```
In [ ]: # Initialize Otter
        import otter
        grader = otter.Notebook("lab3.ipynb")
```

1 EE 120 Lab 3: Practical Fourier Analysis

Signals and Systems at UC Berkeley

Acknowledgements:

- **Fall 2018** (Beta): Dominic Carrano
- **Spring 2019** (v1.0): Dominic Carrano, Babak Ayazifar

- **Fall 2019** (v2.0): Dominic Carrano

- **Spring 2020** (v3.0): Dominic Carrano
- **Fall 2020** (v3.1): Anmol Parande
- **Spring 2022** (v4.0): Drake Lin
- **Fall 2022** (v4.1): Drake Lin
- **Spring 2023** (v5.0): Gavin Zhang

```
In [1]: from scipy import signal
        from scipy.io.wavfile import read
        from IPython.display import Audio
        from math import ceil, floor
        import lab3_helper
        import numpy as np
        import timeit, time
        import matplotlib.pyplot as plt
        %matplotlib inline
```

2 Background

2.1 Recap: Why Fourier Analysis Matters

You've already seen that complex exponentials are a key player in signals and systems for two reasons. In particular, a complex exponential at frequency ω: 1. (**The signals perspective**) Corresponds to a wave or oscillator with frequency ω. 2. (**The systems perspective**) Is an eigenfunction of any LTI system, with the (typically, complex) eigenvalue given by the system's frequency response evaluated at ω.

Perhaps unsurprisingly, then, *Fourier analysis*, which is a set of techniques for decomposing signals as linear combinations of complex exponentials, is incredibly useful. We can make use of it to study the frequency composition of a signal as suggested by (1), or analyze and even implement LTI systems as suggested by (2). In this lab, we'll work through an example of how both are done in practice.

2.2 Fourier Analysis on a Computer

We have nearly endless computational power at our fingertips today, so it's only natural that we seek a way to add Fourier analysis to our ever-expanding Python signals and systems toolkit. You've already seen in the first couple labs that all signals we work with on computers are discrete and have finite duration N, enabling a convenient representation as a length N numpy array. Outside these N samples, we assume the signal is zero. **The question, then, is how do we digitally take a Fourier Transform of the signal, given this array representation?**

To answer this question, let's take a look at the four main flavors available to us—summarized in the table below—and see if we can adapt any to fit our needs. As a reminder, FS is an abbreviation for Fourier Series, FT for Fourier Transform.

	Discrete time (DT)	Continuous Time (CT)
Periodic	DTFS	CTFS
Not necessarily periodic	DTFT	CTFT

Since we operate in DT on a computer, working with sets of bits, the CTFT and CTFS are both ruled out. Our signals are typically aperiodic, so the most natural choice seems like the DTFT, which will be covered in a few weeks.

A general rule of thumb regarding the relationship between the time domain and the frequency domain is that impulsive (discrete values) in one domain $\iff$ periodic in the other domain. DTFS takes advantage of the fact that an impulsive, periodic signal in the time domain results in an impulsive, periodic signal in the frequency domain. However, for discrete time (i.e. impulsive) signals that are not periodic, their frequency components (i.e. transform) will be periodic (between the range of 0 to 2 π) but also continuous. This means that the frequency domain of an aperiodic discrete time signal is a continuous spectrum, and therefore has an infinite number of values, which is reflected in the equations for the DTFT:

$$x(n) = \frac{1}{2\pi}\int_0^{2\pi} X(\omega)e^{i\omega n}dw \iff X(w) = \sum_{n=-\infty}^{\infty} x(n)e^{-iwn}$$

For periodic discrete time signals, we can represent both the signal and its DTFS coefficients as arrays. For a N-periodic signal x (which has N DTFS coefficients):

$$x(n) = [x(0), x(1), x(2), x(3), ..., x(N-1)]$$

$$X_k = [X_0, X_1, X_2, X_3, ..., X_{N-1}]$$

which is representable using numpy arrays. However, for aperiodic signals, there are an uncountably infinite number of Fourier Coefficients, since $X(\omega)$, the DTFT of $x(n)$, can take in any ω from $[0, 2\pi)$.

As a result, in practice, we use the *Discrete Fourier Transform (DFT)*

$$X[k] = \sum_{k=0}^{N-1} x(n) e^{-i\frac{2\pi}{N}kn}$$

which corresponds to *sampling* the DTFT at N evenly spaced frequencies in the range $[0, 2\pi)$. This idea that the DFT corresponds to sampling the DTFT is clear from the formulae: the DTFT analysis equation (for a duration N signal) evaluated at the frequency $\omega = 2\pi k/N$ yields the expression for the kth DFT coefficient. As a reminder, there is no operational difference between the DFT and the DTFS as one is simply the scaled version of the other.

In total, to compute the signal's *N-point DFT*, we calculate N samples of the DTFT

$$\{X[k]\}_{k=0,1,2,...,N-1} = X(\omega)|_{\omega=\frac{2\pi}{N}k,\quad k=0,1,2,...,N-1} = \left\{X(0), X\left(\frac{2\pi}{N}\right), X\left(2\cdot\frac{2\pi}{N}\right), ..., X\left((N-1)\cdot\frac{2\pi}{N}\right)\right\}$$

and use this as our computer-ready Fourier Transform. We can't get the entire DTFT, unfortunately, but we can increase N by *zero-padding*, or appending some of the implicit zeros to the end of our signal, until the number of samples we have of the DTFT is sufficient (this idea is explored in Discussion 8 Question 3). **This is how we can use the Fourier Transform in practice!**

2.3 We're done, right?

It seems like our problem is solved: if we want to take the Fourier Transform of a signal in Python, we can just compute its N-point DFT through the analysis equation. As you'll soon see, however, directly using the analysis equation is limitingly slow, yielding an $O(N^2)$ algorithm.

In this lab, we'll see how clever manipulation of the DFT analysis equation to eliminate redundant computations can bring that down to $O(N \log N)$, and just how much of a difference that makes. The latter algorithm is known as the *Fast Fourier Transform (FFT)*. It is **not** a different transform, simply a particular algorithm for implementing the DFT.

3 Setup

Run the cell below to load in a snippet of *Everything Goes On* by Porter Robinson and League of Legends, which we'll use for benchmarking.

```
In [100]: fs, song_clip = read("porter_intro.wav")
          song_clip = np.swapaxes(song_clip, 0, 1)
          Audio(song_clip, rate=fs)
```

```
Out[100]: <IPython.lib.display.Audio object>
```

When we listen to the song, we hear both the left and right audio channels superimposed. The variable `song` is actually a 2 row matrix, with each row a distinct signal for each audio channel. We'll average these together to obtain a single 1D signal for analysis.

```
In [101]: # Note: The "10*fs:" crop grabs samples starting 10 seconds in
          left = song_clip[0, 10*fs:]
          right = song_clip[1, 10*fs:]
          song = (left + right) / 2
```

4 Q1: The Naive DFT [Optional]

To motivate the need for a *fast* Fourier Transform, we'll compute the DFT of a clip of song, a bottleneck operation in Shazam, which calculates and compares DFTs of songs to identify them.

Before we can do that, though, we need to actually implement the DFT itself.

4.1 Q1a: DFT Code [Optional]

Your job is to fill in the function `dft` below according to the docstring.

We can compute an N-point DFT using the DFT analysis equation

$$X[k] = \sum_{n=0}^{N-1} x(n) e^{-i\frac{2\pi}{N}nk}$$

for $k \in \{0, 1, 2, ..., N-1\}$. We collect the coefficients $X[0], X[1], ..., X[N-1]$ in a numpy array and call this sequence the N-point DFT of x.

Do not use any vectorization, we're saving that for the next question. Implement this in the simplest way possible, using for loops.

```
In [2]: def dft(x):
            """
            Compute the N-point Discrete Fourier Transform of x based on the analysis equation, where N
            the length of the numpy array x.

            In your array X that you output, X[k], the kth element (k = 0, 1, ..., N-1)
            should be the value of the kth DFT coefficient.

            **Hint 1:** The function call `np.exp(z)` returns $e^z$, where $z$ is any number, real or c
            **Hint 2:** In Python, `1j` is used to represent the complex number $\sqrt{-1}$,
                        typically denoted as $i$ or $j$ depeding on the notation being used.
            """
            N = len(x)
            X = np.zeros(N, dtype=np.complex128) # output array - have to declare as complex so it's no

            # BEGIN SOLUTION
            for k in range(N):
                for n in range(N):
                    X[k] += x[n] * np.exp(-1j * 2 * np.pi * n * k / N)
            # END SOLUTION

            return X
```

Let's do a couple quick sanity checks on `dft` to make sure it returns the correct results. Run the cells below to check your work. Make sure all tests pass before you move on.

```
In [ ]: grader.check("q1a")
```

4.2 Q1b: DFT of song Clip [Optional]

Now to put our DFT function to the test! Typically, Shazam would split the signal into blocks, compute FFTs of those blocks, and compare against a database to match the song. Let's see how long it takes to compute the DFT for the first: 1. 512 samples. 2. 1024 samples. 3. 2048 samples. 4. 4096 samples.

Will our DFT be fast enough to handle Shazam's real-time computation needs? Run the cell to find out! (**Note: This will probably take a few minutes to run**)

```
In [104]: song_512 = song[:512]
          song_1024 = song[:1024]
```

```
song_2048 = song[:2048]
song_4096 = song[:4096]

t0 = time.time() # initial time
dft(song_512)
t1 = time.time()
print("Took {0} sec for 512-point DFT".format(round(t1 - t0, 5)))

dft(song_1024)
t2 = time.time()
print("Took {0} sec for 1024-point DFT".format(round(t2 - t1, 5)))

dft(song_2048)
t3 = time.time()
print("Took {0} sec for 2048-point DFT".format(round(t3 - t2, 5)))

dft(song_4096)
t4 = time.time()
print("Took {0} sec for 4096-point DFT".format(round(t4 - t3, 5)))
```

```
Took 0.96318 sec for 512-point DFT
Took 4.06003 sec for 1024-point DFT
Took 16.44318 sec for 2048-point DFT
Took 71.75813 sec for 4096-point DFT
```

4.2.1 Is it practical? [Optional]

We'll use the 4096-point DFT as the standard for benchmarking. We couldn't find any online resources indicating what DFT block size the production version of Shazam uses. However, we dug through the source code of Dejavu, an open source version, and found that they use 4096 (see the `DEFAULT_WINDOW_SIZE` definition in this file), so this seems like a good size to test with.

Q: Shazam usually needs 3-5 seconds of data before confidently making a decision about what song is being played. Our data was sampled at 44.1 kHz, so we'd have to separately compute around fifty 4096-point DFTs before being done (since $4096 \times 50 \approx 44100 \times 5$).

Extrapolating from the data you got in the cell above for 4096, roughly how long it take to compute those fifty 4096-point DFTs, so Shazam can identify our song?

A: (TODO)

A: (TODO) It took ~67 seconds to compute a single 4096-point DFT, so it would take about 3350 seconds (almost an hour!) for Shazam to finish classifying our song.

Q: If Shazam took as long to classify songs as you found in the question immediately preceding this one, would you still use it?

A: (TODO)

A: (TODO) No! That's way to slow.

Can't we just use 512? It's tempting to think we can just use a 512-point DFT instead of 4096-point one, since it's faster. After all, why would Dejavu or Shazam bother to split songs into 4096-sample chunks instead of 512-sample chunks?

In many applications that rely on DFTs, Shazam included, the DFT length has a serious impact on functionality. EE 123 gives a more in-depth treatment of this problem, but the upshot is that shorter DFTs mean less time-domain information per block of samples, and past a certain point, you don't have enough data to do anything meaningful.

As a brief example, let's consider song, since that's what we care about from a Shazam perspective. Typically, songs are sampled at 44.1 kHz or 48 kHz (for the song we have here, it was 44.1). 512 datapoints per DFT represents only $512/44100 \approx .012$ seconds of data per block, whereas 4096 gives us $4096/44100 \approx .1$ seconds of data per block, which is a much better tradeoff.

4.3 Q1c: DFT Computational Analysis [Optional]

If you'd like to fully understand why the DFT runtime is $O(N^2)$, take a crack at answering the questions below. This part can be done entirely by inspecting the DFT analysis equation, and does not require a working implementation of the previous parts. You should answer these with respect to the analysis equation, not your specific implementation, although the two should match.

Q: How many complex multipications are performed by `dft` in computing a *single* DFT coefficient for a size N input?

Note: Multiplying two complex numbers counts as a single multiplication. For example, $x(1) \cdot e^{-i\frac{2\pi}{N}}$ counts as one complex multiplication, as each element of $x(n)$ is a complex number, in general.

A: (TODO)

A: (TODO) It took ~.0001 seconds to compute a single 4096-point DFT, so it would take about .005 seconds for Shazam to finish classifying our song with our upgraded version. It's called the *fast* Fourier Transform for a reason!

Q: How many complex additions (adding two complex numbers) are performed by `dft` in computing a **single** DFT coefficient for a size N input?

A: (TODO)

A: (TODO) $N-1$, one per term in the summation minus one for the fact that we add the first two terms with a single addition operation. However, in the solution given here, we begin by zeroing out our array and then adding up all the terms, so technically N complex additions are performed by the given solution code for `dft`. You could achieve $N-1$ instead by doing the first multiplication and then writing that to the array to start rather than explicitly add it to zero. For a vectorized approach to get a performance boost (CS 61C anyone?), create a matrix where each column (or row) holds the multiplication results, and then sum the columns (or rows) to obtain your coefficients. Either answer is acceptable, as long as it's consistent with your implementation. Ultimately, the extra operation doesn't make any noteworthy difference for large enough N, which is all we really care about with asymptotic analysis.

Q: Adding up your previous two answers, how many total complex number operations (complex additions and complex multiplications together) are performed in calculating a single DFT coefficient for an input of size N?

A: (TODO)

A: (TODO) $N + (N-1) = 2N-1$.

Q: Multiplying your previous by the number of coefficients, N, what is the total number of complex number operations performed in calculating the entire N-point DFT?

A: (TODO)

A: (TODO) $N \times (2N-1) = 2N^2 - N$.

Q: Expressing your previous answer in Big-O notation, what is the asymptotic runtime of the function `dft` for calculation of the DFT via the analysis equation?

A: (TODO)

A: (TODO) $O(N^2)$. So, asymptotically, the DFT has a runtime that is a quadratic function of its input size. It's not exponential, but we can certainly do better.

5 Q2: Matrix-Vector DFT

Directly implementing the DFT as a set of for loops much too slow for practical applications. Fortunately, our for loops are just a series of dot products, and we can implement the DFT as a matrix-vector multiplication.

The DFT analysis equation can be written as the matrix-vector equation $\vec{X} = M\vec{x}$, where:

$$\underbrace{\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ \vdots \\ X[N-1] \end{bmatrix}}_{\vec{X}} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-i\frac{2\pi}{N}} & e^{-i\frac{2\pi}{N}2} & \dots & e^{-i\frac{2\pi}{N}(N-1)} \\ 1 & e^{-i\frac{2\pi}{N}2} & e^{-i\frac{2\pi}{N}4} & \dots & e^{-i\frac{2\pi}{N}2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-i\frac{2\pi}{N}(N-1)} & e^{-i\frac{2\pi}{N}2(N-1)} & \dots & e^{-i\frac{2\pi}{N}(N-1)(N-1)} \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ \vdots \\ x(N-1) \end{bmatrix}}_{\vec{x}}$$

Note that the entry of M at row n, column k is $e^{-i\frac{2\pi}{N}nk}$, where $n, k \in \{0, 1, 2, ..., N-1\}$.

5.1 Q2a: Matrix-Vector DFT Code

Implement `dft_mtx_vec` below, which constructs the $N \times N$ DFT matrix M (where N is the length of the input vector) and returns the matrix-vector product $M\vec{x}$, which yields the result of applying the DFT analysis equation.

Hint: After determining N, the code `n = np.arange(N); k = n.reshape((N, 1)); idx = n * k` will generate an $N \times N$ matrix `idx` such that `idx[m][l] = m * l`. See how you can use this matrix to then easily compute the DFT matrix.

```
In [3]: def dft_matrix_vector(x):
            # BEGIN SOLUTION
            N = len(x)
            n = np.arange(N)
            k = n.reshape((N, 1))
            M = np.exp(-1j * 2 * np.pi * k * n / N)
            return M.dot(x)
            # END SOLUTION
```

Let's run the same sanity checks.

```
In [ ]: grader.check("q2a")
```

5.2 Q2b: Benchmarking the Matrix-Vector DFT

Same setup: we want to see how long it takes to compute 512, 1024, 2048, and 4096-point DFTs of our song, now using our improved method. Run the cell below to benchmark `dft_mtx_vec`.

```
In [67]: song_512 = song[:512]
         song_1024 = song[:1024]
```

```
song_2048 = song[:2048]
song_4096 = song[:4096]

t0 = time.time() # initial time
dft_matrix_vector(song_512)
t1 = time.time()
print("Took {0} sec for 512-point DFT".format(round(t1 - t0, 5)))

dft_matrix_vector(song_1024)
t2 = time.time()
print("Took {0} sec for 1024-point DFT".format(round(t2 - t1, 5)))

dft_matrix_vector(song_2048)
t3 = time.time()
print("Took {0} sec for 2048-point DFT".format(round(t3 - t2, 5)))

dft_matrix_vector(song_4096)
t4 = time.time()
print("Took {0} sec for 4096-point DFT".format(round(t4 - t3, 5)))
```

```
Took 0.02131 sec for 512-point DFT
Took 0.07646 sec for 1024-point DFT
Took 0.24784 sec for 2048-point DFT
Took 1.4361 sec for 4096-point DFT
```

5.2.1 Is it practical?

Q: Extrapolating from the data you got in the cell above for 4096, roughly how long it take to compute the fifty 4096-point DFTs needed for Shazam to identify our song?

A: (TODO)

A: (TODO) It took ~.75 seconds to compute a single 4096-point DFT, so it would take about 35-40 seconds for Shazam to finish classifying our song with our upgraded version. Significantly better than before, but even 35 seconds is still a long time to wait for most users.

We're doing much better now, but can still improve some more.

6 Q3: The Fast Fourier Transform (FFT)

Matrix-vector DFT does pretty well, but we can still do better.

The strategy in developing a faster method, which is formally called the *Decimation in Time Fast Fourier Transform* algorithm, will be to split up the signal $x(n)$ into successively smaller signals, take DFTs of those smaller signals, and build up the full result from these smaller parts. If you've taken CS 170, you've heard this (fittingly) referred to as a *divide-and-conquer* algorithm.

6.0.1 Powers of 2

Throughout this question, we'll assume that the signal duration, N, is a power of two so that we can keep splitting the signal in half until we have N separate duration 1 signals. If not, we can just pad on implicit zeros until the length is a power of two (which is what signal processing practitioners often do anyways), so this is a fine assumption.

6.1 Dividing

Our goal is to divide the task of computing the N-point DFT of $x(n)$ into computing two $N/2$-point DFTs: - One over the even-indexed entries $x(0), x(2), x(4), ..., x(N-2)$. - One over the odd-indexed entries $x(1), x(3), x(5), ..., x(N-1)$.

Remember, N is a power of two, so both it and $N-2$ are even. This means $N-1$, the index of the last signal value (since we start counting at zero), is odd. We can use this division to split up the analysis equation as

$$X[k] = \sum_{n=0}^{N-1} x(n) e^{-i\frac{2\pi}{N}nk} = \sum_{n \text{ even}} x(n) e^{-i\frac{2\pi}{N}nk} + \sum_{n \text{ odd}} x(n) e^{-i\frac{2\pi}{N}nk}$$

which will make things a bit easier to work with.

6.2 Conquering

Writing n even and n odd as our summation indices is a bit annoying. Let's reindex things so we have contiguous indices rather than ones with gaps (e.g., $0, 2, 4, ...$) in them.

To generate a contiguous set of $N/2$ integers to sum over for the evens, note that

$$n = 0, 2, 4, ..., N-2 = 2 \cdot \left(0, 1, 2, ..., \frac{N-2}{2}\right) = 2 \cdot \underbrace{\left(0, 1, 2, ..., \frac{N}{2} - 1\right)}_{m}$$

and $N/2$ is an integer since N is a power of two.

Similarly, to generate the odds, we have

$$n = 1, 3, 5, ..., N-1 = (0, 2, 4, ..., N-2) + 1 = 2 \cdot \underbrace{\left(0, 1, 2, ..., \frac{N}{2} - 1\right)}_{m} + 1$$

This provides a key insight: in both cases, if we have a summation running over $m = 0, 1, 2, ..., \frac{N}{2} - 1$, which would correspond to an $N/2$-point DFT, to generate even values of n, we can use $n = 2m$. Similarly, to generate the odds, we use $n = 2m + 1$.

Equipped with this, we can write

$$\begin{aligned} X[k] &= \sum_{m=0}^{N/2-1} x(2m) e^{-i\frac{2\pi}{N}(2m)k} + \sum_{m=0}^{N/2-1} x(2m+1) e^{-i\frac{2\pi}{N}(2m+1)k} && (1) \\ &= \left[\sum_{m=0}^{N/2-1} x(2m) e^{-i\frac{2\pi}{N/2}mk}\right] + e^{-i\frac{2\pi}{N}k} \left[\sum_{m=0}^{N/2-1} x(2m+1) e^{-i\frac{2\pi}{N/2}mk}\right] && (2) \\ & && (3) \end{aligned}$$

Defining

$$a(n) = [x(0), x(2), x(4), ..., x(N-2)], \quad b(n) = [x(1), x(3), x(5), ..., x(N-1)]$$

and $A[k], B[k]$ as their $N/2$-point DFTs, respectively, we see that

$$\boxed{X[k] = A[k] + e^{-i\frac{2\pi}{N}k} B[k], \qquad k = 0, 1, 2, ..., N-1}$$

We can now compute the N-point DFT X from the two $N/2$-point DFTs A, B using this formula!

This provides a recursive method for computing the DFT: - Construct a, b. - Recursively compute their $N/2$-point DFTs A, B. - For the kth DFT coefficient, $k = 0$ to $k = N - 1$, compute $X[k] = A[k] + e^{-i\frac{2\pi}{N}k} B[k]$.

Important implementation detail: When indexing into A, B, the index k must be interpreted $\text{mod} N/2$ since these are $N/2$-point DFTs, not N-point DFTs.

6.3 Q3a: FFT Code

Implement the function `my_fft` according to the docstring. **If the length of the input array `x` is not a power of two, just call `dft_matrix_vector` on it.** You may, but are not required to, use the provided is_power_of_two function.

The FFT algorithm often can often look daunting to students when they learn about it the first time, this lab hopes to show that the its implementation is relatively simple. For reference, the staff solution has only 10 lines of code between the TODO markers.

```
In [4]: def is_power_of_two(n):
            """
            Return true if n is a power of 2, else false, assuming n is a positive integer.
            Adapted from https://stackoverflow.com/a/57027610.
            """
            return n & (n - 1) == 0
```

```
In [5]: def my_fft(x):
            """
            Uses the decimation in time FFT algorithm to compute an N-point DFT of the numpy array x,
            where N is the length of x.

            Hint 1: The FFT is a recursive algorithm, and the base case is N=1,
                    when you only have one signal value and have to compute one DFT coefficient.

                    What does the analysis equation tell you that this one DFT coefficient,
                    X[0], is in terms of the one signal value, x[0]?

            Hint 2: Python's slicing features - check out https://stackoverflow.com/questions/509211/ -
                    be very useful for grabbing the even and odd-indexed components of the signal.
            """
            N = len(x)
            X = np.zeros(N, dtype=np.complex128)

            # BEGIN SOLUTION
            if not is_power_of_two(N):   # input size not a power of two, use dft
                return dft_matrix_vector(x)
            if N == 1:                   # base case - DFT of a single value is that value
                return x
            even, odd = x[::2], x[1::2]  # even/odd split + size N/2 FFTs
            E, O = my_fft(even), my_fft(odd)
            for k in range(N):
                X[k] = E[k % (N // 2)] + np.exp(-2 * np.pi * 1j * k / N) * O[k % (N // 2)]

            # END SOLUTION

            return X
```

Now for some sanity tests.

If you're failing any tests: remember, $A[k]$ and $B[k]$ only have $N/2$ elements, but we use them to compute a total of N DFT coefficients. We already discussed above how to deal with this issue.

```
In [ ]: grader.check("q3a")
```

6.4 Q3b: Benchmarking our Homemade FFT

Let's see how our from-scratch FFT does. For full credit, it should blow your Q1 implementation out of the water (the runtimes won't be close), and be faster than your Q2 implementation.

```
In [75]: song_512 = song[:512]
         song_1024 = song[:1024]
         song_2048 = song[:2048]
         song_4096 = song[:4096]

         t0 = time.time() # initial time
         my_fft(song_512)
         t1 = time.time()
         print("Took {0} sec for 512-point DFT".format(round(t1 - t0, 5)))

         my_fft(song_1024)
         t2 = time.time()
         print("Took {0} sec for 1024-point DFT".format(round(t2 - t1, 5)))

         my_fft(song_2048)
         t3 = time.time()
         print("Took {0} sec for 2048-point DFT".format(round(t3 - t2, 5)))

         my_fft(song_4096)
         t4 = time.time()
         print("Took {0} sec for 4096-point DFT".format(round(t4 - t3, 5)))
```

```
Took 0.01797 sec for 512-point DFT
Took 0.03615 sec for 1024-point DFT
Took 0.0605 sec for 2048-point DFT
Took 0.12671 sec for 4096-point DFT
```

Q: Extrapolating from the data you got in the cell above for 4096, roughly how long it take to compute the fifty 4096-point DFTs needed for Shazam to identify our song?

A: (TODO)

A: (TODO) It took ~.1 seconds to compute a single 4096-point DFT, so it would take about 5 seconds for Shazam to finish classifying our song with our upgraded version. Much better than before!

Let's see how NumPy's FFT function does, since in practice, this is what we'd want to use.

```
In [76]: t0 = time.time()
         np.fft.fft(song_512)
         t1 = time.time()
         print("Took {0} sec for 512-point DFT".format(round(t1 - t0, 10)))

         np.fft.fft(song_1024)
         t2 = time.time()
         print("Took {0} sec for 1024-point DFT".format(round(t2 - t1, 10)))

         np.fft.fft(song_2048)
         t3 = time.time()
         print("Took {0} sec for 2048-point DFT".format(round(t3 - t2, 10)))

         np.fft.fft(song_4096)
         t4 = time.time()
         print("Took {0} sec for 4096-point DFT".format(round(t4 - t3, 10)))
```

```
Took 0.000566721 sec for 512-point DFT
Took 0.0005815029 sec for 1024-point DFT
Took 0.0009977818 sec for 2048-point DFT
Took 0.0003964901 sec for 4096-point DFT
```

Q: Roughly how long it take to compute the fifty 4096-point DFTs needed for Shazam to identify our song, using NumPy's library grade FFT implementation?

A: (TODO)

A: (TODO) It took ~.0001 seconds to compute a single 4096-point DFT, so it would take about .005 seconds for Shazam to finish classifying our song with our upgraded version. It's called the *fast* Fourier Transform for a reason!

7 Q3c: iFFT

The corresponding synthesis equation to the DFT, which converts a signal from the frequency domain to the time domain, is known as the inverse Discrete Fourier Transform (iDFT), and is governed by the equation:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{i\frac{2\pi}{N}nk}$$

Note that the iDFT equation is almost the exact same as the DFT equation, except that the complex exponentials have positive instead of negative exponents, and that each value is scaled by 1/N. The iFFT can also be defined in the exact same way, with an equivalent relationship to the FFT algorithm.

Implement iDFT and iFFT according to the equation above. Feel free to copy paste code from your DFT and FFT implementations and slightly modify it.

Hint: The $\frac{1}{N}$ term becomes problematic for a recursive implementation of iFFT, so try implementing a helper function that calculates

$$x_{helper}(n) = \sum_{k=0}^{N-1} X[k]e^{i\frac{2\pi}{N}nk}$$

using the recursive divide and conquer strategy and using it to generate the iFFT as

$$x(n) = \frac{1}{N}x_{helper}(n)$$

Note that the iFFT has a different input/output than the FFT, so make sure that your variable names match up. Also, be sure not to mutate the input of iFFT.

```
In [42]: def idft_matrix_vector(x):
             # BEGIN SOLUTION
             N = len(x)
             n = np.arange(N)
             k = n.reshape((N, 1))
             M = np.exp(1j * 2 * np.pi * k * n / N)
             return M.dot(x)/N
             # END SOLUTION

         def my_ifft(X):
             N = len(X)
             if not is_power_of_two(N):   # input size not a power of two, use dft
                 return idft_matrix_vector(X)
             else:
                 return my_ifft_helper(X)/N

         def my_ifft_helper(X_freq):
             """
             Calculates N*x_time(n) using the decimation in time iFFT algorithm to compute an N-point i
             where N is the length of X_freq. Assume N is a power of two.

             """
             N = len(X_freq)
             x_time = np.zeros(N, dtype=np.complex128)

             # BEGIN SOLUTION
             if N == 1:                     # base case - DFT of a single value is that value
                 return X_freq
             even, odd = X_freq[::2], X_freq[1::2]  # even/odd split + size N/2 'FFTs
             E, O = my_ifft_helper(even), my_ifft_helper(odd)
```

```
    for k in range(N):
        x_time[k] = E[k % (N // 2)] + np.exp(2 * np.pi * 1j * k / N) * O[k % (N // 2)]

    # END SOLUTION

    return x_time
```

Now for some sanity tests! Firstly, ensure that iFFT(FFT(x)) = x, since that is the whole point of the function. Note that due to finite floating point precision, implemented FFT and iFFT will be slightly off (1 might turn into 1.00000000e+00-5.13169754e-15j).

```
In [34]: x = np.array([1,2,3,4])
         print("The signal x is: " + str(x))
         result = np.round(np.abs(my_ifft(my_fft(x))))
         print("The implemented iFFT of the FFT of x is: " + str(result))
```

```
The signal x is: [1 2 3 4]
The implemented iFFt of the FFT of x is: [1. 2. 3. 4.]
```

```
In [ ]: grader.check("q3c")
```

8 Q3d: Convolution using FFT

One of the reasons why FFT is so useful is because it can be used to compute convolutions quickly. For two N-periodic discrete signals, their periodic convolution is defined as:

$$y(n) = \sum_{m=0}^{N-1} x(m)h(n-m)$$

and is N-periodic. The periodic convolution theorem tells us that:

$$\mathfrak{F}\{y\}_k = \mathfrak{F}\{x\}_k \cdot \mathfrak{F}\{h\}_k$$

where $\mathfrak{F}$ is the DFT operator and $\cdot$ is pointwise multiplication.

Implement periodic convolution using your FFT and iFFT functions, and compare it to the provided periodic convolution function.

Hint: remember that the iDFT is the inverse of the DFT; that is, iDFT(DFT(x)) = x.

```
In [50]: def periodic_conv(x, y):
             assert len(x) == len(y)
             N = len(x)
             out = np.zeros(N, dtype=np.complex128)
             y_rev = np.flip(y)
             y_rev_shift = y_rev_shift = np.roll(y_rev, 1)
             for n in range(N):
                 out[n] = np.dot(x, y_rev_shift)
                 y_rev_shift = np.roll(y_rev_shift, 1)
             return out
```

```
In [51]: def fft_conv(x, h):
             '''
             Compute the periodic convolution of x and h using your implementation of FFT and iFFT. Thi
             '''
             y = None
             # BEGIN SOLUTION
             y = my_ifft(my_fft(x)*my_fft(h))
             # END SOLUTION
             return y
```

Some sanity checks:

```
In [ ]: grader.check("q3d")
```

8.1 (Optional) Q3e: FFT Computational Analysis

We've experimentally seen that the FFT is much faster than the DFT. If you'd like a step by step walkthrough of how to formally prove the algorithm's $O(N \log N)$ runtime, have a go at this question. This type of algorithmic analysis is taught in CS 61B and used throughout CS 170, so it should be accessible with CS 61AB under your belt. It is optional, however, so don't feel the need to fully understand it for EE 120 purposes.

Q: The FFT is a recursive algorithm, dividing its input size in half with each recursive call. When we compute an N-point FFT, assuming N is a power of two, how many *layers* of recursion are there? (Equivalently, how many times can N be divided by two before it becomes 1, the base case?) **Your answer should be a function of N.**

A: (TODO)

A: (TODO) $\log_2(N)$. For example, if $N = 16$, then the recursive calls will be for $N = 8, N = 4, N = 2, N = 1$, resulting in a total of $\log_2(16) = 4$ recursive calls.

Q: We know each recursive call to the FFT produces two more recursive calls (assuming we haven't hit the base case). At the kth layer of the recursion, $k = 0, 1, ..., ?$, where ? is your answer to the previous question, what is the total number of recursive calls? To clarify, $k = 0$ corresponds to the initial call, where we have only one recursive call. At $k = 1$ we have two recursive calls, spawned by the initial call, and so on. **Your answer should be a function of k.**

A: (TODO)

A: (TODO) 2^k, since we start at 1 and double the number at each layer since each function call spawns two "children".

Q: Consider a specific call to FFT at layer k of the recursion. What is the size of the input to FFT at this layer? **Your answer should be a function of k and N.** (Hint: We keep halving our input size each time, going from N to $N/2$ to $N/4$ and so on - how does this generalize for an arbitrary $k = 0, 1, ...$?)

A: (TODO)

A: (TODO) It took ~.1 seconds to compute a single 4096-point DFT, so it would take about 5 seconds for Shazam to finish classifying our song with our upgraded version. Much better than before!

Q: Note that at any given layer, the total amount of work we do (other than the recursion) is a linear function of our input size, since the only other work done is combining the recursion results and computing M DFT coefficients, where M is the input size. So, the total work done (as a function of N), obtained by summing over the work done by all recursive calls, can be computed as:

$$W(N) = \sum_{k=0}^{\text{Number of layers } -1} (\text{Number of recursive calls of size } k) \cdot (\text{Work done by a recursive call of size } k)$$

which simplifies to

$$W(N) = \sum_{k=0}^{a-1} b(k)c(N, k)$$

where a is your answer to the first question, $b(k)$ is your answer to the second question, and $c(N, k)$ is your answer to the third question. Plug in your answers and obtain a simplified expression for $W(N)$.

(Hint: $b(k)c(N, k)$ should cancel nicely to be independent of k.)

A: (TODO)

A: (TODO) Plugging in our answers, we obtain

$$W(N) = \sum_{k=0}^{\log_2(N)-1} 2^k \cdot \frac{N}{2^k} = \sum_{k=0}^{\log_2(N)-1} N = N\log_2(N)$$

Note that the total amount of work done across all recursive calls at layer k is constant, equalling N. The reason is that while the number of calls at each layer exponentially increases (doubling each time), the total amount of work done by each call exponentially decreasies (halving each time), for a net cancellation.

Q: Express your previous answer in Big-O notation, providing the asymptotic runtime of the FFT as a function of its input size N. Does this runtime grow more slowly than that of the naive DFT?

A: (TODO)

A: (TODO) We have the FFT asymptotic runtime as $O(N \log N)$, which isn't just faster than the DFT's $O(N^2)$ by a constant factor; it's asymptotically faster. That is, in the limit as $N \to \infty$, the FFT will become infinitely faster than the naive DFT, since the function $N \log N$ grows more slowly than the function N^2. Even at a small scale, we can already see the disparity!

By doing similar analysis, you can show that periodic convolution takes $O(N^2)$ time since it has to calculate an N-length dot product for each of the N values, whereas using FFT to compute the periodic convolution is essentially just 3 FFT operations, which remains $O(N \log N)$ time.

9 Q4: Efficient Oscilloscope Calibration

In Q1-Q3, you saw the Fourier Transform's use in analyzing signal frequency content, and the necessity of a fast algorithm for doing so. Now, we'll turn our attention to implementing LTI systems (i.e., filters) with it. Specifically, we'll use the FFT to devise an efficient algorithm for cross-correlation, and make use of it for calibrating measured signals.

9.1 Motivation

An *oscilloscope*, shown below (image credit: Tektronix), is the tool of choice for measuring electronic signals and testing equipment. You might have worked with one in courses such as EE 16AB or Physics 111A.

On the bottom right of the image, there is a row of five metallic ports. The first four are inputs for a separate measurement *channel* of the scope, which are juxtaposed on the scope's display (here, in yellow, blue, and purple).

In many cases, we measure the same signal on multiple channels (say, at different points in a circuit) and want to see what differences, if any, exist. However, this poses a challenge: the signals may be temporally offset from each other, so we need to align them before we can do any meaningful comparision. For example, this often happens because the two oscilloscope cables have different lengths—one signal has to travel through a physically longer path to the scope, and thus shows up later in time.

In this question, we'll develop an efficient algorithm to:
1) Compute the number of samples one signal lags behind another by, and
2) Align the signals by shifting by the offset.

The second item is pretty easy to accomplish—NumPy has functions for doing this—so most of our work will be in handling the first item.

9.2 Q4a: The Gaussian Pulse

The *Gaussian pulse*, a bell-curve shaped signal, is one of the most interesting and important entities in both theoretical and practical signal processing setups, so we'll be using it throughout this question as the input to our virtual oscilloscope.

One of the many things it's used for is measuring the impulse response of a CT-LTI system. Since the Dirac delta is a mathematical abstraction, and not a physically realizable signal, we can't just generate one and send it into a CT-LTI system to measure its impulse response. Instead, it's common practice to send in a very narrow, unit area Gaussian pulse (since we recover the Dirac delta in the limit of the width approaching zero), and treat the system's output as a measured impulse response. The approximation is good as long as the Gaussian used is sufficiently narrow. For example, according to reference 8 (page 26), Lawrence Livermore National Laboratory's National Ignition Facility generates impulses as Gaussian pulses with a full width at half maximum (explained below) of 88 picoseconds, which is about as close to an ideal impulse as modern hardware can generate.

9.2.1 Full Width at Half Maximum

In signal processing, we typically characterize Gaussians by their *Full Width at Half Maximum* (FWHM), shown below (image credit: Wikipedia). It is what it sounds like: the width of the pulse at half its max amplitude.

We can express the Gaussian, which is assumed to be centered at $t = 0$, in terms of its FWHM as $f(t) = 2^{-(2t\ /\ \mathrm{FWHM})^2}$. Using this formula, implement `gaussian_pulse` below according to the docstring.

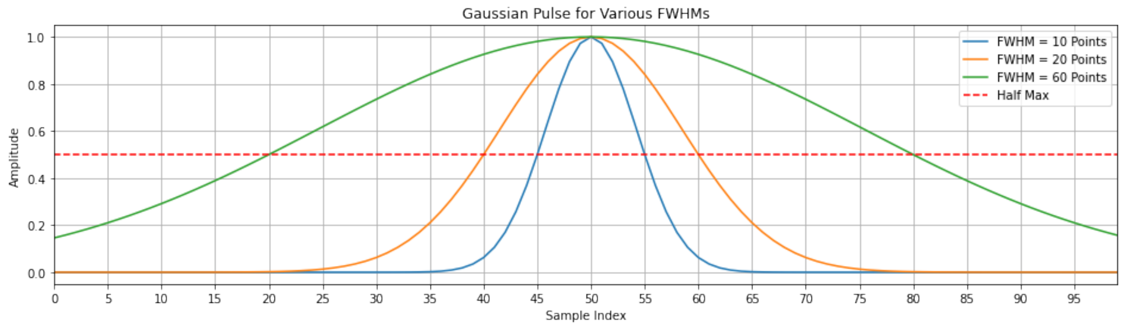
```
In [5]: def gaussian_pulse(L, fwhm):
            """
            Parameters:
            L    - The number of samples to generate the pulse over.
            fwhm - The pulse's full width at half maximum.

            Returns:
            Discrete sample of Gaussian pulse with FWHM of "fwhm"
                defined over the sample points t = -L // 2, -L // 2 + 1, ..., L // 2 - 1.
```

```
    """
    # BEGIN SOLUTION
    t = np.arange(-L // 2, L // 2)
    sig = 2 ** (-((t / (fwhm / 2)) ** 2))
    return sig
    # END SOLUTION
```

```
In [ ]: grader.check("q4a")
```

Plot the Gaussian pulses for various FWHMs:

```
In [8]: L = 100
        FWHMs = [10, 20, 60]
        plt.figure(figsize=(16, 4))
        for fwhm in FWHMs:
            plt.plot(gaussian_pulse(L, fwhm))
        plt.xlim([0, L-1])
        plt.plot(.5 * np.ones(L), 'r--')
        plt.legend(["FWHM = {0} Points".format(fwhm) for fwhm in FWHMs] + ["Half Max"])
        plt.title("Gaussian Pulse for Various FWHMs")
        plt.ylabel("Amplitude")
        plt.xlabel("Sample Index")
        plt.xticks(np.arange(0, 100, 5))
        plt.grid()
        plt.show()
```



9.3 System Model

We'll assume each oscilloscope channel measures the signal fed to it by:
1. Delaying it by some amount (different for each channel), and then 2. Adding some amount of random noise.

For simplicity, we'll consider the case of two channels, although the same principles apply if you had more channels and wanted to align all of their measurements. **Crucially, we only care about the *relative* delay between channel 1 and channel 2 because we want to align them with each other, not shift them to some absolute point in time.** This allows us to think of channel 1 as having zero delay, acting as our *reference channel*.

9.3.1 Block Diagram

Below is a delay-adder-gain block diagram of our model, where $y_1(n)$ and $y_2(n)$ are the signals measured on channels 1 and 2, respectively. In reality, the delays shown would be in continuous time, as the signals involved are only discretized once they reach the oscilloscope, but we'll think of everything as one DT system for simplicity here.

Note: - The z^{-d} delay block says that $y_2(n)$ arrives d samples after $y_1(n)$. **If $d < 0$, channel 2's signal arrives before channel 1's.** - $z_1(n), z_2(n)$ are additive noise corrupting our signals. - Each has a different strength, depending on the channel. - We'll assume *White Gaussian Noise* (WGN). We're not just doing this for academic purposes—there's ample empirical evidence that this is a good model for oscilloscope noise (see reference 9). - Channel 1 measures $y_1(n) = x(n) + z_1(n)$. - Channel 2 measures $y_2(n) = x(n - d) + z_2(n)$.

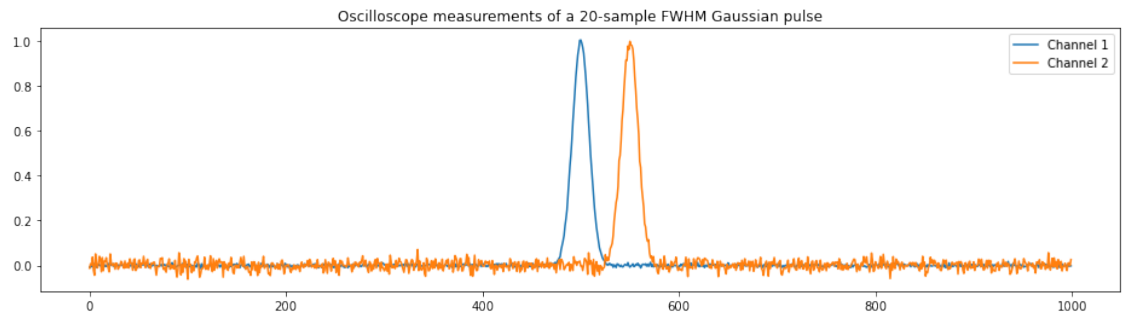
Our goal: Find d, and shift $y_2(n)$ to the left by d samples.

We've provided the class `Oscilloscope` for you below, which models this system. You are not responsible for understanding it, just run the next two cells to get a feel for what it does.

```
In [6]: class Oscilloscope:
            """
            Oscilloscope modeled as a set of channels, each of which has its own delay and noise level.
            All noise is modeled as Additive White Gaussian Noise (AWGN), and delays as shifts to the r
            """
            def __init__(self, num_channels, channel_noise_levels, channel_delays):
                """
                Parameters:
                num_channels         - The number of oscilloscope channels.
                channel_noise_levels - The standard deviation of the AWGN on each channel.
                channel_delays       - The delays, in samples, for each channel.

                A Value_Error is raised if num_channels does not match the number of elements of
                channel_noise_levels and channel_delays, or if all entries of channel_delays are
                not integers.
                """
                channel_delays = np.array(channel_delays)
                if not num_channels == len(channel_noise_levels):
                    raise ValueError("Number of channels must match number of noise levels provided.")
                if not num_channels == len(channel_delays):
                    raise ValueError("Number of channels must match number of delays provided.")
                if not np.array_equal(channel_delays, channel_delays.astype("int")):
                    raise ValueError("All delays must be integer-valued.")
                self.num_channels = num_channels
                self.channel_noise_levels = channel_noise_levels
                self.channel_delays = channel_delays

            def measure(self, signal):
                measurements = []
                for i in range(self.num_channels):
                    curr_noise = np.random.normal(scale=self.channel_noise_levels[i], size=np.shape(sig
                    curr_channel_measurement = np.roll(signal, self.channel_delays[i]) + curr_noise
                    measurements.append(curr_channel_measurement)
                return measurements
```

Let's simulate measurement of a Gaussian pulse using our Oscilloscope. We'll assume a 50-sample delay, and a modest amount of noise on each channel.

```
In [10]: # Measure a 20-sample FWHM gaussian pulse
         noise_levels = [.005, .02]
         delays = [0, 50]
         num_channels = 2
         scope = Oscilloscope(num_channels, noise_levels, delays)
         signal = gaussian_pulse(1000, 20)
         measured_signals = scope.measure(signal)

         # Display results
         plt.figure(figsize=(16, 4))
         plt.title("Oscilloscope measurements of a 20-sample FWHM Gaussian pulse")
         plt.plot(measured_signals[0], label="Channel 1")
         plt.plot(measured_signals[1], label="Channel 2")
         plt.legend()
         plt.show()
```



You should see two Gaussian pulses, each noised, with a 50-sample offset between them. This is what our raw data would look like in a real world setup!

Now, we'll align these signals and compensate for the 50-sample delay using cross-correlation.

9.4 Q4b: Cross-Correlation (Conceptual)

The *normalized* cross-correlation of two deterministic DT signals x and y, which is another DT signal R_{xy}, is defined as

$$R_{xy}(n) = \frac{\sum_{k=-\infty}^{\infty} x(k)y(k+n)}{\left(\sqrt{\sum_{k=-\infty}^{\infty} |x(k)|^2}\right)\left(\sqrt{\sum_{k=-\infty}^{\infty} |y(k)|^2}\right)}.$$

The question remains: We want to know what shift n makes $x(k)$ and $y(k+n)$ the most "aligned" in some sense, and then align them by shifting by that amount. How does cross-correlation tell us this?

9.4.1 Cosine Similarity Interpretation

Recall that in $\mathbb{R}^N$ (N-dimensional Euclidean space), if the angle between two vectors $\vec{x}, \vec{y}$ is θ, then

$$\cos(\theta) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}|| \cdot ||\vec{y}||},$$

where $||\vec{x}|| = \sqrt{\vec{x} \cdot \vec{x}}$ is the length, or *norm*, of the vector $\vec{x}$. The expression on the right hand side of the equation above is often referred to as the *cosine similarity* between $\vec{x}$ and $\vec{y}$.

The reason why we call this a "similarity" pops out when we look at what it assigns for different values of θ: - If $\theta = 0$, the vectors are parallel, and their cosine similarity is 1. - If $\theta = \pi/2$, the vectors are orthogonal, and their cosine similarity is 0. - If $\theta = \pi$, the vectors are antiparallel, and their cosine similarity is -1. - The cosine similarity is bounded between -1 and 1 for any θ due to the normalization (this can be proved via the Cauchy-Schwarz inequality). **This means the cosine similarity is maximized when the vectors are perfectly aligned!**

If we think of our duration N signals (remember, we're on a computer, so everything is finite-duration) as vectors in $\mathbb{R}^N$, then $R_{xy}(n)$ is precisely the cosine similarity between a shifted (left by n samples) version of y and an unshifted version of x.

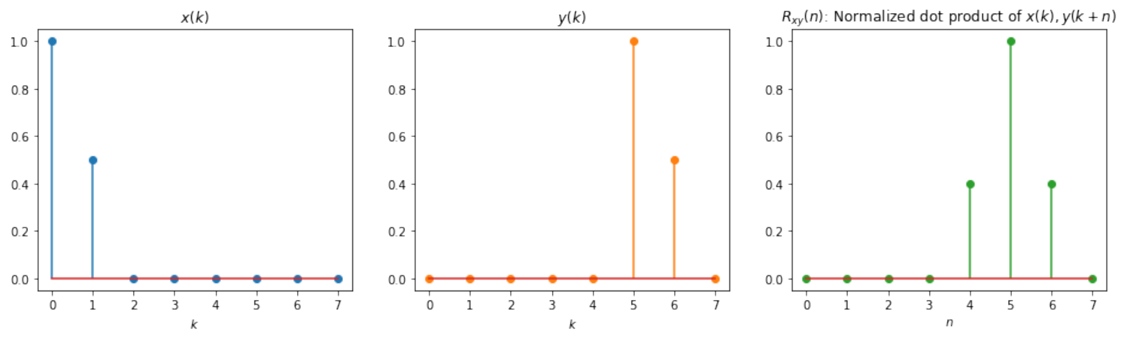
We can find the offset between the signals, then, by determining what shift produces the largest cosine similarity (i.e., makes the signals most "aligned" in Euclidean space) between them! Equivalently, we just find where $R_{xy}(n)$ is maximized, and call that the relative delay between them. That is, the number of samples *after* x that y appears.

9.4.2 An Example

To get a feel for why a bigger cross-correlation means the signals are more aligned, let's do an example. We'll manually take the dot products ourselves so it's clear which dot product corresponds to what offset.

Run the cell below and answer the questions that come after it.

```
In [11]: x = np.array([1, .5, 0, 0,  0,  0, 0, 0])
         y = np.array([0,  0,  0, 0, 0, 1, .5, 0])
         N = len(x)

         # np.roll(y, -n) shifts x to the left by n, i.e. produces y(k+n)
         R_xy = np.array([np.dot(x, np.roll(y, -n)) for n in range(N)])
         R_xy = R_xy / (np.linalg.norm(x) * np.linalg.norm(y)) # normalize

         plt.figure(figsize=(16, 4))
         plt.subplot(1, 3, 1)
         plt.stem(x)
         plt.title("$x(k)$")
         plt.xlabel("$k$")

         plt.subplot(1, 3, 2)
         plt.stem(y, linefmt="C1", markerfmt="C1o")
         plt.title("$y(k)$")
         plt.xlabel("$k$")

         plt.subplot(1, 3, 3)
         plt.stem(R_xy, linefmt="C2", markerfmt="C2o")
         plt.title("$R_{xy}(n)$: Normalized dot product of $x(k), y(k+n)$")
         plt.xlabel("$n$")
         plt.show()
```



For all questions, only consider the eight shift values of $n$ that are shown in the cross-correlation output, 0 through 7.

Q: For what shift values n do $x(k)$ and $y(k+n)$ *not* overlap? What is $R_{xy}(n)$ there? Considered as vectors in $\mathbb{R}^8$, what is the angle θ between them for those shift values?

A: (TODO)

A: (TODO) They do not overlap for $n = 0, 1, 2, 3, 7$. Accordingly, the cross-correlation is zero there. Using the cosine similarity interpretation from before, when considered as vectors in 8-dimensional Euclidean space, they are are orthogonal, with $\theta = \pi/2$.

Q: For what shift value n_* are $x(k)$ and $y(k+n_*)$ identical? What is $R_{xy}(n_*)$? Considered as vectors in $\mathbb{R}^8$, what is the angle θ between them for this shift value?

A: (TODO)

A: (TODO) It is clear from the plots that $x(k) = y(k+5)$, i.e. that x can be obtained by shifting y left by five samples, so $n_* = 5$. We see that $R_{xy}(5) = 1$ as expected. For this shift, the signals are identical, so when considered as vectors in Euclidean space, they are parallel, with $\theta = 0$.

9.5 Q4c: Cross-Correlation (Implementation)

Now that we've seen how cross-correlation can solve our signal alignment problem, we need an efficient way to implement it.

9.5.1 Filtering Interpretation

The numerator in our expression for cross-correlation is the main troublemaker, since the denominator is just a product of two norms which we know how to calculate with `np.linalg.norm`. However, the numerator looks suspiciously similar to a convolution. By making the change of variables $m = -k$, so that $k = -m$, we see that

$$R_{xy}(n) = \frac{\sum_{k=-\infty}^{\infty} x(k)y(k+n)}{||x|| \cdot ||y||} = \frac{\sum_{m=-\infty}^{\infty} x(-m)y(n-m)}{||x|| \cdot ||y||},$$

and it's clear that R_{xy} is the convolution of y with a time-reversed version of x, all divided by the product of their norms. We've rewritten the denominator in terms of the norms of x and y here for brevity.

9.5.2 Implementation Details

As we saw above, cross-correlation reduces to a convolution. What a relief! Convolution can be implemented efficiently by taking the product of the FFTs (convolution in time is multiplication in frequency). We just need to figure out how to handle the time-reversal.

Time-Reversal Since we're working with finite-duration signals here, we can't just use `x[::-1]` and declare victory. If x holds values corresponding to $n = 0, ..., N-1$, then reversing the array `x` in-place puts the value for $n = N-1$ at the place for $n = 0$, not $n = -N+1$. We could solve this issue by carefully doing some zero-padding, but this is a bit annoying. This is where the Fourier Transform properties come to the rescue.

If $x(n)$ and $X(\omega)$ form a time-frequency pair (i.e., $X(\omega)$ is the DTFT of $x(n)$), then $x(-n)$ and $X(-\omega)$ also form a time-frequency pair. This doesn't appear to help much, though, as there's still a reversal involved.

However, we know that when x is real-valued (as is the case here), its spectrum is *conjugate-symmetric*: $X(-\omega) = X^*(\omega)$. So, instead of dealing with any reversal going on in either domain, we can just conjugate the spectrum! **This is much simpler and more efficient, as complex conjugation is an element-wise operation.** Contrast this with array reversal, which typically requires you to swap a bunch of data (or pointers).

Zero-Padding We know that the DT convolution (or cross-correlation) of a duration M signal and duration N signal produces a duration $M+N-1$ signal. In the most general case, all $M+N-1$ points are nonzero and must be stored in our output. Since we're implementing a modified form of convolution here, we need to account for this, and make sure our output is at least of size $M+N-1$. Essentially, we're doing a "full" mode cross-correlation.

For added efficiency, we'll just use the next power of two after or including $M+N-1$. This won't affect our results, since the zeros contribute nothing to the dot products.

9.5.3 Your Job

Implement the function `xcorr` below which returns the **normalized** cross-correlation of two signals x and y through the algorithm we've developed in this question: 1. Zero pad x and y to `next_power_of_2(len(x) + len(y) - 1)`, obtaining $x_{\text{pad}}, y_{\text{pad}}$. 2. Multiply the FFT of y_{pad} with the complex conjugate of the FFT of x_{pad}. 3. Take the inverse FFT of the product computed in step 2. 4. Divide by $||x|| \cdot ||y||$. 5. *Center* the cross-correlation using np.fft.fftshift. This places the dot product corresponding to a delay of zero in the middle of the output. 6. Return the real part. This eliminates the small, but fake, imaginary part caused by numerical imprecision. The cross-correlation of two real signals is always real-valued.

You are welcome to use np.fft.fft, np.fft.ifft, np.linalg.norm, np.concatenate, np.zeros, np.conj, and the provided `next_power_of_2` function in your implementation. You may **NOT** use `np.convolve`, `np.correlate`, or any other library functions that directly compute convolution or cross-correlation.

```
In [7]: def next_power_of_2(n):
            """Returns the next power of 2 after and including n."""
            return 2 ** int(np.ceil(np.log2(n)))

        def xcorr(x, y):
            """
            Returns the cross-correlation of x and y, padded to the next power of 2 after len(x) + len(
            Assumes that x, y are both real signals.
            """
            # BEGIN SOLUTION
            L = next_power_of_2(len(x) + len(y) - 1)
            x = np.concatenate((x, np.zeros(L - len(x))))
            y = np.concatenate((y, np.zeros(L - len(y))))
            R_xy = np.fft.ifft(np.fft.fft(y) * np.conj(np.fft.fft(x))) / (np.linalg.norm(x) * np.linalg
            return np.real(np.fft.fftshift(R_xy))
            # END SOLUTION
```

Run the cell below to test your `xcorr` function. These tests aren't comprehensive, but should be good enough for our purposes. This will be used for grading purposes, so make sure you're passing all tests before moving on.

```
In [ ]: grader.check("q4c")
```

9.6 Q4d: Optimal Offset

Implement `opt_offset`, which takes in two signals, x and y, and returns the "optimal offset" between them, defined as the index (relative to zero) at which their cross-correlation is maximized. This corresponds to the number of samples that x is *ahead* (i.e., to the left of) y by.

Hint: Our cross-correlation is zero-centered. To find the actual delay, we need to extract the index at which the cross-correlation is maximized, and subtract the index corresponding to a zero delay (i.e., the center) from it.

```
In [8]: def opt_offset(x, y):
            """Returns the most likely number of samples that x lags behind y via cross-correlation."""
            # BEGIN SOLUTION
            R_xy = xcorr(x, y)
            zero_delay_index = len(R_xy) // 2
            opt_delay_index = np.argmax(R_xy)
            return opt_delay_index - zero_delay_index
            # END SOLUTION
```

Here a few sanity checks on `opt_offset` you can run.

```
In [ ]: grader.check("q4d")
```

9.7 Q4e: Realignment

To complete the lab, and provide us with a nice abstraction to solve our original problem, implement `align`. This function takes in two signals, finds the optimal offset between them, and shifts the first to the right by that amount so they're aligned.

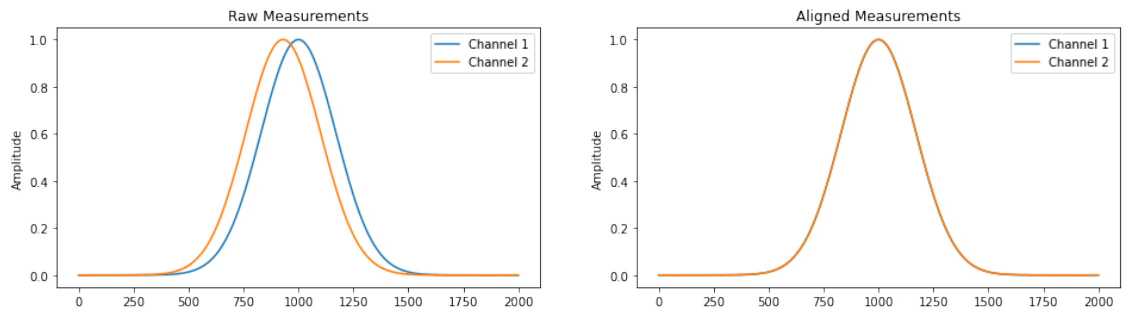
This should be very simple (two lines of code, at most) once you have `opt_offset` working.

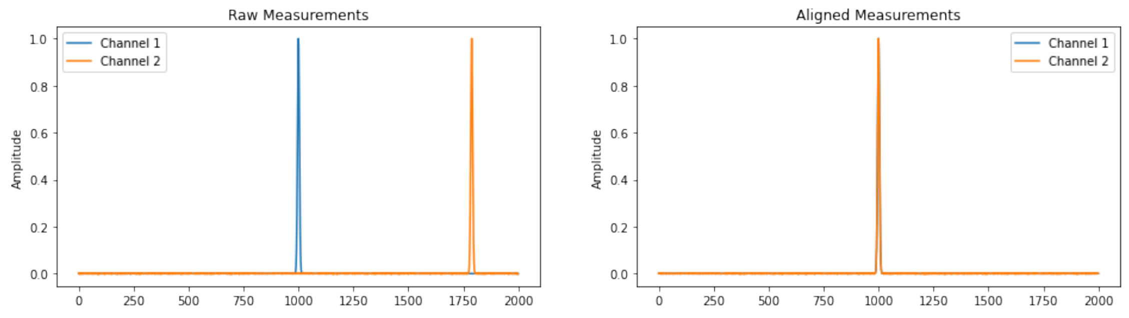
```
In [9]: def align(x, x_ref):
            """Assuming x is a delayed copy of x_ref, finds the timing offset and shifts x to be aligne
            # BEGIN SOLUTION
            lag = opt_offset(x, x_ref)
            return np.roll(x, lag)
            # END SOLUTION
```

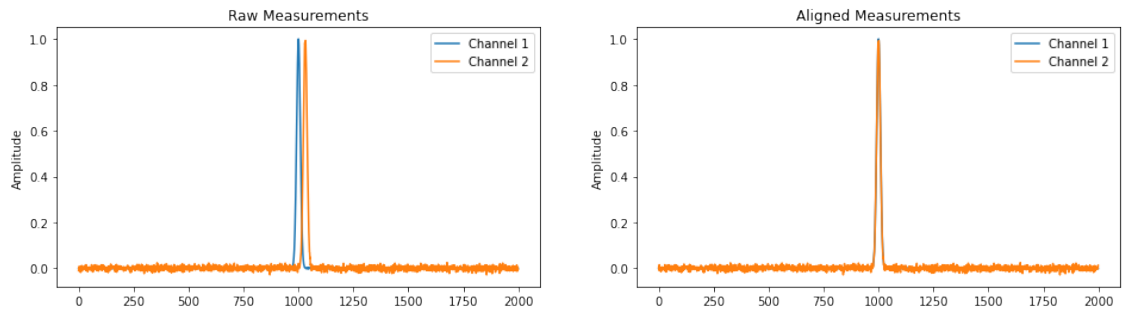
And now, the moment we've been building to: let's align some oscilloscope measurements! Run the cell below.

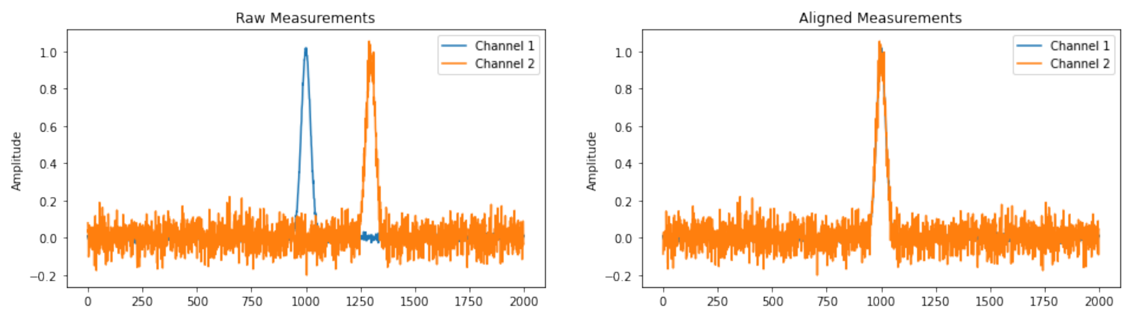
```
In [15]: # Mock setup to test out the calibration
         num_channels = 2
         record_length = 2000
         fwhms = [400, 10, 20, 50, 100]
         ch1_noise_levels = [0,     .0001, .001, .01, .1]
         ch2_noise_levels = [.0003, .0009, .008, .06, .23]
         ch1_delays = [0, 0, 0,  0, 0]
         ch2_delays = [-70, 788, 31, 297, 563]

         for i in range(len(fwhms)):
             # Construct oscilloscope
             curr_noise_levels = [ch1_noise_levels[i], ch2_noise_levels[i]]
             curr_delays = [ch1_delays[i], ch2_delays[i]]
             scope = Oscilloscope(num_channels, curr_noise_levels, curr_delays)

             # Measure
             curr_signal = gaussian_pulse(record_length, fwhms[i])
             measured_signals = scope.measure(curr_signal)
             ch1_meas = measured_signals[0]
             ch2_meas = measured_signals[1]

             # Align and display results
             ch2_corrected = align(ch2_meas, ch1_meas)
             plt.figure(figsize=(16, 4))
             plt.subplot(1, 2, 1)
             plt.title("Raw Measurements")
             plt.plot(ch1_meas, label="Channel 1")
             plt.plot(ch2_meas, label="Channel 2")
             plt.ylabel("Amplitude")
             plt.legend()
```

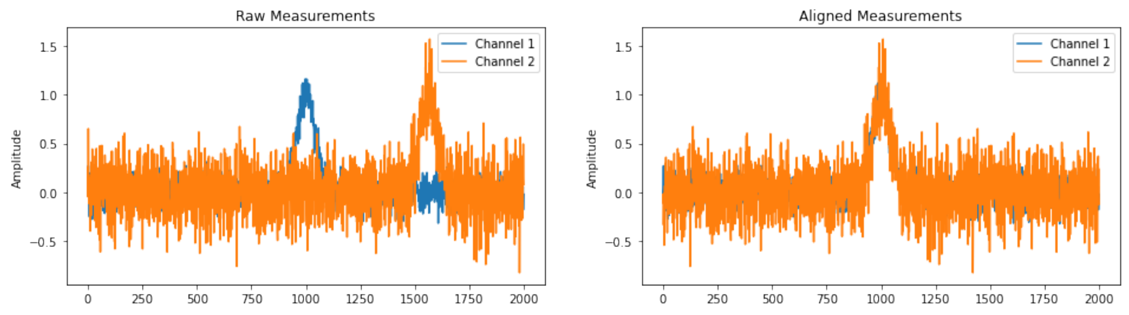
```
    plt.subplot(1, 2, 2)
    plt.title("Aligned Measurements")
    plt.plot(ch1_meas, label="Channel 1")
    plt.plot(ch2_corrected, label="Channel 2")
    plt.ylabel("Amplitude")
    plt.legend()
plt.show()
```











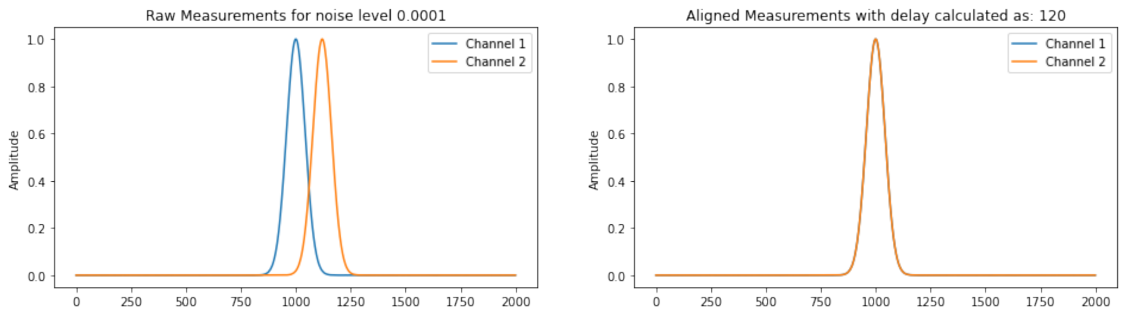
You should see two columns and five rows. The first column contains the "raw", unaligned measurements, and the second is the result of calling your `align` function to compensate for the delays. If everything is correct, you should see nicely aligned signals in the second column.

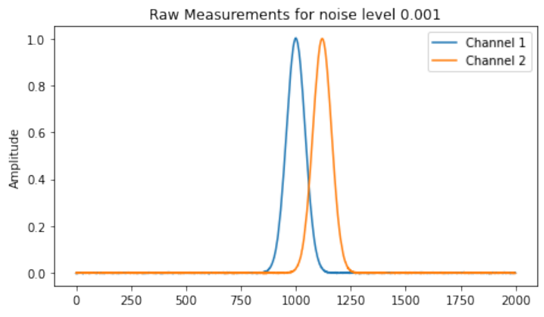
Take a look at how well your alignment works, even on the bottom row, where the noise is incredibly strong. Cross-correlation's offset detection capability in the presence of additive random noise is remarkably robust, which is why it's the standard solution used in practice for this problem.

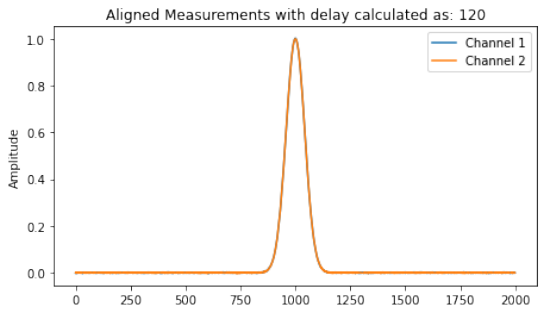
9.8 (Optional) Q4f: Noise Levels

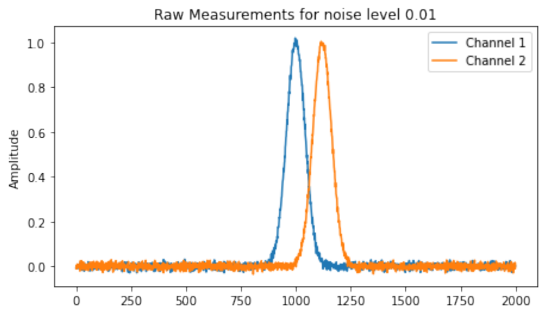
See how high you can crank up the noise levels before our realignment technique fails for this specific setup of the experiment with an actual delay of 120 samples. Note that the noise is random, so try re-running the cell a few times to get a grasp on which noise levels consistently fail or succeed.

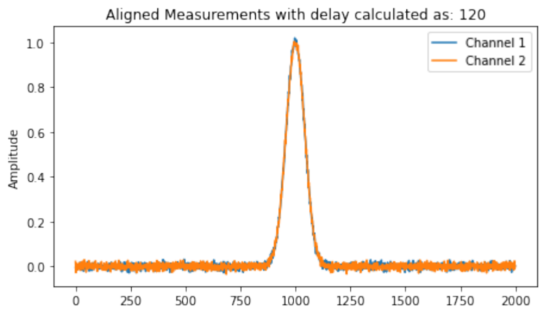
```
In [10]: # Mock setup to test out the calibration
         num_channels = 2
         record_length = 2000
         fwhm = 100
         noise_levels = [.0001, .001, .01, .1, 1]
         delay = 120

         for i in range(len(noise_levels)):
             # Construct oscilloscope
             curr_noise_levels = [noise_levels[i], noise_levels[i]]
             delays = [0, 120]
             scope = Oscilloscope(num_channels, curr_noise_levels, delays)

             # Measure
             curr_signal = gaussian_pulse(record_length, fwhm)
             measured_signals = scope.measure(curr_signal)
             ch1_meas = measured_signals[0]
             ch2_meas = measured_signals[1]

             # Align and display results
             ch2_corrected = align(ch2_meas, ch1_meas)
             empirical_offset = opt_offset(ch1_meas, ch2_meas)
             plt.figure(figsize=(16, 4))
             plt.subplot(1, 2, 1)
             plt.title("Raw Measurements for noise level " + str(noise_levels[i]))
             plt.plot(ch1_meas, label="Channel 1")
             plt.plot(ch2_meas, label="Channel 2")
             plt.ylabel("Amplitude")
             plt.legend()


             plt.subplot(1, 2, 2)
             plt.title("Aligned Measurements with delay calculated as: " + str(empirical_offset))
             plt.plot(ch1_meas, label="Channel 1")
             plt.plot(ch2_corrected, label="Channel 2")
             plt.ylabel("Amplitude")
             plt.legend()
         plt.show()
```

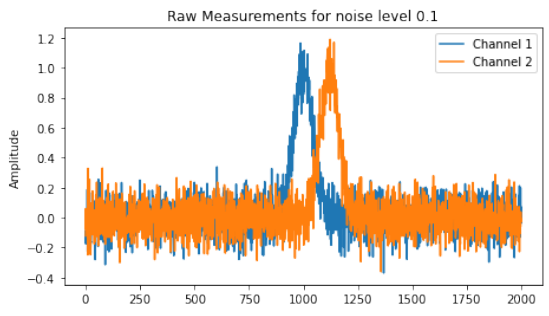


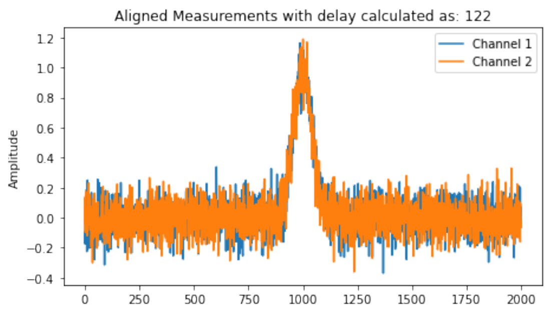

Raw Measurements for noise level 0.001
Amplitude
Channel 1
Channel 2

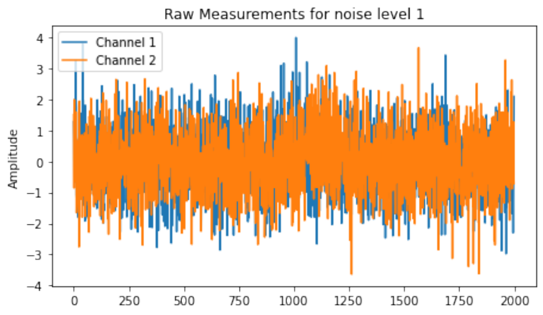

Aligned Measurements with delay calculated as: 120
Amplitude
Channel 1
Channel 2

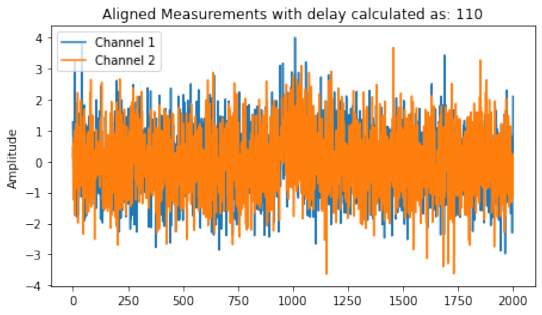

Raw Measurements for noise level 0.01
Amplitude
Channel 1
Channel 2


Aligned Measurements with delay calculated as: 120
Amplitude
Channel 1
Channel 2


Raw Measurements for noise level 0.1
Amplitude
Channel 1
Channel 2


Aligned Measurements with delay calculated as: 122
Amplitude
Channel 1
Channel 2





Q: For what threshold of noise_level does the system consistently incorrectly calculate the delay?

A: (TODO)

A: Starting at noise level 0.1, the alignment is consistently incorrect.

9.9 (Optional) Q4g: Monte Carlo Simulation

Since the noise is random, you can even re-run the experiment multiple times and empirically find the probability of the procedure failing as a function of the white Gaussian noise's standard deviation. Complete the function below to implement a Monte Carlo Simulation to empirically estimate the probability of failure for a given noise level.

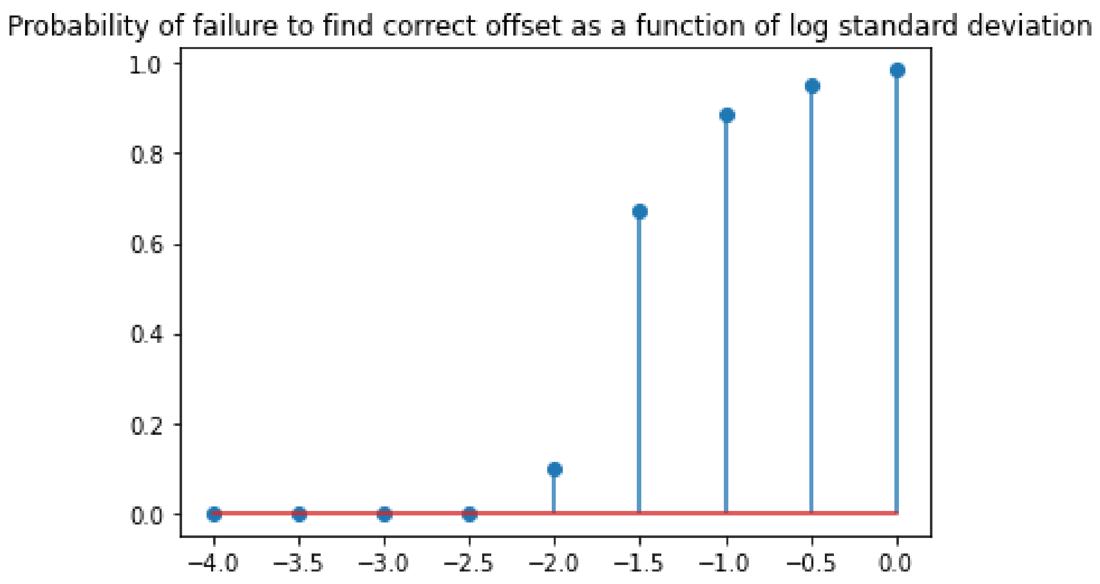
A Monte Carlo simulation for failure rate can be implemented as simulating the experiment a large number of times, counting how many times the experiment failed, and returning the proportion of failed experiments to total number of experiments (i.e. the empirical failure rate).

```
In [11]: def simulation(n_trials, num_channels, record_length, fwhm, noise_level, delay):
             """
             Monte Carlo simulation to estimate probability of cross-correlation based alignment to cal

             Parameters:
             n_trials
             num_channels
             record_length
             fwhm
             noise_level - standard deviation of Gaussian noise
             delay - the true delay of system

             Returns:
             (float) - estimated probability of failure
```

```
            """
            n_failures = 0
            noise_levels = [noise_level, noise_level]
            delays = [0, delay]
            scope = Oscilloscope(num_channels, noise_levels, delays)
            curr_signal = gaussian_pulse(record_length, fwhm)
            for _ in range(n_trials):

                # BEGIN SOLUTION
                measured_signals = scope.measure(curr_signal)
                ch1_meas = measured_signals[0]
                ch2_meas = measured_signals[1]
                empirical_offset = opt_offset(ch1_meas, ch2_meas)
                n_failures += (empirical_offset != delay)
                # END SOLUTION

            p_hat = n_failures/n_trials
            return p_hat
```

Run the cell below to plot the empirical failure probability corresponding to noise levels.

```
In [12]: n_trials = 1000
         num_channels = 2
         record_length = 2000
         fwhm = 100
         logs = np.arange(-4, 0.01, 0.5)
         noise_levels = [10**i for i in logs]
         prob_failures = [None for i in logs]
         delay = 120
         for i in range(len(noise_levels)):
             prob_failures[i] = simulation(n_trials, num_channels, record_length, fwhm, noise_levels[i]
         plt.figure()
         plt.title("Probability of failure to find correct offset as a function of log standard deviati
         plt.stem(logs, prob_failures)
         plt.show()
```



10 References

[1] *Cooley-Tukey FFT algorithm (Wikipedia).* Link
[2] *EE 123 - Spring 2018 - Lecture 3C, Fast Convolutions and the FFT.* Link
[3] *EE 123 - Spring 2018 - Lecture 4A, The FFT.* Link
[3] *Full width at half maximum (Wikipedia).* Link
[4] *Matched filter (Wikipedia).* Link
[5] *An Industrial Strength Audio Search Algorithm.* Link
[6] *EE 120 - Fall 2019 - Lecture 10 Notes.* Link
[7] *EE 123 - Spring 2016 - Lab 3, Time Frequency Part I.* Link
[8] *NIF Laser Capabilities.* Link
[9] *Evaluating Oscilloscope Vertical Noise Characteristics.* Link

Additionally, these materials, while not used in developing this lab, may be of interest: - *Explained: The Discrete Fourier Transform.* Link - *Rensselaer Polytechnic Institute. ECSE-4530, Digital Signal Processing - Lecture 11: Radix 2 Fast Fourier Transforms.* Link

10.1 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

Please upload the zip file produced by the result of this command to Gradescope.

```
In [ ]: grader.export(force_save=True, run_tests=True)
```