MATH 52: MATLAB HOMEWORK 1

1. Approximating functions

A typical method for understanding complicated mathematical objects is to attempt to approxmate them as limits of relatively simple objects. The prototypical example is to approximate an arbitrary function $f : \mathbb{R} \to \mathbb{R}$ as a linear combination of the monomials $1, x, x^2, \cdots$. In this case the nth order approximation of $f(x)$ is of the form

$$c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n = \sum_{j=0}^{n} c_j x^j$$

where c_j are real numbers that depend on the function f. Unless f is a polynomial, there is no n so that the nth order approximation of $f(x)$ is identically $f(x)$, so instead we hope that the limit of nth order approximations converge to $f(x)$. Expressed as an equation our hope is

$$f(x) = \lim_{n\to\infty} \sum_{j=0}^{n} c_j x^j .$$

There is a great deal which needs to be made precise in a statement such as this; for example, what does this limit mean, for what values of x can we expect this equality to hold. For now we content ourselves with asking how to find the coefficients c_j in a systematic way. Provided $f(x)$ has the appropriate derivatives, one answer to this question is given by the theory of Taylor series. For points x which are 'sufficiently close to' 0, we have

$$f(x) = f(0) + f'(0)x + \frac{f''(x)}{2}x^2 + \cdots + \frac{f^n(0)}{n!}x^n + \cdots .$$

Notice that to find the coefficients c_j we need to know information about the derivatives of $f(x)$ at $x = 0$.

Example.

Let $f(x) = e^x$, and recall that $f^{(n)}(x) = e^x$ for all n. Hence $f^{(n)}(0) = 1$ for any n, and so the Taylor series of e^x begins

$$e^x = 1 + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \frac{1}{120}x^5 + \cdots .$$

MatLab also has a built in feature for computing the Taylor series of a function: `taylor`. For instance, if we wanted the first 10 terms of the Taylor Series for e^x instead of the first 6 (which we just listed), we use the following MatLab command.

```
>> syms x
>> taylor('exp(x)',x,10)
```

The function `taylor` has three arguments: the first is the function (notice it's within single quotes), the second is the variable, and the third is the number of terms in the series you would like. Notice also that we had to begin by declaring x as a variable using the `syms` command. If you don't do this, MatLab will have a fit!

Exercise. Compute the first seven terms of the Taylor series for $f(x) = -\log(1-x)$. Compute the first seven terms of the Taylor series for $g(x) = e^x/cos(x) + tan^3(x)$.

2. Approximating periodic functions: Fourier series

While the monomials $\{x^j\}$ are a useful collection of "basic functions" for approximating arbitrary functions, in certain contexts it can be useful to choose a different collection of "basic functions." For instance, if you wish to study all functions $f : \mathbb{R} \to \mathbb{R}$ which satisfy $f(x) = f(x + 2\pi)$ (i.e., functions which are periodic with period dividing 2π), it is typical to use the functions

$$\cdots, e^{-3ix}, e^{-2ix}, e^{-ix}, 1, e^{ix}, e^{2ix}, e^{3ix}, \cdots.$$

Here i is the complex number $\sqrt{-1}$. Though it might seem bizarre to use complex functions to approximate functions $f : \mathbb{R} \to \mathbb{R}$, it is an old but important result of that these functions are "enough" to express all such periodic functions, in the sense that for any function $f(x)$ satisfying $f(x) = f(x + 2\pi)$ there are real numbers $\cdots, a_{-3}, a_{-2}, a_{-1}, a_0, a_1, a_2, a_3, \cdots$ so that

$$f(x) = \lim_{n\to\infty} \left[\sum_{j=-n}^{n} a_j e^{ijx} \right].$$

This is called the Fourier series expansion of $f(x)$. If you like, you can think of $\sum_{j=-n}^{n} a_j e^{ijx}$ as the nth order approximation of $f(x)$. The coefficients $\{a_i\}$ are called the complex Fourier coefficients of the function $f(x)$.

Example.

The function

$$f(x) = 2\left(\frac{x+\pi}{2\pi} - \left\lfloor \left(\frac{x+\pi}{2\pi}\right) \right\rfloor \right) - 1$$

is shown in blue in Figure 1. On the same axes we show the first, second, third, fourth and fifth order approximations of $f(x)$ (colored green, red, teal, purple, and yellow, respectively) using the complex Fourier coefficients

$$a_0 = 0, a_{\pm 1} = -\frac{i}{\pi}, a_{\pm 2} = \frac{i}{2\pi}, a_{\pm 3} = -\frac{i}{3\pi}, a_{\pm 4} = \frac{i}{4\pi}, \text{ and } a_{\pm 5} = -\frac{i}{5\pi}.$$

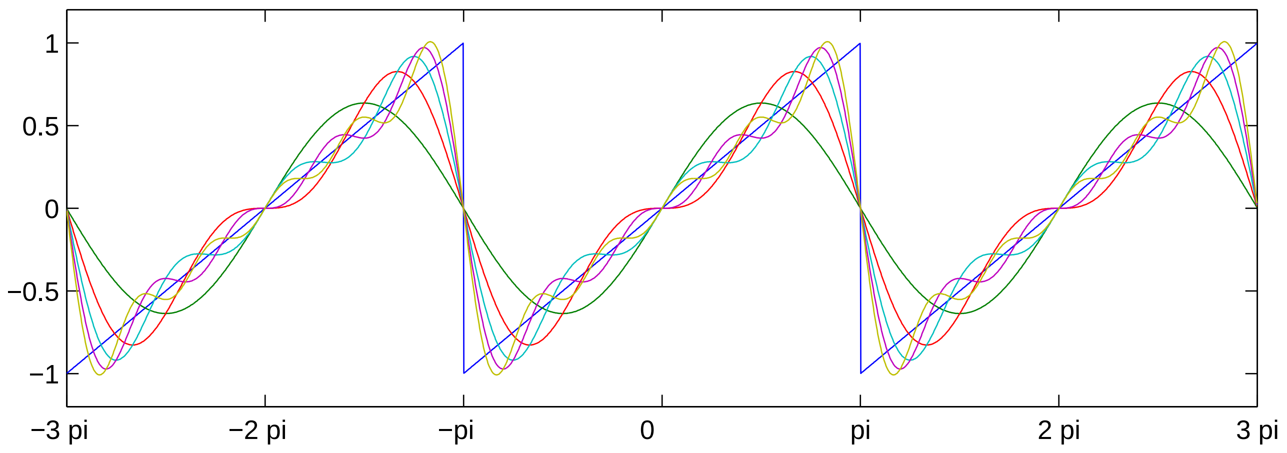

Figure 1. The function $f(x)$ and the Fourier approximations

As with Taylor series there are natural questions to ask about the Fourier series of a function $f(x)$, and again we will limit our discussion to how one computes the corresponding complex Fourier coefficients. Happily, just as with Taylor series, the theory of Fourier series comes equipped with an easy method for computing these coefficients; but while Taylor series use derivatives, one uses integrals to compute Fourier coefficients. Specifically,

$$a_j = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)e^{-jix}\,dx.$$

Example.

Notice that in the example above, the function $f(x)$ is equivalent to $g(x) = \frac{x}{\pi}$ on the interval $[-\pi, \pi]$. To find the complex Fourier coefficient a_1 for $f(x)$, one may run `int` function on MatLab.

```
>> clear x; syms x;
>> int((x/pi)*exp(-i*x),x,-pi,pi)/(2*pi)
ans =

-i*pi
```

The `int` command on MatLab takes four arguments: the first is the function to be integrated, the second the variable of integration, the third the left hand endpoint of the interval of integration, and the final argument is the right hand endpoint of the interval of integration.

Notice that this answer is the coefficient that we used in the previous example. To find the remaining coefficients, one can compute similar integrals to the one above. For instance, to find the coefficient a_{-2} one computes

```
>> clear x; syms x;
>> int((x/pi)*exp(-i*-2*x),x,-pi,pi)/(2*pi)
```

The previous example illustrates an important idea in the application of Fourier series: one only needs to know the behavior of a function on the interval $[-\pi, \pi]$ in order to computer Fourier coefficients. Indeed this idea can be used to give a Fourier decomposition of a function on the domain $[-\pi, \pi]$–even if the function is not periodic. In doing this, one is essentially making a periodic function out of the data of the function $f(x)$ on the interval $[-\pi, \pi]$.

Exercise. Compute the complex Fourier coefficients $a_{-3}, \cdots, a_3$ for the function defined by $f(x) = x^2 - 4$ on $[-\pi, \pi]$ and which is 2π periodic (i.e., $f(x) = f(x + 2\pi)$ for all x).

3. From the continuous to the discrete: Discrete Fourier series

The machinery developed so far has allowed us to write functions $f : \mathbb{R} \to \mathbb{R}$ as sums of basic functions. These ideas extend to other classes of functions as well. In particular, it is often convenient to work with functions whose domains are discrete instead of continuous. For the purposes of this problem set, a function with discrete domain will simply be a function whose domain is a subset of the integers. For simplicity, we will write $\mathbb{Z}$ to stand for the set of integers: $\mathbb{Z} := \{\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots\}$.

Example.

The function $f : \mathbb{Z} \to \mathbb{R}$ given by $f(x) = x^2$ is a discrete-domain function, while the function $g : \mathbb{R} \to \mathbb{R}$ given by $g(x) = x^2$ is not (see Figure 2).

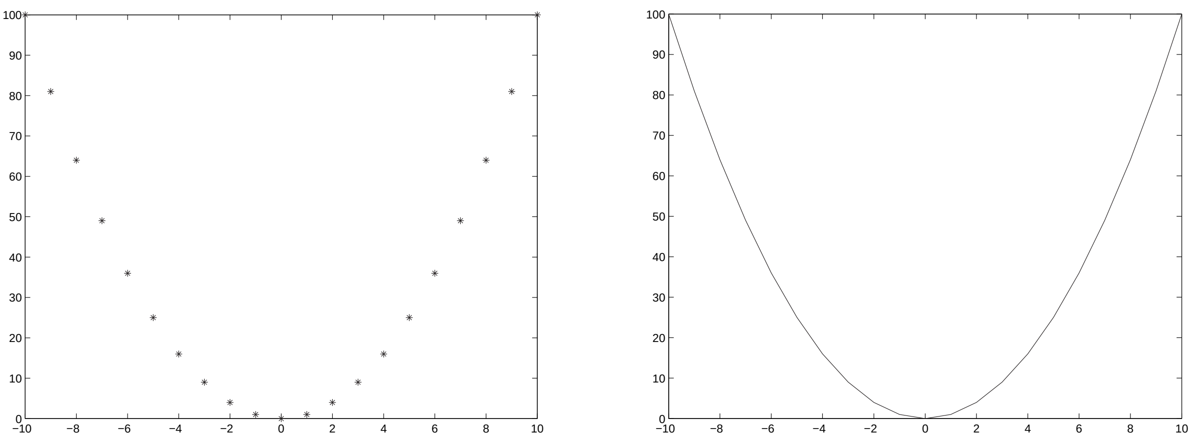

Figure 2. $f(x)$ has a discrete domain, $g(x)$ does not

Discrete-domain functions appear everyone in your daily life. For instance, the sound produced by a major C chord on a piano is be represented by the function whose graph is seen on the left of Figure 3. Digitizing this sound so that it can be recorded onto CD, however, produces encoded information will look instead like the right of Figure 3.

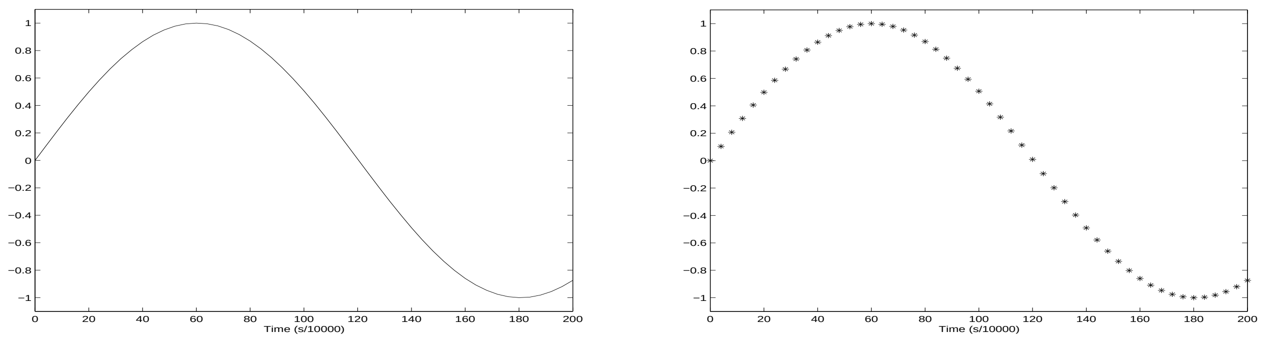

Figure 3. Continuous (real) sound versus discrete (digitized) sound

This example is a typical use of discrete domain functions: in science and technology, 'real-world' data is sampled at distinct time intervals, and the sampled data is used to model the observed phenomenon. In general one hopes to observe enough data so that a human observer cannot distinguish between the actual (continuous) phenomenon and the encoded (discrete) representation; for encoding sound, this means sampling at least 48,000 times per second.

Now suppose that $f : \mathbb{Z} \to \mathbb{R}$ is a discrete domain function. Suppose that it is also m-periodic, meaning that m is an integer with the property that for any other integer x, $f(x+m) = f(x)$. Just as in the continuous-domain case, $f(x)$ can be written as a sum of basic functions. The difference in this case, however, is that our basic functions take the form $e^{i2\pi\frac{j}{m}x}$ for values of j ranging between 1 and m. Hence we look to find coefficients b_j with

$$f(x) = \sum_{j=1}^{m} b_j e^{i2\pi\frac{j}{m}x}. \tag{1}$$

Since our function is discrete, this is called the discrete Fourier series of $f(x)$. Notice that here our function $f(x)$ is expressable as a *finite* series; in fact, the discrete Fourier series has precisely m terms, where m is the period of the function $f(x)$. To compute the coefficients b_n, we again compute an integral...of sorts. Specifically,

$$b_j = \frac{1}{m}\sum_{x=1}^{m} f(x)e^{-i2\pi\frac{j}{m}x}.$$

Since our function has a domain which is discrete instead of continuous, the integration which we performed in the continuous case can be computed using summation.

Example.

Consider the (discrete-domain) function $f(x)$ which satisfies $f(x) = f(x+6)$ and so that $f(0) = f(2) = f(4) = 0$ and $f(1) = f(3) = f(5) = 1$. Then we compute the coefficient b_2 as follows.

```
>> (1/6)*(1*exp(-i*2*pi*2/6*1)+1*exp(-i*2*pi*2/6*3)+1*exp(-i*2*pi*2/6*5))
ans =

 -1.8504e-016 -1.1102e-016i
```

This means that $b_2 = -1.8504 \times 10^{-16} + -1.1102 \times 10^{-16}i$. These very small numbers are probably a result of round-off error from MatLab's calculations, and so $b_2 = 0$.

Exercise. Compute the coefficients b_0, b_1, b_3, b_5 for the function $f(x)$ in the previous example. Then use Equation 1 to compute $f(0), f(1), f(2), f(3), f(4), f(5)$ and $f(6)$.

4. Digital images and DFT in 2 dimensions

Just as sound waves are digitized for manipulation or storage on a computer or CD, images also undergo a digitization process in order to appear on your laptop. To simplify the description of this proceducre, we will assume that the image we are interested in digitizing is a black and white photo.

The idea is relatively simple: the image is divided into small squares called pixels. In each pixel, the average darkness of the picture is measured. Typically, a pixel which is perfectly white will be given a darkness rating of 0, and a pixel which is perfectly black will be given a rating of 255. Shades of gray between white and black receive integer ratings between 1 and 254, with darker grays receiving higher ratings. Once the darkness of each pixel is measured, these measurements are stored in a matrix. It is typical that an image which is m pixels wide and n pixels tall be stored in an $n \times m$ matrix A whose entry in the ith row and jth column gives the darkness of the pixel in the ith row and jth column of the segmented image. We will call $a_{i,j}$ the entry in the ith row and jth column of A.

Example.

In the picture below we have magnified a picture which consists of only 4 pixels (arranged in a square). We give the darkness ratings for each of these pixels.

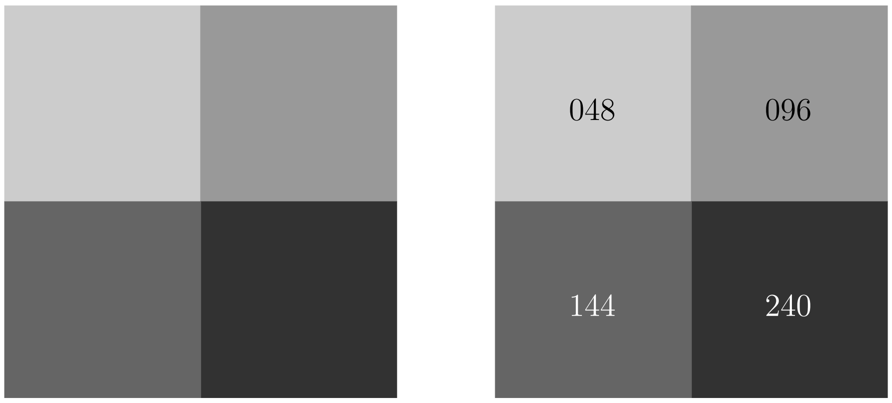

Figure 4. Image Pixelation

The matrix which corresponds to this image is

$$A = \begin{pmatrix} 48 & 96 \\ 144 & 240 \end{pmatrix}.$$

Using the integers 0 to 255 to represent darkness is not a hard and fast rule, as different file types or image displayers might have different conventions. For instance, it is also typical to represent the darkness of a pixel by a decimal number between 0 and 1. MatLab can display images with either convention, and we will use this ability (implicitly) for some of our exercises.

An $n \times m$ image can be thought of as a discrete-domain function $f(x, y)$ in 2 dimensions defined by $f(x, y) = a_{x,y}$ as values of x run from 1 to m and values of y run from 1 to n. With this in mind, one can attempt to decompose an image into its constituent 'frequencies' by finding coefficients $c_{k,l}$ with

$$f(x,y) = \sum_{k=1}^{n}\sum_{l=1}^{m} c_{k,l} e^{i\left(\frac{2\pi k x}{m} + \frac{2\pi l y}{n}\right)}. \tag{2}$$

This is the analogue of the Fourier transform we performed in the previous section, except instead of dealing with discrete-domain functions of 1 variable, we now have a discrete-domain function of 2 variables. Just as in the one variable case, these coefficients are computed using 'discrete integrals' (i.e., sums):

$$c_{k,l} = \frac{1}{nm}\sum_{x=1}^{m}\sum_{y=1}^{n} f(x,y) e^{-i\left(\frac{2\pi k x}{m} + \frac{2\pi l y}{n}\right)}. \tag{3}$$

Our goal is to use these discrete Fourier series in two variables to manipulate images.

Example.

Let's do a slightly more complicated example. MatLab will allow you to import images from the web in certain standard image formats. The image `clock.jpg` can be found on the course webpage; download it onto your computer by right-clicking the image and selecting 'Save Image As...' Be sure to remember where you saved the image (we put ours on the desktop). MatLab imports this image using the command `imread`. Notice that when we imported the image (below), we have a semicolon at the end of our command. If you don't include this, MatLab will show you the entire matrix that corresponds to this image, and that will be a real pain in the neck. So use the semi-colon!

```
>> Image=imread('C:\Documents and Settings\Public\Desktop\clock.jpg');
```

To see the image has been loaded, use MatLab's `imshow` command

```
>> imshow(Image)
```

which will display

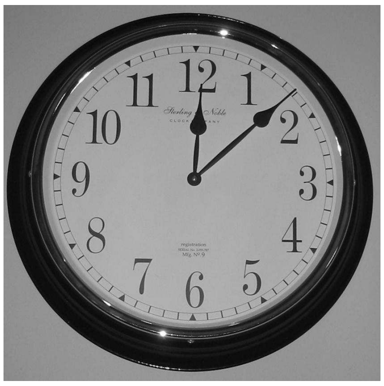

FIGURE 5

Example (continued).

Now we would like to use the Fourier transform to manipulate the image. To do this, we will first need to compute the Fourier coefficients, á la Equation 3. Of course it is inconvenient to compute each coefficient one at a time, so to save you some pain, MatLab will compute all the coefficients $c_{k,l}$ at once store them in an $n \times m$ matrix. This is done by running the command `fft2`. (Again, notice that we ended this statement with a semicolon to prevent MatLab from displaying an enormous matrix!)

```
>> J=fft2(Image);
```

The output matrix contains all the frequency information for your image, in that the coefficients $c_{k,l}$ measure the relative strength of the 'frequency' $e^{i\left(\frac{2\pi kx}{m}+\frac{2\pi ly}{n}\right)}$ in your image. Since the relative strength of each frequency determines your image, manipulating these coefficients translates into manipulation of your image.

For instance, suppose that we omitted all the entries of our matrix J whose i or j coordinate was at least 50. We will store these values in a matrix we call Jblur.

```
>> Jblur=J;
>> for i=50:843
for j=50:843
Jblur(i,j)=0;
end
end;
```

What does the image which corresponds to `Jblur` look like? The mathematical procedure for converting frequency information back into an image is called the inverse Fourier transform, and is simply Equation 2. Happily, MatLab has a built-in inverse Fourier transform function which takes care of implementing this equation, called `ifft2`.

```
>> Iblur=ifft2(Jblur);
```

This isn't quite all we have to do in order to view the resultant image. We will also need to scale the entries of Iblur so that MatLab knows how to display the image. We do this by dividing each entry of `Iblur` by the magnitude of the largest entry of `Iblur` (this means that the entries of `Iblur` will now be between -1 and 1).

```
>> Iblur = Iblur/max(max(abs(Iblur)));
```

We are now ready to view the image. Typing

```
>> imshow(Iblur)
```

displays the image

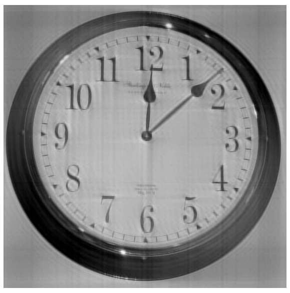

FIGURE 6

Exercise. The method we used for blurring the image above is not very sophisticated, since it dropped all frequency above a certain range, even if those frequencies made significant contributions. Here's a more refined method for omitting 'insignificant' frequencies. It drops any frequency which is relatively small compared to the dominant frequency given by $J(1,1)$ (which, by the way, has by far the largest magnitude of any of the entries of J). (Notice also that by 'relatively small' in the previous sentence, we measure smallness logarithmically. This is because the magnitudes of frequencies vary exponentially in general.)

```
>> Jblur2=J;
>> for i=1:843
for j=1:843
if (log(1+J(i,j))<.5*(log(1+J(1,1)))) Jblur2(i,j)=0;
end
end
end;
>> Iblur2=ifft2(Jblur2)/max(max(abs(ifft2(Jblur2))));
>> imshow(Iblur2)
```

Run this code and see how it affects the output of the given image. Now experiment with the coefficient 0.5 used in the `if` clause. At what value have you lost most of the detail in the image?

Now run the code above, but switch the inequality in the `if` clause from 'less than' to 'greater than.' Experiment with the coefficient 0.5, and in particular bring the coefficient fairly close to 1. What do you see?