CS 3430: S22: Scientific Computing
Assignment 08
Fourier Coefficients, Function Approximations, and Music

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 19, 2022

Learning Objectives

1. Fourier Coefficients
2. Approximation of Functions with Fourier Series
3. Audio Processing

Introduction

In this assignment, we'll approximate functions with Fourier series whose coefficients will be computed with integral approximation and process a few music files.

You'll save your solutions in `cs3430_s22_hw08.py` and `cs3430_s22_hw08_uts.py`. All unit tests for this assignment will involve plots. We'll re-use Romberg integration from Assignment 07 for definite integral approximation.

Problem 0: (0 points)

Review the slides for Lectures 15 and 16 in Canvas and become comfortable with periodic functions, harmonics, trigonometric polynomials, Fourier series, and Fourier coefficients. You may also review Lecture 14 for Romberg Integration.

Problem 1: (1 point)

Recall from Lecture 16 that a Fourier series corresponding to some function $f(x)$ is defined as

$$f(x) \approx \frac{a_0}{2} + \sum_{k=1}^{\infty}(a_k\texttt{cos}(kx) + b_k\texttt{sin}(kx)).$$

Since we cannot deal with ∞ to solve practical scientific computing problems, we have to approximate $f(x)$ with partial sums of the Fourier series. The n-th partial sum of the Fourier series above is defined as

$$s_n(x) = \frac{a_0}{2} + \sum_{k=1}^{n}(a_k\texttt{cos}(kx) + b_k\texttt{sin}(kx)),$$

where n is a positive integer. For example, the 4-th partial sum is

$$s_4(x) = \frac{a_0}{2} + \sum_{k=1}^{4}(a_k\texttt{cos}(kx) + b_k\texttt{sin}(kx)) =$$

$$\frac{a_0}{2} + (a_1\texttt{cos}(1x) + b_1\texttt{sin}(1x)) + (a_2\texttt{cos}(2x) + b_2\texttt{sin}(2x)) +$$
$$(a_3\texttt{cos}(3x) + b_3\texttt{sin}(3x)) + (a_4\texttt{cos}(4x) + b_4\texttt{sin}(4x)).$$

Recall from Lecture 16 (See slides 15–18) that we can compute the Fourier coefficients a_n, b_n with the following formulas.

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\texttt{cos}(nx)dx, n = 0, 1, 2, ...,$$

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\texttt{sin}(nx)dx, n = 1, 2, ...$$

Equivalently, some texts use the k subscripts a_k, b_k.

$$a_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\texttt{cos}(kx)dx, k = 0, 1, 2, ...,$$

$$b_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\texttt{sin}(kx)dx, k = 1, 2, ...$$

The integrals in the above formulas can be approximated with Rombergs (or some other integral approximation methods such as Riemann sums). But since we've implemented Rombergs in Assignment 07, we might as well re-use them in this assignment.

Given a function $f(x)$, we can compute the first $n + 1$ a_k coefficients and the first n b_k coefficients and approximate $f(x)$ at a given value of x with the n-th partial sum s_n. For example, if we've computed a_0, a_1, a_2, a_3 and b_1, b_2, b_3, we can approximate $f(x)$ with $s_3(x)$ as follows.

$$f(x) \approx s_3(x) = \frac{a_0}{2} + (a_1\texttt{cos}(1x) + b_1\texttt{sin}(1x)) +$$

$$(a_2\texttt{cos}(2x) + b_2\texttt{sin}(2x)) + (a_3\texttt{cos}(3x) + b_3\texttt{sin}(3x)).$$

Implement the function `nth_partial_sum_of_fourier_series(x, acoeffs, bcoeffs)` that takes a value x, an array of $n + 1$ a_k coefficients and an array of n b_k coefficients and returns the value of $s_n(x)$.

Problem 2 (2 points)

Given a function $f(x)$, there are 3 main steps in approximating this function with Fourier series: 1) plot the function (if possible/necessary); 2) compute and plot the several partial sums for different numbers of coefficients on a given interval; and 3) plot the true error between the values of the function and the partial sum approximations on a given interval.

Let's do these steps with $f(x) = x^2$ on the interval $[-\pi, \pi]$. The method `test_plot_fun_00()` in `cs3430_s21_hw08_uts.py` plots this function and produces the plot in Figure 1.

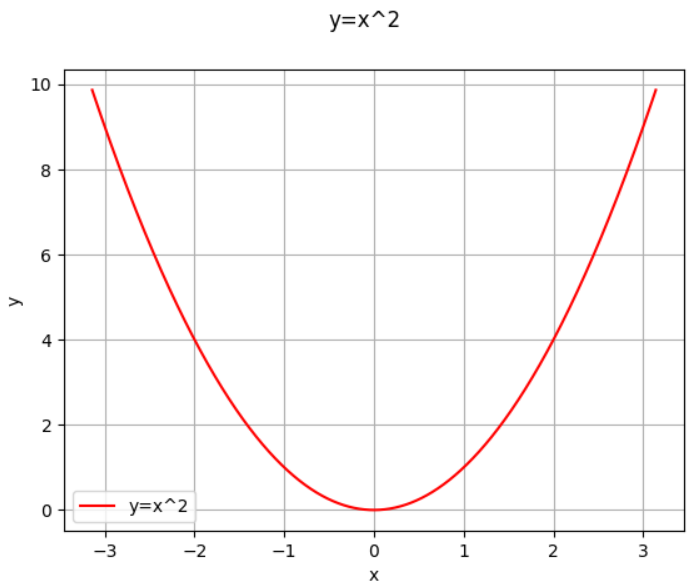

Figure 1: Plot of $y = x^2$ on $[-\pi, \pi]$.

The next task is to approximate the function with partial sums s_n for different values of n (i.e., with different numbers of coefficients). I've implemented `test_nth_partial_sum_fun_00()` in `cs3430_s21_hw08_uts.py` that you can use to plot the curves of $f(x) = x^2$ and $s_n(x)$ for different values of n. My code uses Romberg integration to approximate the definite integrals on $[-\pi, \pi]$ with $R(j, l)$ (i.e., the (j, l) value in the Romberg lattice). I choose the top of the lattice by computing `rmb.rjl(f, a, b, rn, rn)`, where `rmb.rjl()` is the static method from `rmb.py` we implemented in Assignment 07. You can play with different Rombergs to see their impact on true error.

Let's approximate $f(x)$ on $[-\pi, \pi]$ with $s_3(x)$ and plot both curves. Figure 2 shows the two plots. Not bad for just three Fourier coefficients but let's see if we can do better with $s_{10}(x)$. Figure 3 gives the plots of $f(x)$ and $s_{10}(x)$. This is much better, except at the ends, which is frequently the case anyway. What happens if we approximate with 200 coefficients? Figure 4 gives the answer. This approximation is sufficiently close, because the two curves coincide (or "superimpose") and visually blend into one curve.

Let's plot the true error between the values of $f(x)$ and $s_n(x)$ for different values of n. You can do it with `test_plot_error_fun_00()` in `cs3430_s22_hw08_uts.py`. Figures 5, 6, and 7 plot true error between $y = x^2$ and $s_3(x)$, $s_{10}(x)$, and $s_{200}(x)$, respectively. For s_3, true error ranges from ≈ 1.6 to ≈ -0.6; for $s_{10}(x)$, the error range becomes narrower: $[\approx 0.5, \approx -0.2]$; for $s_{200}(x)$, the range narrows down to $[\approx 0.02, \approx -0.006]$.

Do the same Fourier series approximation analysis in `cs3430_s22_hw08_uts.py` for the following 10 functions on $[-\pi, \pi]$. You can skip step 1 (the plotting step) if you want. I've done the first five functions (1 – 5) for you and wrote `test_plot_error_fun_01()`, `test_plot_error_fun_02()`, and `test_plot_error_fun_03()`. You can follow these examples to do functions 6 – 10.

1. $f_1(x) = x^2 + 3$;
2. $f_2(x) = 4x^2 - 5$;
3. $f_3(x) = x^4$;
4. $f_4(x) = -2x^4 + 1$;

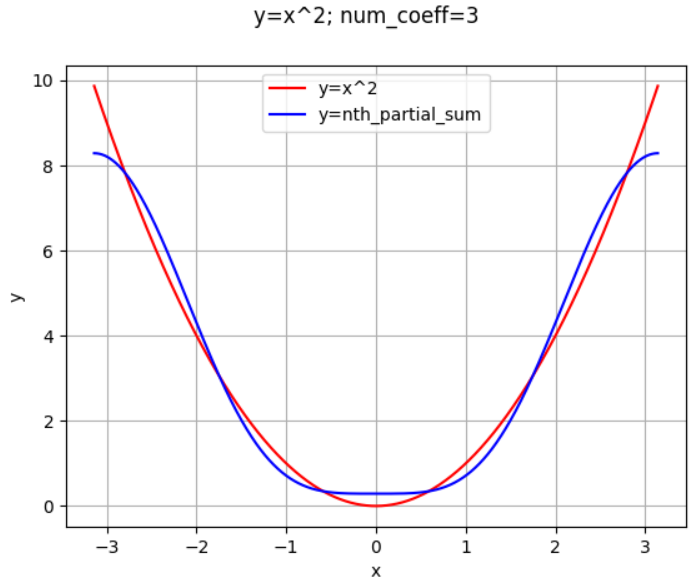

Figure 2: Plots of $y = x^2$ and $s_3(x)$ on $[-\pi, \pi]$.

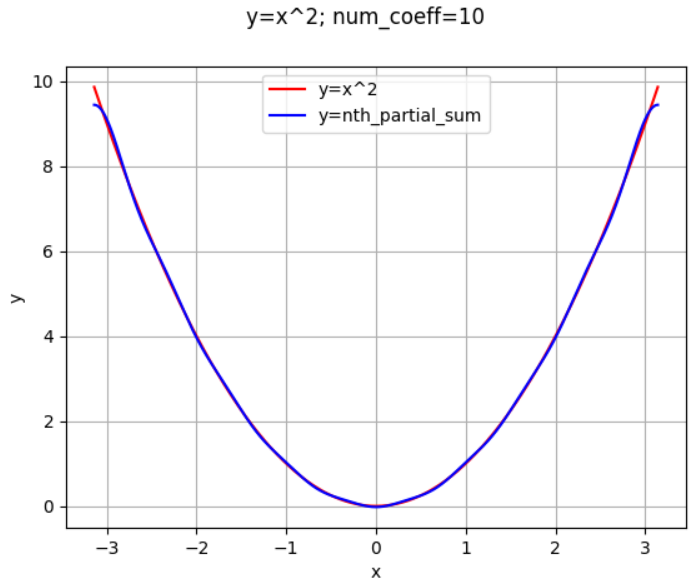

Figure 3: Plots of $y = x^2$ and $s_{10}(x)$ on $[-\pi, \pi]$.

5. $f_5(x) = cos(x)$;
6. $f_6(x) = cos(x) + 2cos(2x)$;
7. $f_7(x) = cos(x) + 2cos(2x) + 3cos(3x) + 4cos(4x) + 5(cos5x)$;
8. $f_8(x) = |x|$;
9. $f_9(x) = |x^5 + 3x + 2|$;
10. $f_{10}(x) = |sin(x) + 2sin(2x) + 3sin(3x)|$.

Specifically, for each function f_i, you need to implement `test_nth_partial_sum_fun_0i()` and `test_plot_error_fun_0i()`. For example, to analyze $f_7(x)$, you'll implement

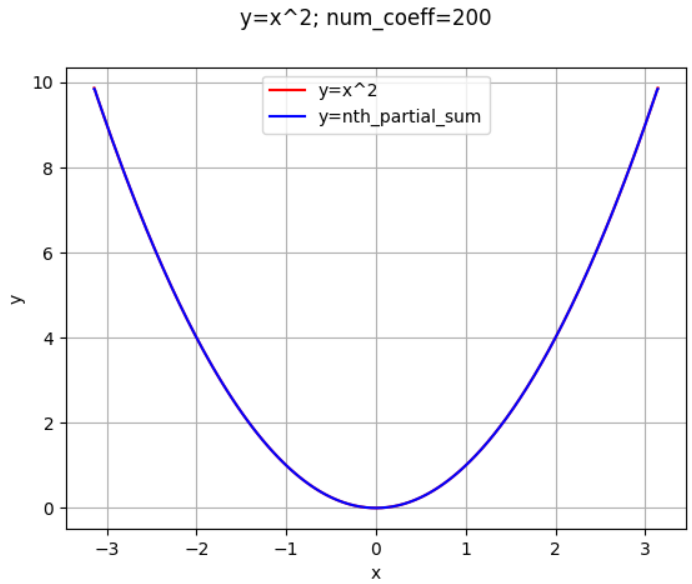

Figure 4: Plots of $y = x^2$ and $s_{200}(x)$ on $[-\pi, \pi]$.

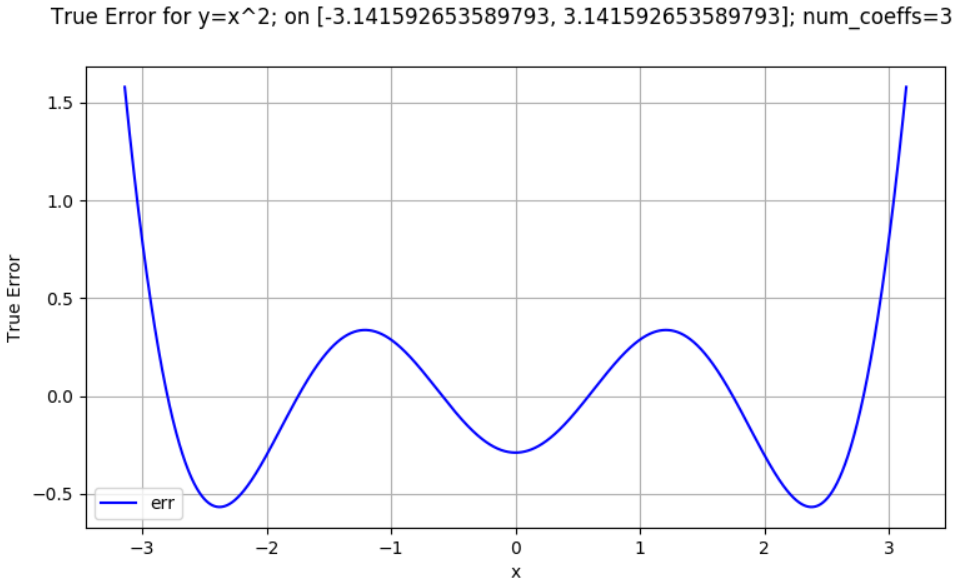

Figure 5: True error between $y = x^2$ and $s_3(x)$ on $[-\pi, \pi]$.

`test_nth_partial_sum_fun_07()` and `test_plot_error_fun_07()` in `cs3430_s21_hw08_uts.py`. You can follow my code in `test_nth_partial_sum_fun_00()` and `test_plot_error_fun_00()`.

I've carefully chosen these 10 functions, because they don't have any jump discontinuities or out-of-phase issues on $[-\pi, \pi]$. Your Fourier series should approximate them well.

In a comment next to each `test_plot_error_fun_0i()`, briefly state the number of coefficients and the Romberg lattice spot that gave you the smallest true error for the function.

My personal favorite plot for this assignment is in Figure 8. Error can also be beautiful, can it not? My code generated this plot when I approximated $f_{10}(x)$ with the 13-th Romberg (i.e., $R(13, 13)$) and 200 Fourier coefficients.

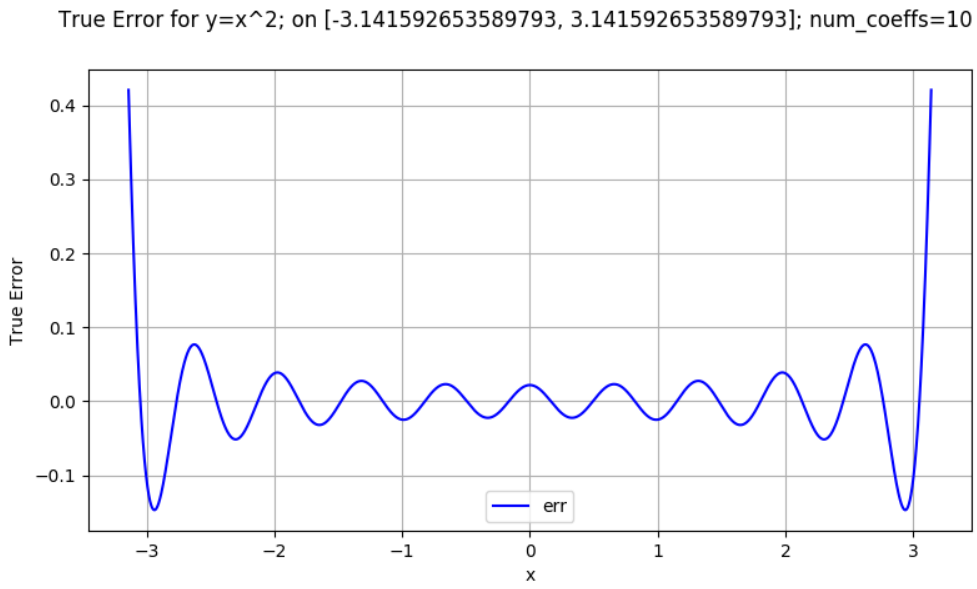

Figure 6: True error between $y = x^2$ and $s_{10}(x)$ on $[-\pi, \pi]$.

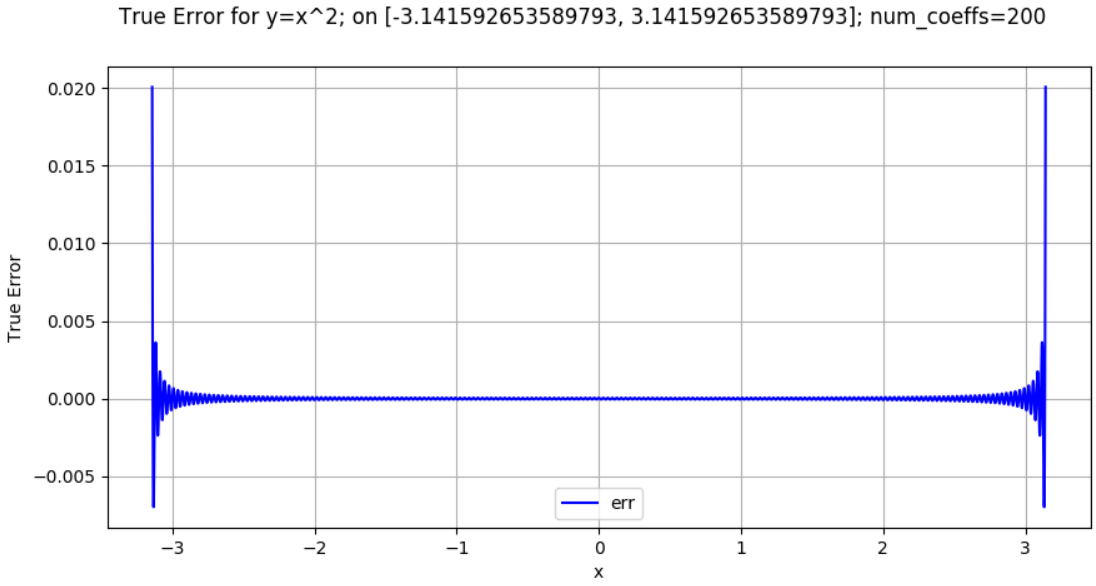

Figure 7: True error between $y = x^2$ and $s_{200}(x)$ on $[-\pi, \pi]$.

Problem 3 (2 points)

Let's do some audio processing. The zip for this assignment contains two directories with wav files – `Guitar` and `Violin`. The files `Guitar_A_.wav`, `Guitar_D_.wav`, `Guitar_E_.wav`, and `Guitar_D_.wav` in `Guitar` contain the recordings of the notes A, D, E, and G, respectively, that I played on my guitar. The files `Beautiful_Gypsy_Lick_1_.wav`, `Beautiful_Gypsy_Lick_2_.wav`, `Beautiful_Gypsy_Lick_3_.wav`, `Beautiful_Gypsy_Lick_4_.wav` contain my recordings of four licks that constitute the melody of "Beautiful Gypsy," an original tune by Colin McAllister. Dr. McAllister is a professor of music at the University of Colorado, Colorado Springs, and happens to be one of my favorite acoustic guitarists. In case you've never heard the term *lick*, it's a sequence of notes. Many jazz and rock

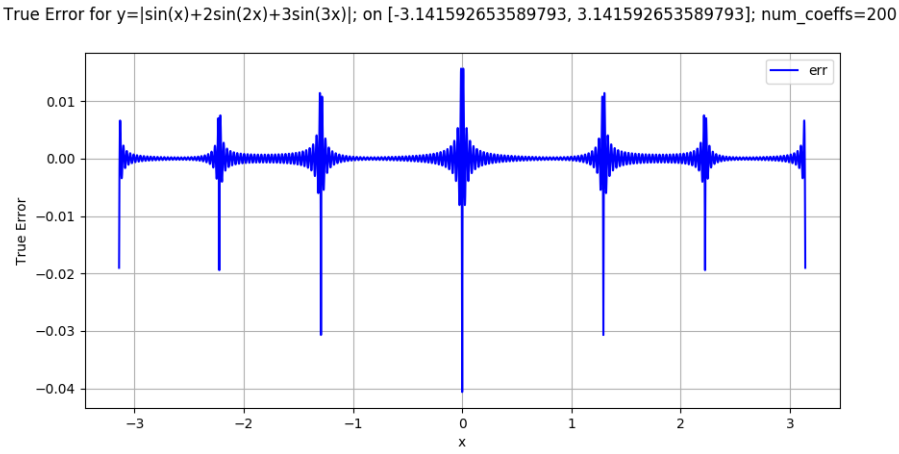

Figure 8: True error between $y = f_{10}(x)$ and $s_{200}(x)$ on $[-\pi, \pi]$.

guitar melodies and improvizations can be viewed as sequences of licks.

The files `Violin_A_.wav`, `Violin_D_.wav`, `Violin_E_.wav`, and `Violin_D_.wav` in `Guitar` contain the recordings of the notes A, D, E, and G, respectively, that I played on my violin. I deliberately scrated the strings with my bow on the first two beats of `Violin_A_.wav` and `Violin_D_.wav` to introduce a small amount of noise into the recordings. Some master violinists do it for artistic effects. But, I'm not that good and did it just for some noise injection.

Here's how we can load a wav file into Python.

```
from scipy.io import wavfile

def read_wavfile(fpath):
    return wavfile.read(fpath)
```

Let's run `test_wavfile()` in `cs3430_s22_hw08_ut.py` to see the values we get when we load a wav file.

```
    def test_wavfile(self):
        fs, amps = read_wavfile('Guitar/Guitar_A_.wav')
        print('\nGuitar_A_.wav data:')
        print('frequency = {}'.format(fs))
        print('num amp readings = {}'.format(len(amps)))
        print(amps[:5])

        print('\nGuitar_D_.wav data:')
        fs, amps = read_wavfile('Guitar/Guitar_D_.wav')
        print('frequency = {}'.format(fs))
        print('num amp readings = {}'.format(len(amps)))
        print(amps[:5])

        print('\nGuitar_E.wav data:')
        fs, amps = read_wavfile('Guitar/Guitar_E_.wav')
        print('frequency = {}'.format(fs))
        print('num amp readings = {}'.format(len(amps)))
        print(amps[:5])
```

When you run the above test, you should see the following output.

```
Guitar_A_.wav data:
frequency = 44100
num amp readings = 415422
[[ 0 -1]
 [-3 -3]
 [-2 -2]
 [-2 -3]
 [-5 -5]]

Guitar_D_.wav data:
frequency = 44100
num amp readings = 441441
[[-2 -2]
 [-2 -3]
 [-3 -2]
 [-1  1]
 [ 0 -1]]

Guitar_E.wav data:
frequency = 44100
num amp readings = 397341
[[ 0  0]
 [-2 -1]
 [-4 -3]
 [ 1 -1]
 [ 0 -2]]
.
----------------------------------------------------------------------
Ran 1 test in 0.010s
OK
```

A call to `read_wavfile()` returns two values (i.e.,, fs and amps). The first one is the frequency at which the file was recorded (e.g., 44100) and an array of amplitude readings (these are integers). When a wav file is recorded with 2 channels, each amplitude reading consists of two integers (one for each channel). A mono wave file will contain only one amplitude.

To keep things simple, we'll use the readings from channel 0 in our audio processing. Here's how it can be done. We get the frequence and amps from a wav file and then use list comprehension to get a numpy array of channel 0 amplitudes. Note that I call the amplitude data `f_of_t` to emphasize that the amplitude readings are produced by some function of time t (i.e., $f(t)$). We may not know what the function is, but we assume that it exists.

```
fs, f_of_t = read_wavfile(fpath)
### take the amps from channel 0.
f_of_t = np.array([amp[0] for amp in f_of_t])
```

There are three more pieces of information that we need to start hunting for the coefficients: 1) the time line; 2) Δt (i.e., the approximate difference between every pair of consecutive time ticks); 3) the measure of our half period. Since we are recovering coefficients on the interval $[-\pi, \pi]$, our half period is always π. Take another look at the formulas for a_n and b_n above. To compute a_n or b_n, we divide the integral by the half period. So, here's how we get these three pieces in place.

```
t = np.linspace(-math.pi, math.pi, len(f_of_t))
dt = t[1] - t[0]
half_period = math.pi
```

Now we can extract the n-th (or the k-th) a and b coefficients from the data.

```
def a_coeff(data, half_period, t, dt, n):
    vf = np.vectorize(lambda t: math.cos((math.pi*n*t)/half_period)*dt)
    return sum(data*vf(t))/half_period

def b_coeff(data, half_period, t, dt, n):
    vf = np.vectorize(lambda t: math.sin((math.pi*n*t)/half_period)*dt)
    return sum(data*vf(t))/half_period
```

Note that we do not use any integration here, because the integral is approximated with a Reimann sum.

Implement the function `recover_fourier_coeffs_in_range(fpath, lower_k = 0, upper_k=50)` that takes the path to a wav file and lower and upper bounds of the a and b coefficients and returns two arrays – the array of a-coefficients and the array of b-coefficients. Below are a couple of runs. Your floats may be slightly different.

```
>>> from cs3430_s22_hw08 import recover_fourier_coeffs_in_range
>>> acoeffs, bcoeffs = recover_fourier_coeffs_in_range('Guitar/Guitar_A_.wav',
                                                       lower_k=0, upper_k=3)
>>> acoeffs
[-1.9655337597097, -0.01041777725028438, 0.0007193855463934214,
-0.010668891764580152]
>>> bcoeffs
[0.0037368997636770688, -0.005942155223967613, -0.007589167312960408]
>>> acoeffs, bcoeffs = recover_fourier_coeffs_in_range('Guitar/Guitar_A_.wav',
                                                       lower_k=200, upper_k=210)
>>> acoeffs
[0.13722870475409696, 0.10055640436382708, 0.11002012559696149,
-0.12568215270645233, -0.092242329291388, 0.03856262166517736,
-0.009747777413318646, 0.04298561534440871, 0.18116733787453954,
-0.126500431900981, -0.07207863616862037]
>>> bcoeffs
[-0.10627617684276534, -0.01088611185651106, 0.0553202741882532,
-0.03366047808261343, 0.13368286315765812, 0.0951395517007761,
-0.028563372760522013, 0.17982965274587778, -0.09021503266256628,
-0.0919161686765141, 0.14661621829438923]
```

Note that when `lower_k` $= 0$ the number of a-coefficients is $1 +$ the number of b-coefficients, because there is no b_0.

The function `plot_recovered_coeffs()` in `cs3430_s22_hw08.py` gives you a tool that you can use to scatter plot the recovered coefficients.

Can we tell how similar different musical instruments sound? We can certainly try. The site en.wikipedia.org/wiki/Piano_key_frequencies gives the frequencies in Hz of the standard piano notes. For example, A, G, E, and D notes in the 7-th octave have the following frequencies: A7 – 3520.000, G7 – 3135.963, E7 – 2637.020, and D7 – 2349.318.

Let's discover how much each of these four notes is present in `Beautiful_Gypsy_Lick_1_wav`. This is done in `test_ut_10()` in `cs3430_s22_hw08_uts.py`. The recovery logic is the same for each note. For example, we know that A7's frequency is 1760. Thus, we recover and a- and b-coefficients in the

range that starts below 1760 and ends above 1760. Why not exactly at a given frequency? Because of the frequency spill (some notes whose frequencies are close spill over each other's bands) and because we're computing coefficients at discrete (i.e., non-negative integer) frequencies.

In the lines below, the frequency range for A7 is $[3518, 3522]$ (i.e., slightly above and below 3520. The recovered coeffs are plotted with `plot_recovered_coeffs()`.

```
acoeffs, bcoeffs = recover_fourier_coeffs_in_range('Guitar/Beautiful_Gypsy_Lick_1_.wav',
                                                   lower_k=3518, upper_k=3522)
plot_recovered_coeffs(acoeffs, 'ACOEFFS: RCD={}; RCV={} in [{},{}]', 'Beautiful_Gypsy_Lick_1',
                      'A7', lower_k=3518, upper_k=3522)
plot_recovered_coeffs(bcoeffs, 'BCOEFFS: RCD={}; RCV={} in [{},{}]', 'Beautiful_Gypsy_Lick_1',
                      'A7', lower_k=3518, upper_k=3522)
```

Figures 9 and 10 show the a- and b-coefficients computed from the 0-th channel amplitudes of `Beautiful_Gypsy_Lick_1_.wav`. The title of each plot is self explanatory. It starts with the string "ACOEFFS" or "BCOEFFS" to indicate what type of coefficients are being scatter plotted. The string "RCD=..." shows the name of the recorded wav file. The string "RCV=..." specifies the note being recovered. The closed interval specifies the frequency range in which the note is recovered.

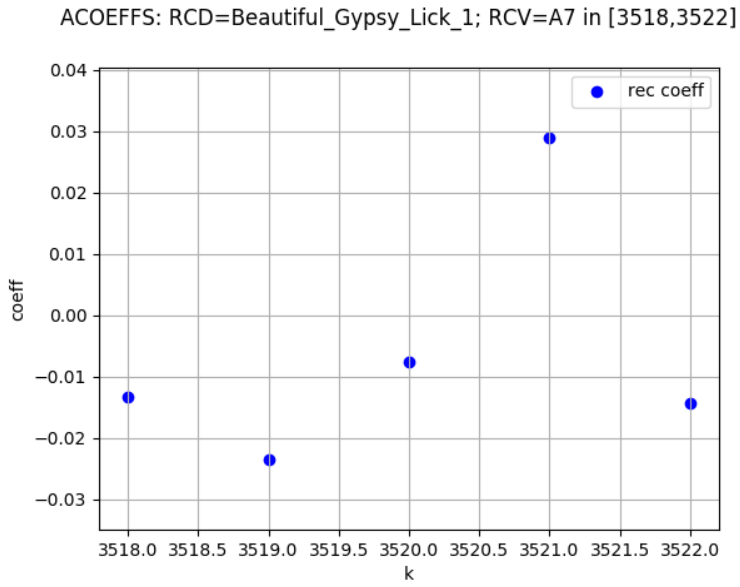

Figure 9: A7 a-coefficients from Beatiful Gypsy Lick 1.

How do we know that a note is present? That's more of an art than a science. In other words, we have to threshold on the magnitude of the coefficient. Let's arbirarily decide that if at least one of the coefficients in the range is greater than 0.1 or smaller that –0.1, then the note is present. Looking at figures 9 and 10 we conclude that A7 doesn't appear to be present in the 0-th channel amplitudes of `Beautiful_Gypsy_Lick_1_.wav`.

How about D7? Figure 11 suggests that it is present.

E7? Both figure 13 and 14 suggest a strong presence.

G7? Both figure 15 and 16 also suggest a strong presence.

Implement the unit test `test_ut_11()` to discover whether A7, D7, E7, and G7 are present in `Beautiful_Gypsy_Lick_2_.wav`. State your comments which notes are present and which are absent according to your plots.

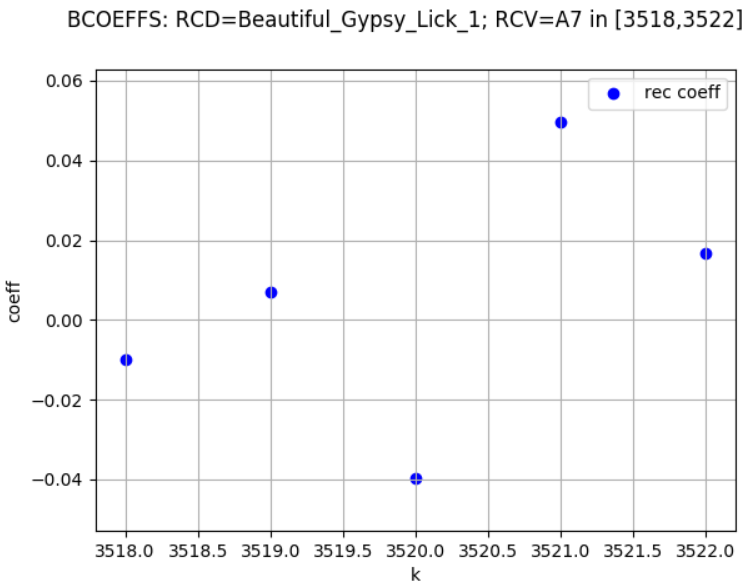

Figure 10: A7 b-coefficients from Beatiful Gypsy Lick 1.

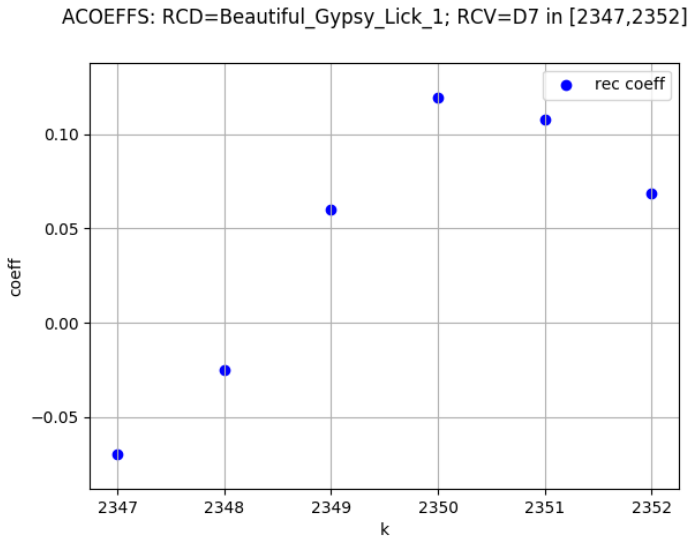

Figure 11: D7 a-coefficients in Beatiful Gypsy Lick 1.

Transcendental Meditation on π

All of the above computation is based on the beautiful number π. Since π has featured so prominently in Lectures 15 and 16 and this assignment, let's do some transcendental meditation (pun intended!) on this great number to pay tribute to it. Those of you who attended Lecture 16 already did some of it. But, I don't want to leave anybody behind.

This great number has been with us for quite some time. Babylonian astronomers were approximating π in base 60 around 2,000 BCE, all of which they learned from the Summerian astronomers, a much more ancient civilization.

You may know that number theorists divide all numbers into *algebraic* and *transcendental*. A number is algebraic if it is a solution to a polynomial equation with integer coefficients (e.g., $x^2 - 4 = 0$). If a number is not a solution to any such equation, it is transcendental. All integers are algebraic. All

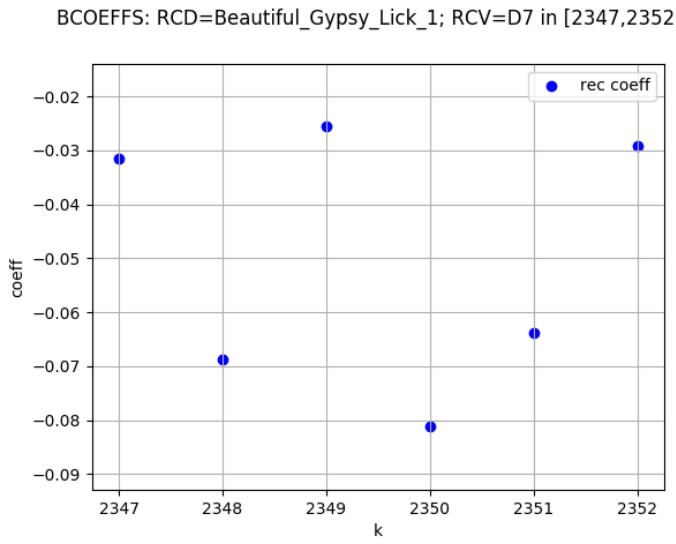

Figure 12: D7 b-coefficients in Beatiful Gypsy Lick 1.

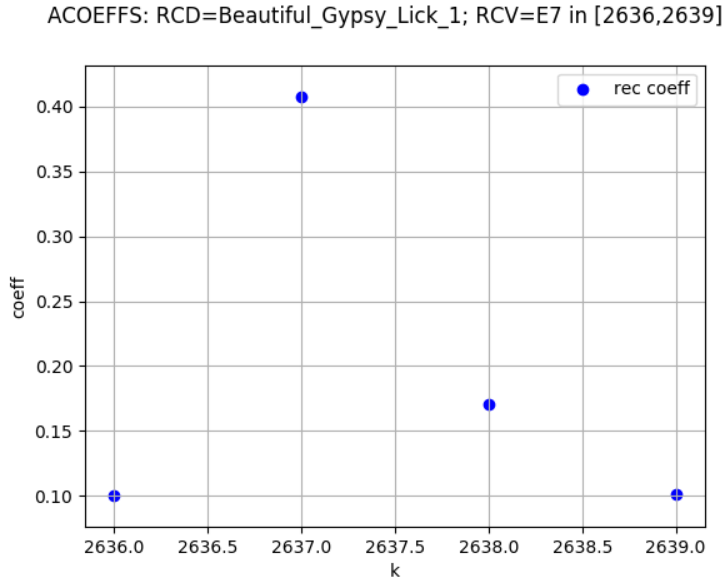

Figure 13: E7 a-coefficients in Beatiful Gypsy Lick 1.

rationals are algebraic.

Some irrational numbers are also algebraic. For example, $\sqrt[3]{5}$ is a solution to $x^3 - 5 = 0$. We can also construct an equation that solves with

$$\sqrt[5]{\frac{7 + \sqrt[11]{13}}{17}}.$$

Around 60 CE the Indian mathematician Aryabhata approximated π with 3.1416. Around 1400 CE, the Persian mathematician Ghyath ad-din Jamshid Kashani computed π correctly to 16 digits.

Ludolph van Cuelen used a method or Archimedes' to compute the first 35 digits of π. He requested that these digits be carved on his gravestone. His request was granted at the time of his death in

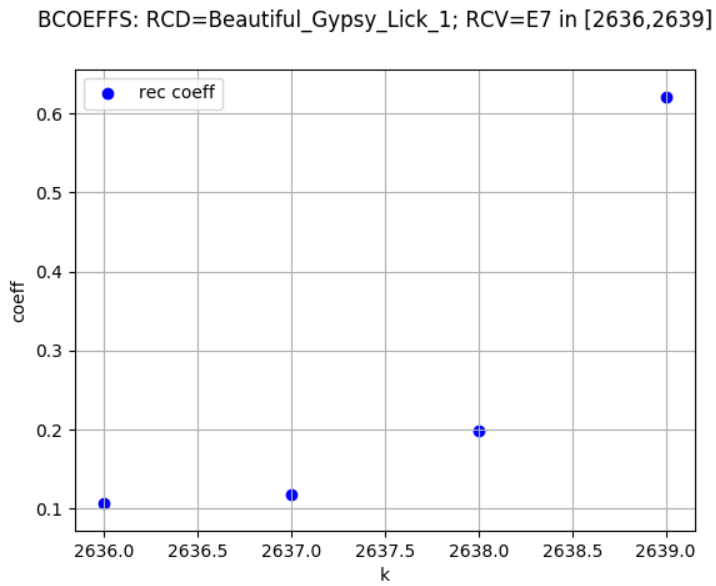

Figure 14: E7 b-coefficients in Beatiful Gypsy Lick 1.

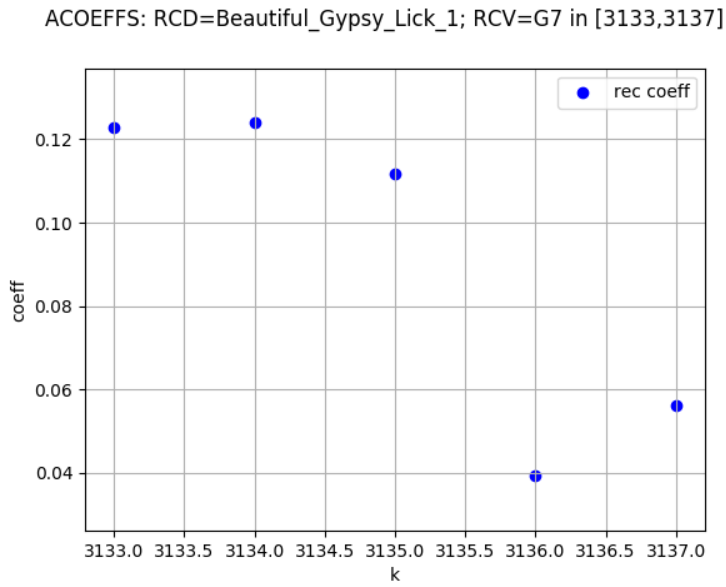

Figure 15: G7 a-coefficients in Beatiful Gypsy Lick 1.

1610.

In 1873, William Shanks computed the first 707 digits of π. It took him 20 years to accomplish this task. In 1944, D. F. Ferguson discovered that Shanks made a mistake on the digit in the 528-th position. Oh well! At least the first 527 digits were correct!

In 1873, Johann Lambert showed that π is an irrational number. Nine years after Lambert's discovery, in 1882, Ferdinand von Lindemann proved that π is transcendental. In other words, π doesn't solve any polynomial equation with integer coefficients.

In 1897, the Indiana General Assembly contemplated a bill declaring that π was equal to 3.2. Fortunately, the bill was tabled and never voted on. Just imagine living in a place where it's a crime to approximate π with 3.14! Things don't go well when politics interferes with science and

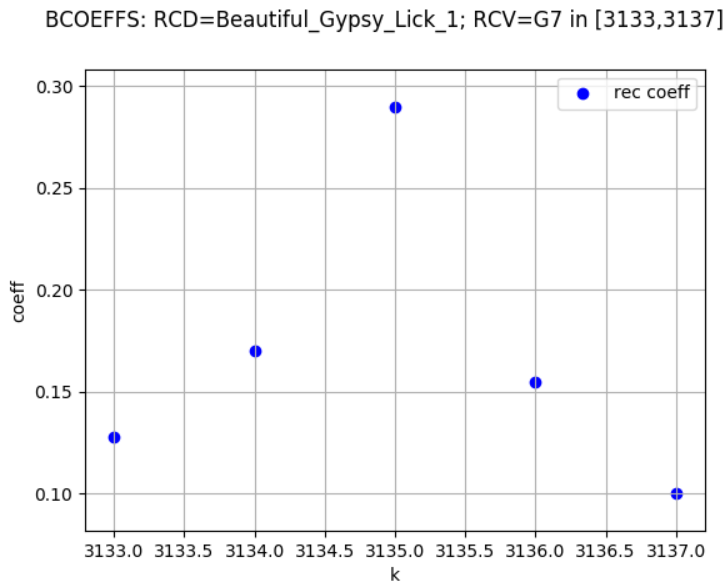

Figure 16: G7 b-coefficients in Beatiful Gypsy Lick 1.

mathematics.

It's now about 10:00pm on 3/19 when I'm putting the finishing touches on this assignment. Five days ago on 3/14 at 1:59pm, or at 1:59am, which is more accurate, some number theory enthusiasts all over the world celebrated the International π Day, right on 3.14159! Remember to join us next time!

What To Submit

Submit your code in `cs3430_s22_hw08.py`, `cs3430_s22_hw08_uts.py`, `rmb.py`, and any other files from your previous assignments that are necessary for us to run your code. Zip your files into `hw08.zip`, and upload your zip in Canvas.

Happy Hacking!