the 2D Fourier series coefficient surface plot that you created. How does the 2D square pulse transform compare with the 1D pulse? Feel free to drag around the plot to get different points of view to help answer the question.

2.3 Fast Fourier Transform (FFT)

Now that you have a good understanding of the two-dimensional DTFS, you will use Matlab's implementation. The two functions of interest are `fft2()` and `ifft2()` corresponding to the 2D DTFS and its inverse. These functions are based on an algorithm called the fast Fourier transform (FFT). This algorithm gives an extremely efficient implementation of the DTFS. If you are interested in understanding the FFT, you may read upon it further in your own time.

Run `Part2.m` again. In addition to the transform error, it first prints the time elapsed executing your 2D DTFS function. Then it prints the elapsed time running the 2D fast Fourier transform.

When you have completed this section, consider the following question.

- How many times faster is the 2D FFT than your implementation?

At this point, show your TA that your implementation functions properly and answer the above questions.

3. Image Compression

In this section you will develop a simple image compression system based on the 2D Fourier transform. Before doing so, we will introduce a few concepts from data compression. The first of which is peak signal-to-noise ratio.

3.1 Peak Signal-to-Noise Ratio

The PSNR is the ratio of the maximum possible power of a signal component to the mean squared error introduced to the signal during compression. The maximum power is the square of the highest possible value that a signal can take on. The log of this ratio is measured in decibels. We start be defining the mean squared error between a 2D function $x \in \mathbb{C}^{N\times M}$ and its compressed version $\hat{x} \in \mathbb{C}^{N\times M}$.

$$e_{av}^2 = \frac{1}{NM} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} |\hat{x}[n,m] - x[n,m]|^2 \tag{12}$$

The peak signal-to-noise ratio is then defined as:

$$PSNR = 10\log_{10}\frac{P_{max}}{e_{av}^2} \tag{13}$$

PSNR is widely used as an objective measure of compressed image quality.

3.2 Compression Rate

The second concept we introduce is compression rate. It measures the average number of compressed bits it takes to store an uncompressed bit, simply defined as:

$$R = \frac{\#\,of\ bits\ used\ to\ store\ \hat{x}}{\#\,of\ bits\ used\ to\ store\ x} \quad \text{(hopefully less or equal to 1)} \tag{14}$$

Now that we have the basic measures needed to evaluate a compression system, you will move forward with the creation of one. Run the `CompressionExample.m` script. This script shows the process of applying the inverse 2D DTFS in two different manners. The first adds the weighted complex exponentials that form the image together in no particular order. The second sorts the Fourier series coefficients in order of descending absolute value and adds complex exponentials together in this order. We can see in the PSNR graph and image comparisons that the second method produces an image closer to the original using a fixed number of Fourier basis vectors. This observation is the motivation for the following compression scheme that you will evaluate.

3.3 Compressor Development

You will develop an image compression function with the following header:

```
function [numDiscardedCoefficients, compressedTestSignal]
 = compressor(testSignal, cutoff)
```

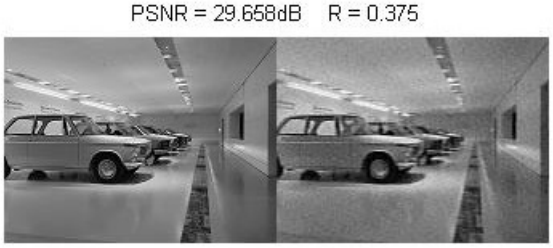
The function will take in an image as a matrix in $\mathbb{C}^{M\times N}$. The 2D DTFS of the image will then be taken. Your function will then set any Fourier basis coefficients with modulus **less** than `cutoff` to zero. The inverse 2D DTFS of this matrix should then be taken and returned in the `compressedTestSignal` variable. i.e. You are only keeping the Fourier basis vectors that correspond to high energy portions of the image's spectrum. You should also keep track of the number of coefficients that you zero out and return that number in the `numDiscardedCoefficients` variable (to allow for the compression rate to be calculated).

Compression can be achieved in this manner by storing the Fourier basis coefficients of the image instead of the actual image. In practice, the zeroed out coefficients are not stored, since their values are known. instead, the **locations** of theses excluded coefficients are stored. Storing the locations, which are integer-valued, will take up far less memory compared to storing the coefficients themselves.

3.4 Evaluation and Experimentation

Once you have implemented `compressor`, run the script titled `Part3.m`. This script will provide a test image to your function and compare the original image to its compressed version. The script will display a figure similar to Figure 5, showing the original image, the compressed image, the PSNR of the compression, and the compression rate.

Figure 5. Example of Compressed Image



If you look at the `calculateCompressionRate.m` function, you will see exactly how the compression rate is calculated. The function assumes its takes 128 bits to store a Fourier basis coefficient (double precision each for the real and imaginary

parts). In order to store the location of a discarded coefficient we need $\log_2(NM)$ bits, and finally we need to store the total number of discarded coefficients with $\log_2(NM)$ bits. This is a crude approximation of the compression rate, but is sufficient to demonstrate the trade-off between PSNR and rate.

Example: Say you have a 2pix by 2pix grayscale image. Each pixel takes 128 bits to store so the total number of bits to store the original image is 128*2*2. Now, for image compression, you take the 2D Fourier series of this small image to acquire four Fourier series coefficients, two along each dimension. Each coefficient also takes 128 bits to store due to double precision. Now, say you discarded two coefficients because their magnitudes are below the cutoff. The number of bits to store the remaining coefficients would be 128*(4-2) = 128*2. However, this is not enough as the computer needs to know which coefficients were zeroed out. Note that you need log2(4) = 2 bits to store each location of the thrown-away coefficients, since there are four possible combinations of (l,k) and each bit conveys two possibilities. Thus, the total number of "location" bits required is 2*log2(4). Finally, the computer needs to know how many coefficients were discarded, and the most number of bits you need is log2(4) since up to 4 pixels can be discarded.

This amounts to 128*(4-2) + 2*log2(4) + log2(4) = 262 bits for the compressed image, versus 128*4 = 512 bits needed to store the original image, which amounts to a compression rate R = 262/512 = 0.518. **End of Example**

Experiment with various values of the `cutoff` variable in `Part3.m` in order to throw away fewer or more coefficients. Observe the effects on the image, PSNR, and compression rate.

When you think you have a correct implementation, consider the following questions.

(a) Is there a rate that you cannot get below no matter how high you set `cutoff`? What we are looking for here is for you to run `Part3.m` script, but you can change the values of `cutoff` from the default value of 10 to something else, and observe the changes in the quality of the compressed image as well as the compression rate. As `cutoff` $\to \infty$, R approaches to what value?

(b) Why does this occur? Can you think of a modification to this compression scheme that would fix this issue?

At this point, show your TA that your implementation functions properly and answer the above questions.

Conclusion

As this course moves forward, you will be exposed to more advanced concepts related to the filtering of 1-D signals. You are encouraged to think about what the multidimensional equivalent of these concepts would look like. In fact, the 2D and 3D application of the concepts you will learn are the main tools used to construct multimedia tools such as a photo or video editing software.