CS 370 Winter 2020: Assignment 3

Due Friday, March 20, 5:00 PM.

Submit all components of your solutions (written/analytical work, code/scripts/notebooks, figures, plots, output, etc.) to CrowdMark in PDF form in the section for each question.
You must also separately submit a single zip file containing any and all code/notebooks/scripts you write to the A3 DropBox on LEARN, in runnable format (that is, .ipynb).

For full marks, you must show your work and adequately explain your logic!

1. (**8 Marks**)

   (a) Compute (by hand) the discrete Fourier transform vector F for the input

   $$f = (2, 3 + i, 5 - 2i, 3)$$

   by directly using our *definition* of the DFT (i.e., *not* the Fast Fourier Transform method).

   (b) Compute the half-length vectors g and h that would be used in *only* the first stage of the butterfly FFT process applied to the same input data f (course notes, Section 5.8). Then *using the definition* of the DFT, determine their corresponding DFT vectors $G = DFT(g)$ and $H = DFT(h)$. Do these values make sense, given what you found in (a)? Briefly explain (one sentence) why or why not.

2. (**8 marks**) Consider the sequence of eight numbers

   $$f = [4, -2i, 2i, -8, 4, -2i, 2i, -8].$$

   Perform a complete (butterfly) Fast Fourier Transform process to determine the corresponding vector F satisfying $F = FFT(f)$. Show your work.

3. (**9 marks**) Let $\{F_0, ..., F_{N-1}\}$ be the DFT of a sequence of data $\{f_0, ..., f_{N-1}\}$ where

   $$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n W^{-nk}$$

   and $W = e^{\frac{2\pi i}{N}}$ as usual.

   (a) Derive a simplified expression for F_k when $f_n = sin(\frac{4n\pi}{N}) - 1$, for $n = 0, 1, ..., N-1$. You may assume $N > 4$.

   (b) Derive a simplified expression for F_k when $f_n = W^{2n}$ for $n = 0, 1, ..., N-1$. You may assume $N > 2$.

   (c) Derive a simplified expression for F_k when $f_n = (-1)^n$ for $n = 0, 1, ..., N-1$, for *even* values of N.

4. (**8 marks**) Derive a method to determine the FFT of two *real* signals of the same length that requires performing only a single FFT of a *complex* signal (also of the same length). Describe your method in words and appropriate mathematics, and explain/justify how and why it works.

5. **(Audio Signal Processing, 10 marks)**

The sound file **train_bird.wav** can be downloaded from the Piazza resources page. This sound file can be loaded into your Jupyter notebook and played using:

```
from IPython.display import Audio
import scipy.io.wavfile
Fs, y = scipy.io.wavfile.read('train_bird.wav')
Audio(y, rate=Fs)
```

The sound data consists of a signal array y and a sampling rate Fs.

You should hear a bird chirping and a train whistle. The combined recording has been generated by superimposing the two signals. As seen in the time-domain plot below, it would be rather difficult to separate the bird chirps from the train whistle directly from this mixed signal, since they heavily overlap in time. Instead, you will attempt to separate them in the *frequency-domain* using the DFT, exploiting the fact that the sounds have different frequency ranges.

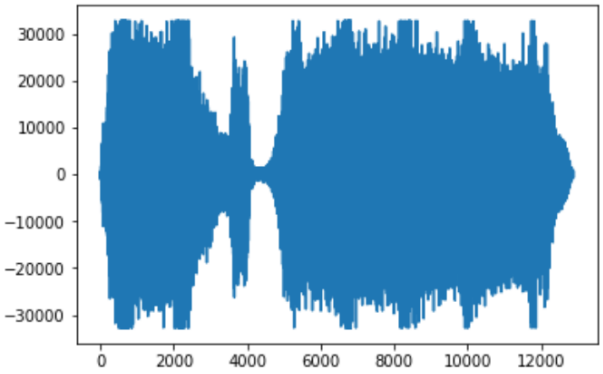

Figure 1: The combined time-domain signal of the train and bird sounds.

Write a Jupyter notebook, `A3Q5.ipynb` to carry out the following...

Plot the original signal, and plot the magnitudes (abs) of the DFT of this signal, with appropriate titles.

Read the appendix to this assignment on Filtering. Then design a low pass filter which isolates the train sound and a corresponding high pass filter to instead isolate the bird chirps. You should be able to do this by inspecting the frequency plot of the input, and using a little bit of trial and error. (Do not worry if you cannot absolutely *perfectly* separate them — separate them as best you can with a reasonable effort.) Playing back the resulting filtered sounds should allow you to hear the separated train and bird sounds, respectively.

Produce plots of each of the final filtered signals, for both the magnitudes of the filtered frequency-domain data and the new time-domain data.

6. (**DFT-based Image Compression, 16 marks**) In Jupyter, a grayscale image can be loaded, converted into a 2D array of floating point values, and viewed with the following commands

```
f = np.array(plt.imread('operahall.png'), dtype=float)
plt.imshow(f,cmap='gray');
```

(provided that `numpy` and `matplotlib.pyplot` are loaded).

In this problem, we study a simple form of compression on grayscale images. We will obtain compression by dropping (relatively) small Fourier coefficients on 32×32 pixel subblocks of an input image. By this we mean that if $\{f_{i,j}\}$ are the original pixel values in a given subblock and $\{F_{k,l}\}$ are the corresponding DFT values, then we drop (i.e., set to zero) any $F_{k,l}$ such that

$$|F_{k,l}| \leq F_{max} \cdot tol.$$

Here F_{max} is the maximum of $\{|F_{k,l}|\}$ *within* a given block and *tol* is a specified (global) drop tolerance. The file `operahall.jpg` included with the assignment is a grayscale image which we will use for all parts of this compression question. Prepare a Jupyter notebook to carry out the operations described below.

(a) **Visualizing A 2D FFT**
Perform the following:

- Load the image, extract the **top-left** 32×32 sub-block of pixels, and plot it.
- Perform a 2D FFT on this block using `fft2` to get a new 2D array of complex numbers, which we'll denote as F.
- Use `imshow`, again with the gray scale colormap, to visualize the magnitudes (i.e., modulus or absolute value) of the 2D array of DFT data. State which pixel is the brightest (whitest) and explain what significance this value has.
- Zero out the DC coefficient of F, and use `imshow` once more to visualize the data.

(b) **Compression Process**
Create a function,

`Compress(X,tol)`

which takes as inputs an original image as an array of floats, `X`, and the drop tolerance parameter, `tol`. It should return a tuple [Y,drop] which contains the compressed image `Y`, as another array of floats, and the computed drop ratio, `drop`, which is defined as:

$$\text{drop ratio} = \frac{\text{Total number of } \textit{nonzero} \text{ Fourier coefficients dropped}}{\text{Total number of } \textit{originally nonzero} \text{ Fourier coefficients}}.$$

If drop ratio = 0, then no nonzero Fourier coefficients have been dropped; if the drop ratio = 1, then all nonzero Fourier coefficients have been dropped. In general, the drop ratio will be between 0 and 1.

Specifically, your function `Compress` should:

- compute the 2D Fourier coefficients (`fft2`) for every 32×32 subblock.
- for each subblock, determine its F_{max} and set each of its Fourier coefficients having magnitude/modulus less than or equal to $F_{max} \cdot tol$ to 0.

- find the total *(global)* number of nonzero coefficients and number of dropped nonzero coefficients, and use them to compute `drop`.
- reconstruct the new/compressed 32×32 image array by using the inverse 2D Fourier transform (`ifft2`), and then discarding any *imaginary* parts of the resulting data.
- after all the 32×32 subblocks for all the components have been processed, return the complete reconstructed ("compressed") image as `Y` and the computed drop ratio as `drop`.

(c) **Compression Levels**

Determine by trial and error on different `tol` values (i.e., not by writing any code) four values of `tol` resulting in drop ratios of approximately 0.5, 0.8, 0.92, and 0.97 (to 2 significant digits on the tolerance). Then in the Jupyter notebook, you should...

- Call your `Compress` function with the set of different `tol` values you determined.
- Plot the four compressed images, using `imshow` for each compressed image `Y`. Each plot should have a title indicating the `tol` value used and the resulting drop ratio.
- Of the four images reconstructed above, visualize the error in the result image with the *least* compression (highest quality), using `imshow` (again with gray scale colormap) on the 2D array of absolute values of the difference between the original image and the compressed image.

Appendix: Filtering a Signal

Recorded sounds are often processed by carrying out a *filtering* operation in the frequency domain. Suppose we are given an input signal $x_i, i = 0, ..., N-1$. Let $X = FFT(x)$. The Fourier representation has frequencies in the range $\{0, ..., N/2\}$, However, note that $X_{N-k} = X_{-k}, k < N/2$ really represents frequencies of size k, not $N-k$, since we have used a complex representation of the Fourier series, and we have defined $X_{k \pm N} = X_k$.

If we use the conventional range of $X_k, i.e., k \in [0, N-1]$, this means that we have to do the following to construct a *lowpass* filter. Let $p < N/2$ be the maximum value of the frequency which will be allowed to pass our *lowpass* filter (i.e., retained). In other words, we will eliminate (zero out) any frequencies in the signal with index $> p$. This is easily accomplished using the lowpass filter

$$\begin{aligned} Q_k &= 0 \;\; ; \;\; k = p+1, ..., N-p-1 \\ &= 1 \;\; ; \;\; \text{otherwise} \end{aligned} \tag{1}$$

Note that due to conjugate symmetry of the DFT of a real signal, the filter should be symmetric about $N/2$.

Another way to think about equation (1) is to imagine plotting X in the range $[-N/2 + 1, .., +N/2]$. Then, we want to zero all the X_k such that $k > p$ or $k < -p$. This defines a filter $Q_k, k \in [-N/2+1, ..., N/2]$. Now, define the filter in the range $[0, ..., N-1]$ by a periodic extension $Q_{N-k} = Q_{-k}, k = 1, .., N/2 - 1$.

The filtered signal in the frequency domain $\hat{X}$ is then

$$\hat{X}_k = X_k Q_k \;\; ; \;\; k = 0, ..., N-1 \tag{2}$$

and the filtered signal in the time domain is $\hat{x} = Real(IFFT(\hat{X}))$.

A high pass filter is constructed in a similar way, except we want to remove frequencies $< p$.

A note on FFT conventions

The convention for the FFT in SciPy (and elsewhere) is slightly different from that used in our class and the course notes (see section 5.5.1). In class, the DFT/IDFT pair was defined as

$$\begin{aligned} F_k &= \frac{1}{N} \sum_{n=0}^{N-1} f_n \, W^{-kn} \\ f_n &= \sum_{k=0}^{N-1} F_k \, W^{kn} \end{aligned}$$

where $W = e^{\frac{2\pi i}{N}}$ whereas SciPy defines the above pair as

$$\begin{aligned} F_k &= \sum_{n=0}^{N-1} f_n \, W^{-kn} \\ f_n &= \frac{1}{N} \sum_{k=0}^{N-1} F_k \, W^{kn} \end{aligned}$$

The two definitions differ in the place where the normalization by $1/N$ occurs, in the forward or inverse transform; since this is simply multiplication by a constant, it is straightforward to convert between the two. In all analytical work, we will use the definition as given in class unless otherwise specified. For coding questions, using SciPy's convention is fine.