PH 120 Project # 3: Frequency Analysis, Vision, and Sound

Due: Friday, January 23, 2004

In the pendulum project, we used techniques like Poincaré sections and bifurcation diagrams to try to understand the *periodicity* of complex motion. These techniques were crude but effective, partially because we had a definite frequency for our "strobe light" — the driving frequency ω_d. The goal of this Project is to develop some more sophisticated techniques for analyzing the frequency content of a signal. A particular application will be an extremely crude model for vision.

This assignment is set up in C++, but if you're familiar with Java you should have no problem using that instead. If you choose to use Mathematica, your task will be to implement the same functionality as I describe in C++; I've included some Mathematica-specific notes to guide you at the end of the problems.

Important: Since we will use basic command-line tools for C++, we won't have any nice notebook system available to record all of your steps. Nonetheless, I **do** want you to keep a careful log of how you worked through each problem, including the same kinds of explanations and interpretations you've been including as comments in your notebook. This writeup should include explanations of what you did, with output data and especially plots. However, I'm not picky about how you put this together: you can use a Mathematica notebook into which you paste your data (Mathematica can also read in text files, if you prefer); or you can just print out your plots in Excel or gnuplot and refer to them in your writeup (hand-labeling is fine); or anything else that produces a clear, readable document. The only thing you need to submit electronically is the code: this should be set up to illustrate how you did each type of calculation, but within a given calculation, one representative example is fine: you don't need to display everything you ran within the code, but someone reading your writeup should be easily able to tweak the code and reproduce what you did. Your writeup, however, should include all the significant calculations you did and results you obtained. (Mathematica users, of course, should still just hand in a Mathematica notebook will all of this information.)

C++ warning: in this assignment, you will be using both integer values and double precision floating-point values. Be careful to keep them straight! This is particularly important when it comes to division. For example: `4/5` will evaluate to 0 in C++, because C++ does integer division. Similarly, `11/5` will evaluate to 2. To do floating-point division, you need to say `4.0/5` or `double(4)/5` so that the compiler knows to promote everything to floating point.

1. Start with the file `guess.cc` from the course folder (or the course website). This is a rather lame program to begin with, but we'll work on it. Start by reading the code in a text editor and seeing if you can understand what it it doing. Then make sure you can compile it and run it, and see that you get the output you expect.

   Just to remind you, to compile with gcc on Linux, open up a shell and type

```
g++ file.cc
```

which will create an executable `a.out`. To run it, type

```
./a.out
```

where the `./` is to tell the system to look in the current directory (some systems nowadays are configured not to run programs in the current directory by default for security reasons).

On Windows, it works the same way from the Cygwin shell, except the executable is called a.exe, so you run it with `./a`. Cygwin is available for free download from `http://www.cygwin.com`. If you want to install it to your own machine, just follow the defaults except for one change: when you get to the screen where you can configure various package groups, change the "Devel" package from "Default" to "Install" (just click on it once).

For OS X, the gnu compiler should work the same way as in Linux, though if it's not pre-installed on your computer you have to download it from Apple's developer tools website `http://developer.apple.com`.

(a) We'd like to turn this into C++ code we can be proud of. Rather than having the work done in `main()`, we'd like to have it all done in the class, in such a way that the rest of the code doesn't get access to the implementation details. In the end, we would like to have an class `Guesser` such that when we create an instance `myGuesser`, the instance gets handed the correct answer when it is created, does a sanity check on that answer, and then `myGuesser(n)` returns true or false depending on whether the guess `n` is correct or not. Here is a suggested route to that result:

   i. Start by protecting the member variable `mAnswer` that encodes the answer. Doing so will break the existing code, since it accesses `mAnswer` directly. Remedy this by creating get and set member functions on the class and calling these. Make the set member function check the input to make sure it doesn't exceed the built-in limit.

   ii. Replace the get member function with a member function `bool guess(int n)` that takes a guess and returns `true` or `false` depending on whether the guess is correct or not. (Note that `bool` is a built-in type, like `double` or `int`, which can take only the values `true` and `false`.

   iii. Replace the set member function by a constructor, so that the correct answer is given when the object is created.

   iv. Replace the `guess` member function with an operator so that you just call `myGuesser(n)` instead of `myGuesser.guess(n)`.

(b) Create a `LenientGuesser` class that inherits from `Guesser`, and overrides the `()` operator to return true if the guess is within 1 of the correct answer.

Mathematica users: You should accomplish a similar task by defining a function `makeGuesser[answer_]`, which returns a guesser function corresponding to the answer you put in. Then create a loop in which you verify that the guesser function you created actually works. Ditto for `makeLenientGuesser[answer_]`. Use this same sort of approach throughout the problem set.

2. Next we would like to assemble the building blocks of our very crude model of the visual system. First, we will make everything one-dimensional instead of two-dimensional. Imagine that the visual field consists of a fixed number of pixels N. An *image* consists of an assignment of a brightness value to each pixel. Two specific examples of images will be of interest to start with: First, we can have a "blip": the n^{th} pixel at brightness 1, with all the others at zero. Here the position n ranges from 0 through $N-1$. Second, we can have a regular "texture" (think corduroy): an oscillating brightness, so that pixel j has brightness $\sqrt{\frac{2}{N}}\left(\cos\frac{\pi\omega j}{N}+1\right)$ where the frequency ω is an integer from 1 to N. (The extra 1 ensures that the brightness is always positive, and the factor of $\sqrt{\frac{2}{N}}$ makes the normalization comparable to the blip.)

   (a) Create two classes, `BlipImage` and `TextureImage`, both inheriting from an abstract base class `Image`. When constructed, one should have to specify either the location n of the blip or the frequency ω of the texture. Create a virtual, overloaded operator, so that if `myImage` is an instance of either class, `myImage(j)` returns the brightness at the pixel j. Generate a plot of a couple of `TextureImage`s and explain what happens to the image when you change ω.

   Mathematica users: Again, you will want to create functions `makeBlipImage[n_]` and `makeTextureImage[w_]` returning *functions* that in turn give the blip or texture as a function of position; similarly for the subsequent parts of the problem.

   (b) Having modeled some very simple images, we would next like to model the way that the visual system could process these images. A *receptor* is a cell that takes some combination of the input pixels and sends the brain an output signal. For example, a one receptor might add up the brightness of all the pixels and divide by the number of pixels, thus returning the average brightness of the entire visual field. Or a receptor could return just the brightness of a particular pixel, ignoring the brightness of all the others. Or it could do more complicated things, such as taking the difference between the sum of all the odd pixels and the sum of all the even pixels.

   Create an abstract base class `Receptor` defining the receptor interface, which should take a pointer to an `Image` (of any kind) and return the response of the

particular receptor to that image. Define derived classes `AverageReceptor`, returning the average brightness, and `BlipReceptor`, returning the value of a particular pixel (when you create one of these, you will have to specify which pixel).

Mathematica users: You should create a function `makeBlipReceptor[n_]` that returns a function mapping an image to a the response of the receptor to the image, so that, for example,
`makeBlipReceptor[n_][makeBlipImage[p_]]`
returns 1 if $n = p$ and zero otherwise; similarly
`makeBlipReceptor[n_][makeTextureImage[w_]]`
should return the response of the blip receptor at n to the texture image with frequency ω.

(c) Suppose you know that your image is a `BlipImage`, but you don't know where it is. Write code that uses `BlipReceptor`s to find the location of the blip. (This is not a complicated algorithm.) Try a few examples and show that your code works.

(d) In reality, it's hard to make a receptor with such high resolution — a receptor centered at pixel n tends to get corrupted by adjacent pixels. Make a `FuzzyBlipReceptor` with the following profile: the `FuzzyBlipReceptor` centered at n returns the average of the brightness at n with weight $1/2$ together with the brightness at $n \pm 1$ with weight $1/4$ each. (For the pixels at the edges, you may assume that the missing pixels have zero brightness, or the same brightness as the center pixel.) Show that you can still locate a `BlipImage` accurately with a `FuzzyBlipReceptor`.

(e) Now take a `TextureImage` with a particular ω. Explain why it's hard to figure out its ω using `FuzzyBlipReceptor`s. (Or, if you think it's easy, write a program to do it!) Instead, define a `TextureReceptor` that works as follows: It sums the value of the brightness over all the pixels, with the j^{th} pixel weighted by the factor $\sqrt{\frac{2}{N}} \cos \frac{\pi \omega j}{N}$, where ω is an integer from 1 to N. Write code to use `TextureReceptor`s to identify the ω associated with a given `TextureImage`. Demonstrate your code with a few examples, and explain how this works.

(f) Now suppose you can be given an image that is either a `BlipImage` or a `TextureImage`. Write an algorithm that uses both `BlipReceptor`s and `TextureReceptor`s to first determine which kind of image you have, and then find the image's n or ω as appropriate. Again, demonstrate your code with an example of each kind.

3. Although your eye actually does contain analogs of both `BlipReceptor`s and `TextureReceptor`s, a different kind of receptor plays a central role. Having a full set of both kinds of receptors seems like overkill; we've created $2N$ receptors to process only N pieces of

information. Also, it's rather artificial: One set of receptors was entirely designed for isolated points, while the other was entirely designed for regular patters. In the real world, these possibilities can mix; for example you could have a regular pattern that only covers a small range of the visual field. We'd like a single, generic set of receptors that can extract information both about an image's location and its patterns, and as a tradeoff we'll be willing to accept only approximate results in both cases.

(a) Define a `GaussianReceptor` like the `TextureReceptor`, except using the function $\sqrt{\frac{\omega}{2\pi^2 N}} e^{-\omega^2(j-n)^2/(2\pi^2 N^2)} \cos\frac{\pi\omega(j-n)}{N}$ for the response.
Thus a particular `GaussianReceptor` is specified both by a value of n and ω. Generate a plot of the receptor profile for some sample values to see what it looks like, and explain what you expect varying these values to do. Loop over all the possible values of both these two parameters (don't set your N too big here!) to see how well you can do at identifying the different kinds of images.

(b) In the previous problem, there were N^2 different possible `GaussianReceptor`s you could construct. Gradually cut down the number of receptors you use to smaller and smaller subsets. When you do so, can you still do a reasonably good job of identifying whether an image is a blip or texture, and finding (approximately) its corresponding n or ω? How few receptors can you use, given a reasonable error tolerance?

(c) One measure of how we are doing is to try to recreate the original image from the output of the receptor. Create a subclass of `Image` called `RecreatedImage`, which starts out empty (all pixels zero). Then give it a member function `addReceptorOutput`, taking a `Receptor` and a coefficient. The `RecreatedImage` should add to the value of each pixel this coefficient times the weight that the receptor uses for that pixel (e.g. in the case of a `GaussianReceptor`, that complicated Gaussian function). You'll need to augment the `Receptor` interface to accommodate this change. The idea is that you start with an image, and find each receptor's response to the image. Then you add up the profile function being used by each receptor, weighted by how that receptor responded to the original input, and see how well you did.

Starting from both a blip and a texture, try reconstructing the original image using `BlipReceptor`s, `TextureReceptor`s, a large set of `GaussianReceptor`s, and a small set of `GaussianReceptor`s. In each case, draw a graph of the resulting image. How do they do?

Note: We're just interested in the general shape — don't worry about problems at the edges, or if you get an overall shift (this would be fixed by including an `AverageReceptor`) or rescaling of the image values (this would be fixed by normalizing more precisely in proportion to the number of receptors).

Epilogue: the tradeoff between localization in space (the ability to identify a blip) and localization in frequency (the ability to identify a texture) is ubiquitous in

the physics of wave phenomena. For example, when you hear that an Internet connection has high "bandwidth," it means that it can transmit a wide range of frequencies. That is, it can transmit a signal that would generate a significant response across a wide range of `TextureReceptor`s. Thus it can send a very short blip, which is what you need to transmit data quickly.

In quantum mechanics, using `BlipReceptor`s is exactly analogous to measuring a particle's position, while (for less intuitive reasons, which you'll learn if you take PH202) using `TextureReceptor`s is exactly analogous to measuring its momentum. The catch is that in quantum mechanics, measurement affects the system, so if you measured with all your `BlipReceptor`s first, the system has been changed so you can't then go back and measure with `TextureReceptor`s (and vice versa). This is the *uncertainty principle*: You can't know precisely the particle's position and momentum simultaneously. In practice, one is more likely to use the analog of a `GaussianReceptor`, which gives you approximate information about both quantities.

Ideas related to image reconstruction also have very practical applications in data compression. Compressing an image, for example to create a `.jpeg` file, essentially means creating a small but well-chosen set of receptors, and then recording only those receptors' response to the image rather than the whole image. For audio signals, which you'll look at in the next problem, compression algorithms like those used to produce `.mp3` files also work this same way.

4. These ideas are also directly applicable to *sound*. In this case, a 1-dimensional model is exactly what we want. Now the position of the pixel labels a moment of time, and the brightness corresponds to the variation in air pressure at that moment (which can be positive or negative). A standard audio format for storing such data is the `.wav` file. In the files `wav.cc` and `wav.h` are classes `WAV_IN` and `WAV_OUT` that read and write `.wav` files. The `main()` routine in `wav.cc` gives an illustration of how to use these — it just reads in and writes out the same file. Note that you only need to understand the public sections of the class declarations (in `wav.h`), since those are the only functions you can call. Once you understand how to use this code (you don't have to worry about the details of how it works), remove the `main` routine, since we will want to incorporate this code into another program (which already has a `main`. We'll want to read in the data from the `.wav` file, modify it, and then write it out so we can play it. I've provided the sound `tada.wav` to use as a sample, though you're welcome to use something else; I would recommend something short (2-3 seconds), monaural, and sampled at a low frequency (under 16 kHz), just to keep total the amount of data under control. (If you have a sound recorded at higher quality, for example from a CD, utilities like Windows sound recorder can save it out with these lower settings).

   Note: To compile including the code in `wav.cc`, just add it to the command line:

   ```
   g++ wav.cc file.cc
   ```

where `file.cc` is your code. You'll also need

```
#include "wav.h"
```

in your code so that it can see the declarations in `wav.h`. (If you want to put your code for this problem in a new source file, you can also use this same approach to put code you wrote in the last problem into a header, which you can then include in both places.)

Java users: I've provided SoundReader.java and SoundWriter.java class files providing similar functionality. You need the javax library on your CLASSPATH.

Mathematica users: Mathematica has all the functionality you need built in. Enter the commend `<< Miscellaneous‘Audio‘` to load its audio package, and then you can use `ReadSoundFile` to read in the data and, after you've modified it, `ListPlay` to play it.

(a) Create a `SoundImage` object that, given the name of an input wave file, reads its data in as an image. You can continue to use a fixed number of pixels (rather than the whole file), but you'll need to increase the number — the lowest sample rate a `.wav` file can have is 8kHz, which means it takes 8000 samples just for 1 second of sound.

(b) Create a `SoundImageSaver` class that, given the name of an output wave file, an image of any kind, and a pointer to a `WAV_IN` object, saves out the image as wave data. (The only reason that it needs the `WAV_IN` object is to copy the sampling data. The wave file I provided is 8000 samples per second, 16 bits per sample, 1 channel, and you're welcome to just hard-wire those settings if you prefer.) Check that you can read your `SoundImage` and write it out. Create a few `TextureImage`s and play them as wave files — do they sound as you'd expect? (Be sure to use a high enough frequency.) You can also try a `BlipImage`, but you probably will have to combine several of them together to be able to hear anything.

Note: When writing the file, your values must be between -1 and 1. (On Mathematica, they're integers from -32768 to +32767.) It's easy to let it go out of range. If you find this happening, just be sure to rescale all of your data to fit in the range again. (Don't cut it down too much or you won't hear anything!) Conversely, if your sound is too quiet, you should scale it up. Probably it's worth making this scale an argument to the saver class.

(c) As you did in the previous problem, break your `SoundImage` up with `BlipReceptor`s and then reassemble it into a `RecreatedImage`. Does it sound the same? What happens if you just use the first $N/2$ blips? (This should not be surprising.) What if you only use `BlipReceptor`s centered at every other point? Every fourth point? What if you use a `BlipReceptor` at every other point, but copy its value to the value you skipped? What if you do this at every fourth point?

(d) Now repeat the process with `TextureReceptor`s. What if you use only the first $N/2$ textures? What if you use only the second $N/2$? What if you skip every other one, either by leaving it out or copying the previous value?

(e) Finally, try to reconstruct the sound using `GaussianReceptor`s. Since you have N^2 possibilities here, it will be important to cut down the number of receptors you use. Start with a short segment of the sound, and see how few receptors you can get away with. Which values of n and ω is it most important to keep? How few can you use and still have a recognizable sound?

5. (Adapted from Georgi, *The Physics of Waves*) Finally, let's connect these ideas with what we studied last week. Use the following function to create a `FractalImage`:

$$f(j) = \sum_{k=0}^{\infty} h^k g\left(\text{frac}\left(2^k j/N\right)\right)$$

where $\text{frac}(t)$ denotes the fractional part of t (so $\text{frac}(3.14)$ is 0.14),

$$g(t) = \begin{cases} 1 & \text{for } 0 \le t \le w \\ 0 & \text{for } w < t < 1-w \\ 1 & \text{for } 1-w \le t \le 1 \end{cases}$$

and h and w are constants with $0 < h < 1$ and $0 < w < 1/2$, parameterizing the shape of the image.

(a) Plot a graph of this image function. Note that for a given precision of the pixel values, you only need a finite number of terms in the sum. Why? Explain qualitatively how the shape of the image depends on the parameters h and w.

(b) Describe how the image changes as we increase the resolution. Explain how the infinite sum, in the limit of infinite resolution, generates a fractal (self-similar) pattern.

(c) Use `TextureReceptor`s to reconstruct the image as you did above. Explain what happens to the reconstructed image if you only keep high or low frequencies. (You might also try listening to it!) What does this say about the frequency content of a fractal?

6. (You don't have to hand anything in for this part.) Start thinking of some ideas for your final project!