PHYS134
Homework/Project 7
2021-04-12

Sodium Doublet Theory In this question, you'll derive the interference pattern as a function of displacement d of a Michelson Interferometer for the Sodium Doublet, two spectral lines roughly located at approximately $\lambda_1 = 589.0\,\text{nm}$ and $\lambda_1 = 589.6\,\text{nm}$. Specifically, you'll derive the distance between maxima (or minima) of the fringe visibility.

1. As in class, assume simple plane-waves and "unfold" the interferometer. Assume what's seen at the detector (say at the origin) is one copy of the incoming time-dependent wave plus another copy shifted in time by τ. Draw a picture of the interferometer with a source and detector along with its unfolded version. Explain why $\tau = 2d/c$, where d is the amount that a mirror has moved from equal-path-length.

2. Calculate the intensity at the detector as a function of τ assuming that the source consists of two angular frequencies ω_1 and ω_2, of equal amplitude. Don't bother with overall constants—only the *form* of the intensity as a function of τ matters. You can do this in one of two ways:

   (a) Option 1: Real cosines [not recommended]. The Intensity is the time average of the electric field squared. This is not recommended because there are a lot of trig identities involved, and averaging over time requires more care.

   (b) Option 2: Complex exponentials [recommended]. The Intensity is the time average of the electric field *magnitude* squared (complex field times its complex conjugate). Here, once you multiply out all of the terms, it's obvious which ones average way over any reasonable time—anything of the form $e^{i\omega t}$, where ω is anywhere near optical frequencies. Only terms involving τ remain. Remember, τ is something you control by setting the knob to a particular d.

3. Write your answer in terms of ω, the average of ω_1 and ω_2 along with their difference $\Delta\omega = \omega_1 - \omega_2$.

4. Plot this for the case where $\omega \approx 20\Delta\omega$, which is not nearly as big a factor as in the sodium doublet, but produces a reasonable plot. I'm not interested in the axes or particular values—just the general picture.

5. Your plot should show "fast fringes" modulated by a slower envelope. The "fast fringes" go to from maxima to maxima as the knob is turned through roughly one average wavelength. The envelope goes from "no fringes" to "bright fringes" and back to "no fringes" on much longer distance scales. Find the amount that d must change to go from "no fringes" back to "no fringes".

■

Sodium Doublet Analysis Watch the video of the experiment (might not be posted yet) Lab6FourierTransformSpectrometer.mp4

1. Analyze the data in `lab6_fts_sodium_doublet.csv` on Sakai. Remember that the "zero" reading on the micrometer has nothing to do with zero path length—that occurs somewhere in the middle of the micrometer's range so we can go in both directions. Also, the 1st, 2nd, 3nd, etc. minima don't really mean much by themselves, but provide an x-axis for a linear fit.

2. Be sure to show the three usual plots (data+fit, residuals, and normalized residuals). Be sure to calculate reduced χ^2 and PTE. Be sure the fit parameters and these values are clearly printed.

3. Assume that you know that $\lambda_{avg} = 589.3$ from some other means, like a diffraction grating that isn't sensitive enough to resolve the two spectral peaks on their own. Translate the fit parameters (or maybe only one fit parameter) into a $\Delta\lambda$ by proper propagation of error.

■

Audio as Arrays in Python In this problem and the next, you'll get a taste for Fourier Transforms, time and frequency plots, and dealing with sampled data. You'll do this in the context of audio, where we have direct access to the amplitude as a function of time (unlike light, where we only can measure intensity). Many of you also have a feel for what different frequencies sound like.

1. You might want to start your notebook with `%pylab notebook` this time, so you can zoom into long audio plots. Unfortunately, you'll need to explicitly start a new figure by typing `figure()` before each plot. I like to make the figure size as wide as possible without creating a scroll bar, so I do `figure(figsize=(9.5,5))` before each plot.

2. To embed and play audio in a notebook, you'll need to import a library. Here's code to import the library, create a tone, and play it.

   ```
   import IPython.display as ipd

   sample_rate = 32000 # sample rate in samples per second (Hz)
   T = 2.0    # total time in seconds
   t = linspace(0, T, int(T*sample_rate), endpoint=False) # time variable
   x = sin(2*pi*440*t)                # pure sine wave at 440 Hz

   ipd.Audio(x, rate=sample_rate) # play the array; normalize overall amplitude
   ```

3. Add some white Gaussian noise to the tone and play it. You can call `randn(len(t))` to generate the right number of random Gaussian samples. You'll want to scale down the amplitude of the noise.

4. Plot both the tone and the tone+noise as a function of time. You'll want to use `xlim()` to zoom in enough to see the sine wave.

5. Find or record a few-seconds-long sound as a `.wav` file, or suffer through my recording in `okso.wav` on Sakai. A `.wav` contains uncompressed audio samples plus some meta information like the sample rate. When scipy reads the wave file, it returns the sample rate and the data. If your file is in stereo, you may need to select one of the two channels. My example is not in stereo. Code to do this is:

   ```
   import scipy.io.wavfile
   (sample_rate, x) = scipy.io.wavfile.read('okso.wav')
   ipd.Audio(x, rate=sample_rate)
   ```

6. Plot your file for the entire few-second time range and then zoom in. I'll help you construct an array of sample times for the x axis of the plot:

   ```
   t = arange(len(x)) / sample_rate  # make a sample-time array
   ```

7. Plot the power spectral density by calling `psd()`. You'll need to pass in the sample rate so it knows how to scale the frequency x-axis. The y-axis is in decibels, where each 10dB is a factor of 10 in *power*. Note the *maximum* frequency, which is half of the sample rate. This maximum frequency would correspond to a wave whose samples go "up,down,up,down,..." (too high-pitched for old people like me to hear and maybe too high for your speakers to produce).

   ```
   psd(x, Fs=sample_rate);
   ```

■

Audio Fourier Transforms in Python (continued from previous problem)

1. Now for the real Fourier Transforms! Specifically the Fast Fourier Transform, which is an efficient, recursive method of taking the Fourier Transform of sampled data. The Fourier Transform can be inverted, so if there are 10 000 samples in time to start with, the FFT will compute 10 000 samples in frequency.

   ```
   xFFT = fft.fft(x)
   ```

2. Examine the array that's returned. It's full of complex numbers. These are the complex coefficients of $e^{2\pi i f t}$ for each finely-spaced frequency f, which can be both positive and negative.

3. To get the frequencies corresponding to each frequency sample, use the function `ft.fftfreq()`, which needs to know the number of samples and the sample spacing:

   ```
   freq = fft.fftfreq(len(xFFT), 1.0/sample_rate)
   ```

4. Check that `max(freq)` is close to half the sample rate—half the sample rate is the maximum frequency that can be sampled (remember "up,down,up,down,...").

5. Check that the first non-zero frequency `freq[1]` is very close to the inverse of the total time of the waveform, `1/max(t)`. The resolution (the spacing) of the frequencies is set not by the sample rate, but by the total time. This lowest non-zero frequency corresponds to a complex exponential ($\cos + i\sin$) that oscillates exactly one full cycle during the total time. The first Fourier coefficient, `xFFT[1]`, corresponds to the (complex) amplitude of that oscillation. The second Fourier coefficient corresponds to the amplitude of the oscillation with two periods, and so on.

6. The frequencies are stored in a particular way, with all of the positive frequencies first, then all of the negative frequencies. Look at this by plotting `freq` as a function of its index, just by `plot(freq)`.

7. Plot the magnitude of `xFFT` with respect to its index with `plot(abs(xFFT))`. It looks like the spectrum drops off and then picks up again at the end, but this is an artifact of how positive and negative frequencies are stored.

8. There are two ways to address this, and we'll do both. First plot the magnitude of the Fourier coefficients versus `freq`. Explore by zooming around. The plot should be symmetric around zero. This is because we started with a real signal, meaning that each negative-frequency coefficient is the complex conjugate of its positive counterpart. This way, when turned back into a function of time, they combine into real sines and cosines:

   $$(a+ib)e^{+2\pi i f t} + (a-ib)e^{-2\pi i f t} = 2a\cos(2\pi f t) - 2b\sin(2\pi f t)$$

9. The other way to address this is to re-arrange the array itself so that the zero frequency isn't in the 0 index, but is in the center of the array. There is a convenient function for this called `fft.fftshift`, which can be used to re-arrange both the `xFFT` Fourier coefficients and the `freq` frequencies. Plot the shifted frequencies vs index. Plot the magnitude of the shifted Fourier coefficients versus index. We'll drop this method now, but when we get to 2D arrays, it's nice to show the 0 frequency at the center of the image rather than the corners.

■

Audio Filtering in Python (continued from previous problem)

1. As an example of what you can do, we'll band-pass filter the wave file. Create a filter profile that's a Gaussian centered around some center frequency with some width. We use `abs(freq)`, as is done below, to filter both positive and negative frequencies symmetrically. Feel free to play with the parameters or try other filter functions.

```
center = 500 # Hz
sigma  = 200 # Hz
filt = exp(-(abs(freq)-center)**2/sigma**2)  # Gaussian shape
```

2. After you create the filter profile, plot it vs `freq`.

3. Now we'll do the filtering by multiplying our Fourier coefficients by the filter profile. This effectively zeros out frequencies far away from the chosen center (`center`) frequency.

```
xFFT_filt = xFFT * filt  # apply the filter in the frequency domain
```

4. Plot this vs `freq` and zoom around.

5. Then we'll inverse Fourier transform to get back a filtered version of the signal "in the time domain" (as a function of time).

```
x_filt_complex = fft.ifft(xFFT_filt)  # inverse Fourier transform!
```

6. Unfortunately, due to rounding errors, the inverse-transformed signal has very small imaginary components. We'll throw those rounding errors away and just keep the real part. You should peek at each of these arrays as you make them.

```
x_filt = x_filt_complex.real  # throw out the tiny imaginary components
```

7. You should be able to play the filtered audio, plot it, and plot its power spectral density.

8. This has all been a warm-up exercise for diffraction, where we'll do 2D FFTs on 2D arrays. We'll have to keeping track of the spatial frequencies (the $\vec{k}$ vectors, each component of which can be positive or negative). Managing the layout of the arrays requires similar bookkeeping.

■