CoE 164

Computing Platforms

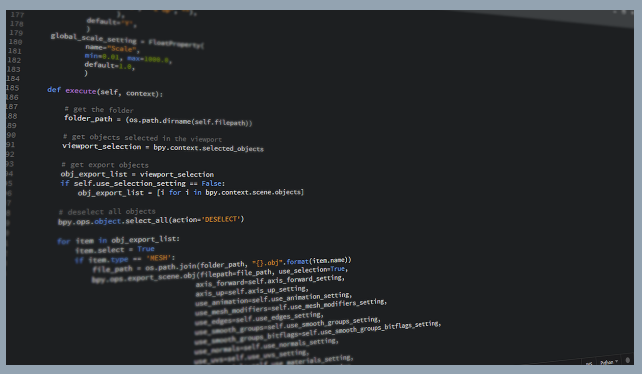
Machine Problem 02

Academic Period: 2nd Semester AY 2020-2021
Workload: 6 hours
Synopsis: The fastest fast Fourier transform?
Submission Platform: Google Forms

Description

It was the early half of 2005, and you were getting exasperated with using Python to do your data science work. That is because you need to import two different libraries that perform the same matrix manipulation and math functions on differently-sized matrices - namely Numeric and Numarray. Numeric is a relatively old library that is rich with methods for matrix manipulation but is proven to be slow on very large data. You actually contributed a library of math functions on top of it when you were a graduate student in the late 1990s. On the other hand, Numarray was created as a response to this problem experienced by Numeric. Unfortunately, it has the reverse issue - it can manipulate large amounts of data, but is slow on small data! And worse, the data structures between the two are totally different! You would need to convert your Numarray matrix to one that can be recognized by Numeric, and vice versa. This is boilerplate code, and increases the amount of code and time just to make this kind of interoperability work.

With that problem in mind, you are determined to make a library that will combine the features of these two libraries together. Numarray has an expressive matrix slicing syntax and broadcasting rules similar to Matlab that Numeric can benefit from. Since Numeric has that rich set of math functions thanks to you, you have decided to bring the features of Numarray to Numeric.



While doing the porting, you suddenly remembered SciPy, which is a math toolkit released last 2001. SciPy has ported your late 1990s math library to theirs and has grown to a big library with lots of math functions. It would be a waste of disk space if you only need, say, the discrete Fourier transform (DFT) in it. So for the next few weeks, you have decided to test whether it is possible to refactor the most important math functions in SciPy and merge it into your Numarray + Numeric library by using the DFT as one of the bases. Although this DFT is actually your port from a FORTRAN library, you still decided to try to re-code it in another programming language.

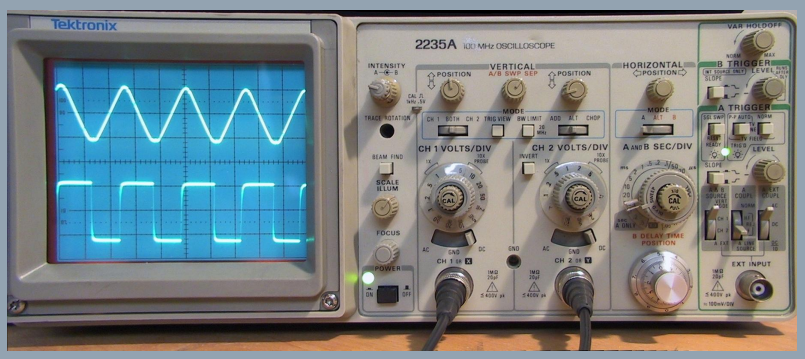
The DFT of a one-dimensional time-domain signal vector x of length N is a one-dimensional frequency-domain signal vector X of the same length. Each of x's sample points, denoted as x_n, can be used in the following equation.

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N(n,k)$$

$\omega_N(n,k)$ is the *twiddle factor*, which are the coefficients of the transform expressed as $\omega_N(n,k) = exp(-\frac{2\pi j}{N}nk)$. On the other hand, the inverse DFT of a one-dimensional frequency-domain signal vector X is a one-dimensional time-domain signal vector x of the same length. Each of X's sample points, denoted by X_n, can be used in the following equation.

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k \omega_N(-\ n,k)$$

The most common way to compute for the DFT is by using the fast Fourier transform (FFT). It is a relatively old algorithm that takes advantage of the symmetry in DFT by following a divide-and-conquer approach and reusing the twiddle factors in mirrored sample points. Within the class of FFT algorithms, the Cooley-Tukey FFT algorithm is the most used one because of its implementation simplicity and numerical stability. It splits a signal into even and odd indices, and are merged at the end to determine the DFT. Your FORTRAN library actually uses an implementation of this, which divides your 1D signal into a 2D one! However, you were also acquainted with a specific variation of it named the *split-radix FFT algorithm*, which was formulated as early as the late 1960s.



The split-radix FFT algorithm is similar to the popular Cooley-Tukey FFT algorithm except that the odd part is also further split into two more alternating elements. In summary, the DFT equation above is rewritten as follows.

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n}\omega_{\frac{N}{2}}(n,k) + \omega_N(1,\ k)\sum_{n=0}^{\frac{N}{4}-1} x_{4n+1}\omega_{\frac{N}{4}}(n,k) + \omega_N(1,\ 3k)\sum_{n=0}^{\frac{N}{4}-1} x_{4n+3}\omega_{\frac{N}{4}}(n,k)$$

$$X_k = X_{even} + \omega_N(1,k)X_{odd1} + \omega_N(1,3k)X_{odd3}$$

The first summation denotes the terms with even indices starting at 0, the second denotes the terms with indices starting at 1 skipping by 4, and the last denotes the terms with indices starting at 3 skipping by 4. After the splitting stage, we can combine these three terms together by reusing more twiddle factors, looking at symmetry, and assembling the DFT by quarters instead by computing these four signal slices below for k between 0 and $\frac{N}{4} - 1$ inclusive.

$$X_k = X_{even,k} + (\omega_N(1,k)X_{odd1,k} + \omega_N(1,3k)X_{odd3,k})$$

$$X_{k+\frac{N}{4}} = X_{even,k+\frac{N}{4}} - j(\omega_N(1,k)X_{odd1,k} - \omega_N(1,3k)X_{odd3,k})$$

$$X_{k+\frac{N}{2}} = X_{even,k} - (\omega_N(1,k)X_{odd1,k} + \omega_N(1,3k)X_{odd3,k})$$

$$X_{k+\frac{3N}{4}} = X_{even,k+\frac{N}{4}} + j(\omega_N(1,k)X_{odd1,k} - \omega_N(1,3k)X_{odd3,k})$$

In summary, $X_{even,k}$ denotes the kth element of the first summation, $X_{odd1,k}$ the kth element of the second summation, and $X_{odd3,k}$ the kth element of the third summation. Finally, the DFT can now be expressed as a concatenation of these four signal slices shown as a bad abuse of math syntax.

$$X = [X_k, X_{k+\frac{N}{4}}, X_{k+\frac{N}{2}}, X_{k+\frac{3N}{4}}]$$

This relatively simple re-expression purportedly reduces the number of additions and multiplications to its lowest. Note that both the Cooley-Tukey and split-radix algorithms only work for signals whose sizes are a power of two.

With this information, you have decided to implement this split-radix FFT algorithm instead for a change. You will still want to deal with signals whose sizes are not a power of two. For simplicity, you have decided to append zeros at the ends of these signals such that the signals become a size of a power of two. Also, the time-domain signals will always be expressed as integers, and the resulting complex-number frequency-domain transformations will always be rounded down to six decimal places.

As you see a long list of porting tasks awaits, you have used the SciPy forums to find a partner that can help you verify that your FFT is correct and it works.

Alternate Universe 1: Found a partner!
Somebody noticed your post and was interested to work with you. Now, you have tasked your partner to implement the inverse split-radix FFT, which will reduce your porting job in this specific library in half, and also have a test file associated with your module.

Alternate Universe 2: No partner!
Unfortunately, nobody read your post and are now left with implementing both the FFT and IFFT modules. However, since you do not want to dwell on this task for too long, you have prioritized implementing the FFT.

Input

The input to the FFT module starts with a number T on a line indicating the number of time-domain signals to transform. Then, the next T lines denote each signal to be transformed. Each line consists of a string of integers, with the first number N indicating the number of sample points that will follow, and the next N integers indicating the sample points x_i.

Output

The output from the FFT module starts with the number T on a line denoting the number of frequency-domain signals that were transformed. Then, the next T lines denote each signal that was transformed. Each line consists of a string of complex numbers, with the first positive integer N indicating the length of the original time-domain signal, the second positive integer K indicating number of sample points that will follow, and the next K complex numbers indicating the sample points X_i.

The complex numbers X_i should be denoted in the format `+r+cj` where r is the real part and c is the imaginary part. Both of these parts should have signs and are expressed rounded down (i.e. floored) to 10^{-6} tolerance, meaning that there should be exactly six digits after their respective decimal points. Zeros can be represented by both a positive or a negative sign. Refer to the sample output for specific examples.

Example

Input

```
5
1 -4
2 1 1
4 1 0 1 0
5 1 2 3 4 5
3 -8 -7 5
```

Output

```
5
1 1 -4.000000+0.000000j
2 2 +2.000000+0.000000j +0.000000-0.000000j
4 4 +2.000000+0.000000j +0.000000-0.000000j +2.000000+0.000000j +0.000000-0.000000j
5 8 +15.000000+0.000000j -5.414214-7.242641j +3.000000+2.000000j -2.585786-1.242641j +3.000000+0.000000j -2.585786+1.242641j +3.000000-2.000000j -5.414214+7.242641j
3 4 -10.000000+0.000000j -13.000000+7.000000j +4.000000+0.000000j -13.000000-7.000000j
```

Additional Description/Requirements

The module will only accept these range or set of variables:

$T \leq 100$

$x_i \in Z,\ |x_i| \leq 10000$

$X_i \in C$

$1 \leq N \leq 10000$

$K \in 2^d,\ d \in N^0$

As this is a partner task, the person tasked with writing the inverse FFT algorithm should create a module whose input and output are reversed in relation to the FFT module. That is, the input to this module follows the output specifications, and the output follows the input specifications. The output of this module should be exactly the same as the input to the FFT module, and each point should be rounded half to even (i.e banker's rounding) to the nearest integer. Sample input and output to this inverse FFT module is shown below.

Input

```
5
1 1 -4.000000+0.000000j
2 2 +2.000000+0.000000j +0.000000-0.000000j
4 4 +2.000000+0.000000j +0.000000-0.000000j +2.000000+0.000000j
+0.000000-0.000000j
5 8 +15.000000+0.000000j -5.414214-7.242641j +3.000000+2.000000j
-2.585786-1.242641j +3.000000+0.000000j -2.585786+1.242641j +3.000000-2.000000j
-5.414214+7.242641j
3 4 -10.000000+0.000000j -13.000000+7.000000j +4.000000+0.000000j
-13.000000-7.000000j
```

Output

```
5
1 -4
2 1 1
4 1 0 1 0
5 1 2 3 4 5
3 -8 -7 5
```

Both modules should accept input through standard input, either through the command line, or through input redirection. Conversely, both should print output to the standard output.

Note that the FFT algorithms work only with signals whose sizes are a power of two. Hence, you need to append zeros at the ends of these signals such that the signals become a size of a power of two before processing. On the other hand, you can assume that the IFFT module will only have to process signals whose sizes are always a power of two.

In homage to SciPy, you have decided to name the future library NumPy, as it is a combination of the Numeric and Numarray libraries. Since you are still in the process of making the library, you decided not to use any of these libraries (i.e. it is forbidden to import NumPy, Numeric, or Numarray to your module) and only use standard libraries if you decided to write your module in Python. At this point, you have been doing a lot of work porting them to NumPy. Hence, it would be a waste of time in the meantime to write a thorough documentation for your module. However, because you believe that you can easily forget things and there may be a chance in the future when you revisit the code, you try your best to write self-documenting code. This means that, among other things, you write your variables throughout your code to clearly label their purposes and add comments to code that can be hard to understand.

You still retained the text logs that you have written together with your 1990s library. Attached with this paper is a file outlining the split-radix FFT algorithm.

You have accidentally discovered through close friends that Google has been using a cloud storage platform internally named Google Drive early this year and are planning to release it in 2012. You have used the service incognito, and have set-up a Google Form (which will be released in 2014) where you will be uploading the modules (as a source code file; in TXT if the system does not support the file extension).

The one who made the FFT module will upload the FFT module and acknowledge their partner on the submission notes (by writing their student number, if any). The one who made the IFFT module will also do the same for their IFFT module.

Grading Rubric

Alternate Universe 1: Found a partner!

FFT module writer

5% Input and output handling for FFT
30% Algorithm to split signals in FFT
30% Algorithm to merge FFT calculations
10% Self-documenting code
10% Acknowledgement of IFFT partner**

IFFT module writer

5% Input and output handling for IFFT
30% Algorithm to split signals in IFFT
30% Algorithm to merge IFFT calculations
10% Self-documenting code
10% Acknowledgement of FFT partner**

Joint

15% Input-output compatibility among the two modules

** 5% if unacknowledged without or non-solid explanation, 0% if unacknowledged with solid explanation

Alternate Universe 2: No partner!

5% Input and output handling for FFT
5% Input and output handling for IFFT
30% Algorithm to split signals in FFT
30% Algorithm to merge FFT calculations
5% Algorithm to split signals in IFFT
5% Algorithm to merge IFFT calculations
10% Input-output compatibility among the two modules
10% Self-documenting code