**Sounds and Fourier Transform**

 *Part 4: Saving Digital Data*

At this point we can transform a not-necessarily period function depending on time into an equivalent one depending on frequency (via the Fourier Transform) and we have at least two ways to compute the Fourier series of a periodic function (via the integral formulas for the and coefficients or by approximating them via the Fast Fourier algorithm). But, what good does this do us?

In this section we will explain how to store sound (which is inherently analog) on the computer (which would be by definition digital). We will start with a discussion of the other formats a computer frequently has to store.

Of course the problem is that a computer can only store 0’s and 1’s. Every time your computer needs to store anything, in needs to convert this into a list consisting exclusively of zeros and ones. How does that work? Let’s start with the basics even though the mathematics involved is simple, but I think it is something that every college graduate should know.

**Bits and Bytes**

A **bit** (b) is a unit of information. It has two states, called 0 or 1. The name bit derives from the words binary digit.

A **byte** (B) is a collection of 8 bits. Most storage space is given in bytes not bit, because most data is stored in bytes (which of course means that the data is also stored in bits since every byte consists of 8 bits).

A **kilobyte** (KB) is either 1024 bytes or 1000 bytes. Traditionally, 1 KB was 1024 bytes, since storage is made of units of powers of 2. However, the prefix K for Kilo usually denotes 1000 (as in 1 km = 1000 m) and 1000 is a lot easy to deal with than 1024. Today, most people will think of 1 KB as 1000 bytes. Sticklers for details should use the abbreviation KiB for 1024 bytes, which reads as KiB = Kibibyts. Similarly,1 MB = 1000 KB = 10002 B, whereas 1 MiB = 1024 KB = 10242 B, etc. See the table for common names and their abbreviations of magnitudes of data.

**Integers**

Converting positive integers to 0’s and 1’s is easy: just convert the integer into its equivalent binary number, i.e. into a base-2 number. Recall that a decimal number x is a sum of powers of 10. For example so that a binary number is the sum of powers of 2. For example: . To convert a decimal number to a binary one is best illustrated by means of an example:

**Example:** Convert the decimal number 217 into its binary equivalent: First, we create a table of powers of two:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

 Then we find the largest power of 2 that still fits into our number 217, which would be 128. Then we start the process:

* 217 contains 1 128, so we add 1 to the table in the 128 column and compute the remainder 217 – 128 = 89
* 89 contains 1 64, so we add 1 to the table in the 64 column and compute the remainder 89 – 64 = 25
* 25 contains zero 32, so we add 0 to the table in the 32 column and compute the remainder 25 – 0 = 25
* 25 contains 1 16, so we put a 1 into the appropriate column and compute the remainder 25 – 16 = 9
* 9 contains 1 8, so we put another 1 into the 8 column and compute the remainder 9 – 8 = 1
* 1 contains no 4, so a zero goes into the 4 slot, remainder 1 – 0 = 1
* 1 contains no 2, so another zero goes into the 2 slot, remainder 1 – 0 = 1
* 1 contains one 1, remainder is 0

As soon as I have reached zero, the conversion is done: 217 = 110110012.

**Example**: What is the largest positive integer if I used 1 Byte to store each integer. How about 2 bytes? What would be the disadvantage of combining large numbers of bytes to stand for one integer?

A one-byte integer consists of 8 bits. Each of these bits would be 1 if the corresponding byte is supposed to be as large as possible. Thus, the largest positive integer you can store in 1 byte is

**Characters**

We figured out that integers are stored in bytes of 0’s and 1’s. That was easy since both binary numbers and decimal numbers are numbers. But how does the computer store non-numeric data such as letters?

Back when computers were invented, the dominate language that they had to handle was English. Thus, the more appropriate question was, back then: how does the computer store English characters? There are 26 small letters, 26 capital letters, as well as all the special symbols like @#$% etc. to store.

In all, there is only a relatively small number of English characters to store, even including special symbols. So, we generate a translation table that assigns each letter and symbol a fixed integer and use that integer – of course converted to its binary representation – to store the appropriate character. The translation table is arbitrary, as long as everyone agrees to use the same table. The table most commonly used is called ASCII (American Standard Code for Information Interchange) table. It uses 1 byte to code for characters, the first 128 of which are shown below:



**Example:** Suppose I use 8-bit (1 byte) binary numbers to store positive integers, as well as the above ASCI table. How would the name ‘Bert’ be stored?

‘B’ = 66 = 10000102, ‘e’ = 101 = 11001012, ‘r’ = 114 = 11100102, ‘t’ = 116 = 11101002 so to store the name we add a leading 0 to the binary number to use 8 bits per letter code and thus Bert = 01000010011001010111001001110100

**Note:** Nowadays English is but one language among many used on computers worldwide. Thus, the 1 byte ASCII table is no longer sufficient to code all possible languages (including for example Greek, Russian, Hebrew, Arabic, and Chinese characters) so modern computers use the 2 byte ISO or Unicode tables to translate between integers and characters (how many different characters can be represented in this way?)

**Pictures**

Next are pictures. How do I store a picture on the computer, i.e. how do I translate a picture into bytes or bits? There are a number of ways to represent images. Some of the more common formats are BMP, GIF, or JPEG. The most straight-forward format is BMP (or bitmap).

Say I want to store a **black and white** picture with 20 pixels arranged in a 4 by 5 rectangular grid. I can store this as a table of 4 rows of 5 bits each, with a 1 for a black pixel and a zero for a white one. Thus, the picture of an x in a 4 by 5 table might look like:

**00000
01010
00100
01010**

or 00000010100010001010. The last method stores all bits in one row, but I don’t know how to reconstruct the picture from that. For example, I could interpret it (correctly) as a 4 by 5 picture, in which case it would be the original x, or as a 5 by 4 picture:

**0000**

**0010
1000
1000
1010**

or as a 2 by 10 or a 10 by 2 picture, or even as a 1 x 20 or a 20 x 1 picture. To resolve the ambiguity, I define the image format BGW as follows:

1st byte: number of rows (nrow)

2nd byte: number of columns (ncols)

Remaining data: nrow x ncols bits

Thus, the above picture of a black and white x would be 000000100000010100000010100010001010. I would save this data as pic.bgw, i.e. with a bgw extension. That way the computer knows to use the BGW interpretation of the data in the file.

**Note**: The first two bytes of the data file, standing for number of rows and cols, respectively, are called the **file header**, the remaining bits would be called the **file data**. The file header describes how the file data is to be organized. The computer knows how to interpret the file header by the file extension. In the above example, the computer knows to expect two bytes, standing for number of rows and columns, respectively, because the file name has a bgw extension, and after reading the first two bytes it knows how much data to read next and how to interpret it.

If your computer does not know the extension bgw, or if the file gets renamed with a different extension, the file is more or less useless. Without knowing how to interpret the file data, the computer could read the data but it would not know what to do with it. In fact, it would not even know how much of the data represents the file header and how much the data.

**Example**: Suppose the data file “face.bgw” contains the bits 0000011000000101000000101000000100010111000000. What picture does this data represent?

So, we understand black and white pictures. Color pictures are stored similarly, except of using just 1 bit per pixel (black or white) I will use more memory to store the RGB (red, green, and blue) components of each pixel. If each color can have, say, up to 255 levels, I would need 3 bytes (or 24 bits) per pixel to indicate which of 255 levels of red, green, and blue to use for each pixel. For example, the pixel 111111110000000000000000 would be colored in full red, while the pixel 001000000100000010000000 would be a mix of 32/255 red, 64/255 half green, and 128/255 blue (see picture). Let’s say I define an image format as follows:

* Bytes 1 and 2: number of rows *nrows*
* Bytes 3 and 4: number of columns *ncols*
* Byte 5: number of bits per color
* Data: *nrow* x *ncols* x color bits

This format is actually pretty close to the bitmap (BMP) format, except the header of a BMP file stores additional information.

**Example**: Say I have a modern cell phone with a camera that can shoot 12 MegaPixel pictures in full color. How much space is needed by each picture if it was saved as a BMP file, and how many pictures would fit onto the internal storage of, say 8 GB.?



12 MegaPixel are approximately equivalent to pictures with about 4000 × 3000 pixels, i.e. 4000 rows of 3000 pixels each (since 4000 × 3000 = 12,000,000). Since each pixel requires 24 bits or 3 bytes of color information, the total storage of each image would be 12,000,000 x 3 = 36,000,000 bytes + a few bytes for the file header, or about 37 MB. Thus, I could fit 8 GB / 37 MB = 216 pictures in BMP format on that camera.

That’s not that much so I need to employ some data compression to reduce the file size per image. That’s what the image format JPEG is: JPEG = bitmap file plus data compression.

The exact way that JPEG data compression works would take us to far away from Analysis (actually, JPEG image compression happens to use FFT), but it uses redundancy of the data to not store every pixel. For example, say an image is entirely black. As a BMP file it would still require 37 MB (any picture in BMP format with the same resolution and color depth uses the same amount of storage).

 

*A black image as BMP file A black image as JPG file*

A black JPEG file I would simply store a single 0 and specify that that this zero is to be repeated across all 4000 rows and 3000 columns (different pictures with the same resolution and color depth require different amount of storage as JPEG). Instead of 37 MB, the file size would be reduced to something like 73KB.

**Movies**

Movies are simply images that change over time, plus sound that’s synchronized to the images. In order to see smooth motion, an image must change about 24 times per second. Full HD resolution is 1920 x 1080 so that an hour long movie in full HD requires 1920 x 1080 x 3 (full color) x 24 (frames/sec) x 60 (secs) x 60 (minutes) = 537,477,120,000 bytes or about 540 GB, not counting sound. Movies occasionally use a technique called “interlacing” that cuts the required data in half, so that a *interlaced* full HD movie requires about 270 GB per hour. Given that a standard, double-sided DVD can store only about 9.4 GB whereas a Blue Ray disk can store about 50 GB we can see that a 2 hour full HD movie requires at least Blue Ray disks plus some strong compression techniques with a compression ratio of about 1 to 10 or better. The problem with compression is that when the movie is playing, it need to be decompressed in real time, i.e. 24 frames per second of 1920 x 1080 x 3 = 6.2 MB (or 3.1 MB interlaced) each must be decompressed in real time, which requires a non-trivial amount of computing power.

In terms of streaming movies uncompressed at full (interlaced) HD requires a data connection of 1920 x 1080 x 3 x 24 Bytes per second or about 160 MB (80 MB interlaced) per second, which would be 160 \* 8 = 1,280 (640 interlaced) Mbits per second (Mbps) – network speed is usually measured in Mbps, not MBps - not counting sound. Thus, assuming a compression ratio of 1 to 10 or better, you would need an Internet connection speed of about 50 to 100 Mbps to stream a full (interlaced) HD movie to a private Internet connection. Streaming movies at 4K resolution is currently out of reach for most customers (see exercises).

**Note**: Some streaming services such as Netflix do claim to stream movies in 4K resolution over an Internet connection with around 10 Mbps. According to my calculations, that seems unlikely, unless they are redefining what full 4K resolution means, or they are using some highly efficient compression algorithm with compression rates of … well, that’s for you to figure out in the exercises. I suspect that they actually stream movies *up to* 4K resolution, depending on the available bandwidth, with 4KB being the top resolution if enough bandwidth was available, otherwise the rate will be adjusted dynamically according to the available bandwidth. Keep in mind that even if you have a 100 Mbps Internet connection, that does not guarantee that the movie can use the entire available bandwidth. There are many factors that can limit the rate at which data is streamed to you, the 100 Mbps would just be an upper limit if everything else was perfect.

**Sound**

Finally, we want to discuss how sound can be stored. The two most common sound formats are WAV and MP3. A typical WAV sound is sampled at 44,100 Hz at 16 bits, in other words we divide the axis into 44,100 subdivisions per second and the axis into heights. Then we record the height level for each subdivision. Thus, we need to store 16 bits of data 44,100 times a second, or 44,100 x 2 bytes = 88,200 bytes per second. In particular, this means that frequencies higher than 44,100 Hz might not get picked up at all, and we are limited to different volume levels. That’s the reason while some audiophiles do not like digital sound: very high frequencies are lost completely, and the volume levels are not continuous.



However, given that human hearing has a range from 20 H to about 20 KHz, loosing frequencies higher than 44 KHz does not seem to matter anyway, and different levels of volume should be good enough to simulate continuous volume (according to defenders of digital music).

So, we saw that saving sound as a WAV file requires 88,200 bytes per second. Thus, a song such as *Stairway to Heaven*, which is about 8 minutes long, requires 88,200 (bytes) x 60 (secs) x 8 (min) = 42 MB. Most songs are recorded in stereo, so we need 42 MB per channel (left and right) for a total of 2 x 42 MB = 84 MB for the entire song, plus some space for the file header. That figures is about right: the Stairway to Heaven file size as a WAV file is 85,098,496 bytes.

**Example:** How many minutes of stereo sound in WAV format fits on a CD? Hint: a standard CD holds around 700 MB.

We figured out that 1 second of stereo sound in WAV format requires 2 x 88,2 KB = 176.4 KB per second. Thus, a 700 MB CD can hold 700 MB / 176.4 KB/s = 3,689 sec or 62 minutes. That’s barely enough for Beethoven’s 9th symphony. Rumor has it that the length of a CD was chosen so that Beethoven’s 9th symphony would fit onto one CD. Note that music CD’s that are available for purchase generally use WAV files for all its songs (and are therefore limited to about 65 minutes of music per CD).

**Using FFT for MP3 sound format**

In addition to compression, we will now use a different strategy to store a sound: We sample each second of a sound file using, for example, 44,100 points per second, just as before, but we store the actual height of the wave at each point as a 4 byte decimal number. Then we use the FFT to compute the first 22,050 coefficients (half the sample size) of the Fourier series representing that function. We save, say, the first 1,000 of these coefficients and represent the wave using the Fourier series of the sampled sound. Thus, instead of 44,100 x 2 = 88,200 bytes/s we get by with 1,000 X 4 bytes = 4,000 bytes/s, and we have not even applied any compression.

 

Here is, for example, the file size for 30 seconds of a mono (one channel) 440 Hz A sound, saved (on the left above) as a WAV file, and (above on the right) as an MP3 file. As a WAV file, it requires 88,200 KB/s, so the file size is 88,200 bytes x 30 sec = 2.6 MB, regardless of what structure the sound actually has. Using FFT, on the other hand, gets by with storing only 4 KB worth of data per second for 30 seconds, so it should require only around 120 KB (+ file header and other info).

The sound format MP3 does, essentially, this: It splits a song into short segments. For each segment, it applies the FFT to decompose the audio wave down to its ingredient frequencies, which are then stored to represent the original wave (approximately). The FFT also tells you how much each frequency contributes to the song, so you know which ones are essential. Really high notes aren’t not important (our ears can barely hear them), so MP3s throw them out, as well as other frequencies that do not contribute much (i.e. the corresponding or is small) can be removed as well, resulting in data compression.  The complete MP3 file format is somewhat more complicated, but its main feature is that it decomposes a sound into its Fourier coefficients, throws away unnecessary ones, and stores the remaining coefficients to approximate the original sound.

Here is an example. Suppose I divide the interval into 64 subdivisions and sample a wave at those points. I get a list of points that are on the (unknown) wave:



I then feed these points into the FFT algorithm to compute the following function:

2.94209\*10^-15 + 10. Cos[x] + 4.63194\*10^-15 Cos[2 x] - 1.10566\*10^-15 Cos[3 x] -
20. Cos[4 x] + 8.92809\*10^-16 Cos[5 x] - 8.5164\*10^-15 Cos[6 x] - 8.03915\*10^-15 Cos[7 x] -
5.83588\*10^-15 Cos[8 x] - 1.63441\*10^-15 Cos[9 x] + 2.55508\*10^-15 Cos[10 x] +
7.20036\*10^-15 Cos[11 x] + 1.40469\*10^-15 Cos[13 x] + 8.17847\*10^-16 Cos[14 x] -
 2.66454\*10^-15 Cos[15 x] + 2.22045\*10^-15 Cos[16 x] + 8.09953\*10^-15 Cos[17 x] +
2.63354\*10^-15 Cos[18 x] - 6.23075\*10^-15 Cos[19 x] - 5.32907\*10^-15 Cos[20 x] +
 9.9678\*10^-15 Cos[21 x] + 1.77641\*10^-15 Cos[22 x] - 3.06308\*10^-15 Cos[23 x] -
8.81906\*10^-15 Cos[24 x] - 7.38732\*10^-15 Cos[25 x] + 1.29046\*10^-15 Cos[26 x] +
 2.65973\*10^-15 Cos[27 x] + 6.21725\*10^-15 Cos[28 x] - 2.79856\*10^-15 Cos[29 x] +
1.93708\*10^-15 Cos[30 x] - 4. Cos[31 x] +

 9.21355\*10^-17 Sin[x] - 4.70722\*10^-15 Sin[2 x] - 7.36959\*10^-15 Sin[3 x] +
1. Sin[4 x] - 3.7774\*10^-15 Sin[5 x] + 12. Sin[6 x] - 2.39432\*10^-15 Sin[7 x] +
2.96508\*10^-15 Sin[8 x] + 6.02368\*10^-15 Sin[9 x] + 7.99361\*10^-15 Sin[10 x] -
 3.5376\*10^-15 Sin[11 x] + 2.9976\*10^-15 Sin[12 x] - 1.53591\*10^-15 Sin[13 x] +
1.46111\*10^-16 Sin[14 x] - 1.35492\*10^-15 Sin[15 x] + 7.88258\*10^-15 Sin[16 x] +
 2.22078\*10^-15 Sin[17 x] - 1.23902\*10^-14 Sin[18 x] - 8.65704\*10^-16 Sin[19 x] +
6.88338\*10^-15 Sin[20 x] - 7.90695\*10^-16 Sin[21 x] - 2.51856\*10^-15 Sin[22 x] -
 9.05566\*10^-15 Sin[23 x] - 2.80808\*10^-15 Sin[24 x] + 4.57622\*10^-15 Sin[25 x] +
7.54952\*10^-15 Sin[26 x] + 5.27398\*10^-15 Sin[27 x] - 2.96985\*10^-15 Sin[28 x] +
 1.23447\*10^-15 Sin[29 x] + 1.22664\*10^-15 Sin[30 x] - 1.16279\*10^-14 Sin[31 x]

Most of the coefficients are very close to zero, so they contribute virtually nothing to the overall sound. Thus, we can reduce the wave to:

Moreover, the cos(31 t) won’t change the sound too much because its frequency is much higher than the other frequencies, and the sin(4t) term won’t contribute much because its amplitude is much smaller than the other components. Thus, we can save this wave as:

which saves us two coefficients. Here is the graph of both of these functions; they are quite close:



The 8 minute song “Stairway to Heaven”, which used 85 MB as a WAV file, requires only 12 MB as MP3 file.

The fact that sound files are so ubiquitous, in fact the whole idea of digital music, music players, and smart phones, would not be possible without the MP3 file format, and hence without the FFT.

**Questions:**

1. The difference between 1 KB and 1 KiB is just 24 bytes. What is the difference between 1 TB and 1 TiB?
2. Prove that
3. If 8 bits are combined into one byte, what is the largest integer you can store using 1 byte? How about using 2 bytes? Note that this time I want to be able to store *positive and negative* integers in one (or two) byte(s).
4. What does the last digit of a binary number indicate? How can you tell whether a binary number equals a power of two? How do you multiply a binary number by 2? How about dividing by 2?
5. Convert 1397 to a binary number, and 101011001100111000111 to a decimal number
6. Add the binary numbers 100111 and 10011 as binary numbers (convert them to check your answer if necessary)
7. Why is it important to use the same number of bytes for each character in the ASCII table. Doesn’t that waste space and, thus, memory?
8. What does the following sentence say (it uses the standard ASCII table and 1 byte integers): 0110001101100001011100100111000001110000011001010010000001100100011010010110010101101101
9. What does the following picture show (file name: mystery.bgw): 0000011000000101000000101000000011101000100000
10. Explain the significance of the extension of a file. What happens if you changed a file extension from, say, BMP to JPG or to DOCX without changing any of the actual data in the file? Explain.
11. How big could pictures stored in the BGW file format described in the text be at most?
12. How many 20 MegaPixel pictures in full color BMP format fit onto a 2 GB Flash Drive?
13. We figured out that 1 hour of full uncompressed HD movies requires about 540 GB of space. How much space does a standard DVD resolution of 720 x 480 pixels require? How about a 4K movie, which runs at a resolution of 3840 x 2160? If you had a compression ratio of 1 to 10, how fast would your Internet connection have to be to stream movies in full 4K resolution?
14. Explain why the MP3 format is what’s called a “lossy format”. Note that if you lost the WAV file of a song and only keep the MP3 version, you will not be able to reconstruct the original complete WAV file. Thus, it is important when converting a song to MP3 format, to pick the least amount of loss possible, which will impact the compression ratio and increase the fie size.
15. Suppose a song was given by the following wave function. We want to save this song as a pseudo-MP3 file, including data compression, using Mathematica. Here are the steps:
	1. Divide the interval from 0 to into 64 subdivisions
	**X = Table[2.0 Pi\*i/64, {i,0,63}]**
	2. Sample the function at those 64 points. For this you need to know the function; if this was a real song you would not know it but you would merely sample the function values numerically. In this case the function happens to be
	**Y = 10 Cos[X] + Sin[4X] - 20 Cos[4X] + 12 Sin[6X] - 4 Cos[31X]**
	3. Apply the FFT to the sample you just computed, using a scaling factor of 1/4:
	**coeffs = Fourier[Y]/4**
	4. Extract the Fourier coefficients from the answer:
	**a0 = Re[coeffs][[1]]/2
	a = Re[coeffs][[2;;31]]
	b = Im[coeffs][[2;;31]]**
	
	5. Set all coefficients that are close to zero equal to zero. You will notice that one frequency is much higher than all others – drop it. Also, one of the remaining frequencies does not contribute much to the amplitude of the wave – drop that one as well. You have left just a few coefficients that you can store as representing the original wave, including “compression” (i.e. removing high frequencies and low-amplitude ones).