# ThinkDSP

This notebook contains solutions to exercises in Chapter 7: Discrete Fourier Transform



In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot

import numpy as np

import warnings
warnings.filterwarnings('ignore')

PI2 = 2 * np.pi

np.set_printoptions(precision=3, suppress=True)
%matplotlib inline
```

**Exercise:** In this chapter, I showed how we can express the DFT and inverse DFT as matrix multiplications. These operations take time proportional to $N^2$, where $N$ is the length of the wave array. That is fast enough for many applications, but there is a faster algorithm, the Fast Fourier Transform (FFT), which takes time proportional to $N \log N$.

The key to the FFT is the Danielson-Lanczos lemma:

$$DFT(y)[n] = DFT(e)[n] + exp(-2\pi i n/N) DFT(o)[n]$$

Where $DFT(y)[n]$ is the $n$th element of the DFT of $y$; $e$ is a wave array containing the even elements of $y$, and $o$ contains the odd elements of $y$.

This lemma suggests a recursive algorithm for the DFT:

1. Given a wave array, $y$, split it into its even elements, $e$, and its odd elements, $o$.
2. Compute the DFT of $e$ and $o$ by making recursive calls.
3. Compute $DFT(y)$ for each value of $n$ using the Danielson-Lanczos lemma.

For the base case of this recursion, you could wait until the length of $y$ is 1. In that case, $DFT(y) = y$. Or if the length of $y$ is sufficiently small, you could compute its DFT by matrix multiplication, possibly using a precomputed matrix.

Hint: I suggest you implement this algorithm incrementally by starting with a version that is not truly recursive. In Step 2, instead of making a recursive call, use `dft` or `np.fft.fft`. Get Step 3 working, and confirm that the results are consistent with the other implementations. Then add a base case and confirm that it works. Finally, replace Step 2 with recursive calls.

One more hint: Remember that the DFT is periodic; you might find `np.tile` useful.

You can read more about the FFT at [https://en.wikipedia.org/wiki/Fast_Fourier_transform (https://en.wikipedia.org/wiki/Fast_Fourier_transform)](https://en.wikipedia.org/wiki/Fast_Fourier_transform).

As the test case, I'll start with a small real signal and compute its FFT:

In [2]:
```
ys = [-0.5, 0.1, 0.7, -0.1]
hs = np.fft.fft(ys)
print(hs)
```

```
[ 0.2+0.j  -1.2-0.2j  0.2+0.j  -1.2+0.2j]
```

Here's my implementation of DFT from the book:

In [3]:
```
def dft(ys):
    N = len(ys)
    ts = np.arange(N) / N
    freqs = np.arange(N)
    args = np.outer(ts, freqs)
    M = np.exp(1j * PI2 * args)
    amps = M.conj().transpose().dot(ys)
    return amps
```

We can confirm that this implementation gets the same result.

In [4]:
```
hs2 = dft(ys)
print(sum(abs(hs - hs2)))
```

```
5.86477584677e-16
```

As a step toward making a recursive FFT, I'll start with a version that splits the input array and uses np.fft.fft to compute the FFT of the halves.

In [5]:
```
def fft_norec(ys):
    N = len(ys)
    He = np.fft.fft(ys[::2])
    Ho = np.fft.fft(ys[1::2])

    ns = np.arange(N)
    W = np.exp(-1j * PI2 * ns / N)

    return np.tile(He, 2) + W * np.tile(Ho, 2)
```

And we get the same results:

In [6]:
```
hs3 = fft_norec(ys)
print(sum(abs(hs - hs3)))
```

```
0.0
```

Finally, we can replace `np.fft.fft` with recursive calls, and add a base case:

In [7]:
```
def fft(ys):
    N = len(ys)
    if N == 1:
        return ys

    He = fft(ys[::2])
    Ho = fft(ys[1::2])

    ns = np.arange(N)
    W = np.exp(-1j * PI2 * ns / N)

    return np.tile(He, 2) + W * np.tile(Ho, 2)
```

And we get the same results:

In [8]:
```
hs4 = fft(ys)
print(sum(abs(hs - hs4)))
```

```
1.66533453694e-16
```

This implementation of FFT takes time proportional to $n \log n$. It also takes space proportional to $n \log n$, and it wastes some time making and copying arrays. It can be improved to run "in place"; in that case, it requires no additional space, and spends less time on overhead.

In [ ]:
```

```