CS 3430: S24: Scientific Computing
Assignment 09
Fourier Series and Coefficients

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 31, 2024

Learning Objectives

1. Fourier Coefficients and Series
2. Approximation of Functions with Fourier Series
3. Detection of Harmonics in Audio Files

Introduction

In this assignment, we will approximate simple, well-behaving functions with Fourier series by computing their coefficients with Romberg integrals. We will detect a few harmonics in audio files that record single string notes of the classical 6-string guitar and the classical 4-string violin. You will save your solutions in `cs3430_s24_hw09.py` and, in this assignment, in `cs3434_s24_hw09_uts.py`. The unit tests for this assignment are in `cs3434_s24_hw09_uts.py`. You should install `scipy`, because we will use the class `wavfile` from `scipy.io` to process `.wav` files.

Optional Problem 0: (0 points)

You may review the slides and/or your class notes for Lectures 17 and 18 on Fourier Series and Coefficients and become comfortable with periodic functions, harmonics, trigonometric polynomials, Fourier series, and Fourier coefficients. You may also want to review Lecture 14 on Romberg Integration. We will Rombergs in this assignment.

Problem 1: (1 point)

Recall that a Fourier series approximating some function $f(x)$ (cf. Slide 32, Lecture 18) is defined as

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

We can replace $\approx$ with $=$ if and only if we can prove that the infinite series on the right actually converges (in the technical sense of Calculus) to the function on the left, which may or may not be possible. Hence, the common and safe use of $\approx$.

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 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. Thus, we have inifinitely many such partial sums. 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 18 (cf. Slides 26–31) that we can compute the Fourier coefficients a_n, b_n in the n-th partial sum with the following formulas.

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

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

The integrals in the above formulas can be approximated with Rombergs (or any other integral approximation methods such as Riemann sums). But since we have implemented Rombergs in Assignment 07, we might as well reuse them in this problem.

Given a function $f(x)$, we can compute the first $n + 1$ a_k coefficients and the first n b_k coefficients and then, after the coefficients are computed, approximate $f(x)$ at a given value of x with the n-th partial sum s_n. E.g., if we have 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

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

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

Implement the function `nth_partial_sum_of_fourier_series(x, acoeffs, bcoeffs)` that takes a real number x, an array of $n+1$ real a_k coefficients and an array of n real b_k coefficients and returns the value of $s_n(x)$. For this problem, we assume that the coefficients have been computed elsewhere (cf. Problem 2). In other words, this function computes s_n for a specific value of n. Remember to divide a_0 by 2!

Problem 2 (2 points)

Given a function $f(x)$, we can approximate it with a Fourier series by ploting the function and computing and plotting several partial sums for different numbers of coefficients on a given interval. We can also plot the true error between the values of the function and different partial sum approximations on the same interval.

Let us approximate $f(x) = x$ on $[-\pi, \pi]$ with $s_{20}(x)$ and plot both curves. Figure 1 shows the two plots. The partial sum approximation is sufficiently close, except at the ends, which is frequently the case.

Do the same Fourier series approximation analysis in `cs3430_s24_hw09_uts.py` for the following 10 functions on $[-\pi, \pi]$, i.e., for each function, generate 2 plots – the plot of the function and its approximation with s_{20} and then the plot of the tree error of the partial sum approximation.

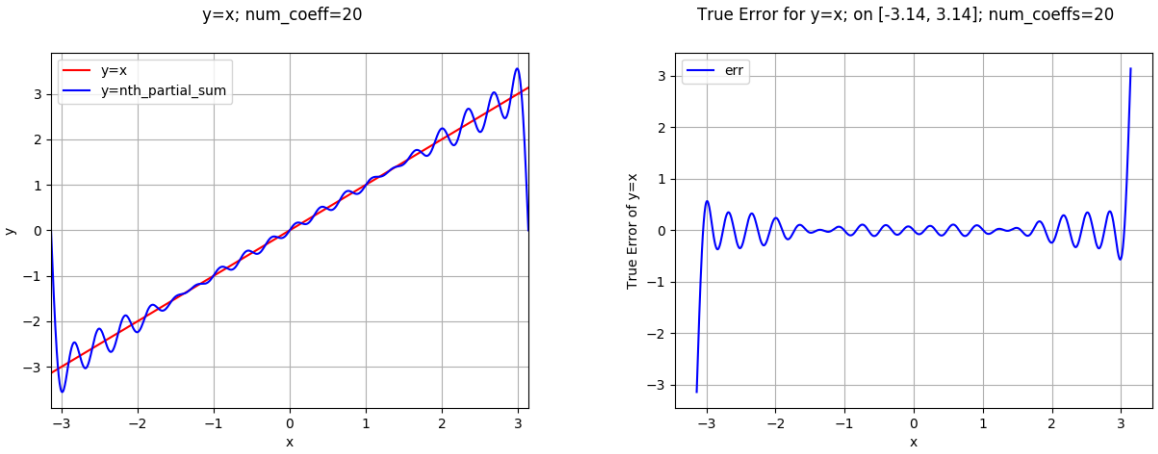

Figure 1: Plots of $y = x$ and $s_{20}(x)$ and the true error on $[-\pi, \pi]$.

1. $f_1(x) = x^2$;
2. $f_2(x) = x^3$;
3. $f_3(x) = x + 1$;
4. $f_4(x) = x^5$;
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. $f_{10}(x) = |sin(x) + 2sin(2x) + 3sin(3x)|$.

Problem 3 (2 points)

On to 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 acoustic 6-string guitar. The files `Beautiful_Gypsy_Lick_1.wav`, and `Beautiful_Gypsy_Lick_2.wav` contain my recordings of four licks from 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. A lick is just a sequence of notes, fingered or picked. Some classical, rock, and jazz guitarists view melodies and improvizations as lick sequences. The files `Violin_A.wav`, `Violin_D.wav`, `Violin_E.wav`, and `Violin_D.wav` in `Violinr` contain the recordings of the notes A, D, E, and G, respectively, that I played on my 4-string fiddle.

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

```
from scipy.io import wavfile
def read_wavfile(fpath):
    return wavfile.read(fpath)
```

If you run `test_guitar_wavfile` and `test_violin_wavfile` in `cs3430_s24_hw09_uts.py`, you will see the following output.

```
Violin_A.wav data:
frequency = 44100
num amp readings = 459081
[[-7 -6]
 [-6 -6]
 [-3 -2]
 [-5 -4]
 [-6 -6]]

...
Violin_E.wav data:
frequency = 44100
num amp readings = 406161
[[-2 -2]
 [-3 -4]
 [-3 -3]
 [-2 -3]
 [-4 -3]]
...
```

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 will use the readings from channel 0 in our audio processing. Here's how it can be done. We get the frequency and amps from a wav file and then use list comprehension to get a numpy array of channel 0 amplitudes. Note that I deliberately call the amplitude data `f_of_t` to emphasize that the amplitude readings are produced by some function of time t. We may not know what the function is, but assume that it exists.

```
fs, f_of_t = read_wavfile(fpath)
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); and 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. 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

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))/math.pi

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))/math.pi
```

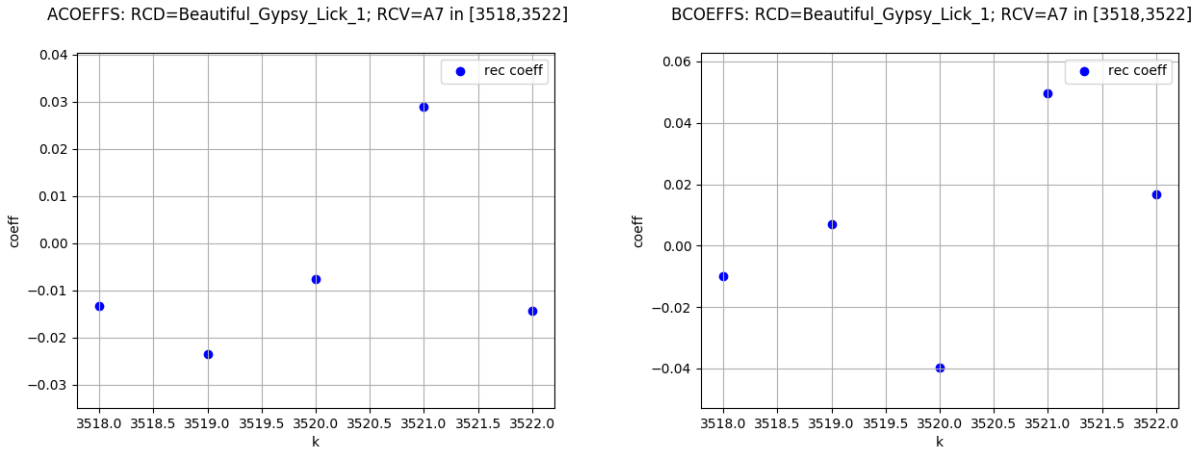

Figure 2: A7 Harmonics in Lick 1

Note that we do not use any integration here, because the integral is approximated with a Reimann sum and then divided by π to obtain the coefficient.

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_s24_hw09 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_s24_hw09.py` gives you a tool to scatter plot the coefficients. The site en.wikipedia.org/wiki/Piano 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, D7 2349.318, E7 2637.020, and G7 3135.963. The plots in Figures , 3, 4, and 5 give the scatter plots of the the coefficients of the A7, D7, E7, and G7 harmonics recovered

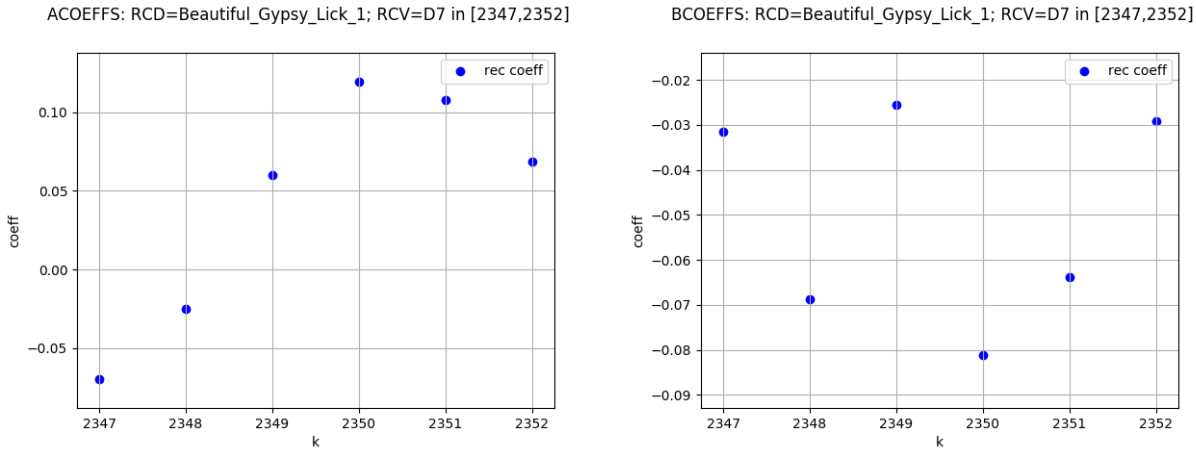

Figure 3: D7 Harmonics in Lick 1

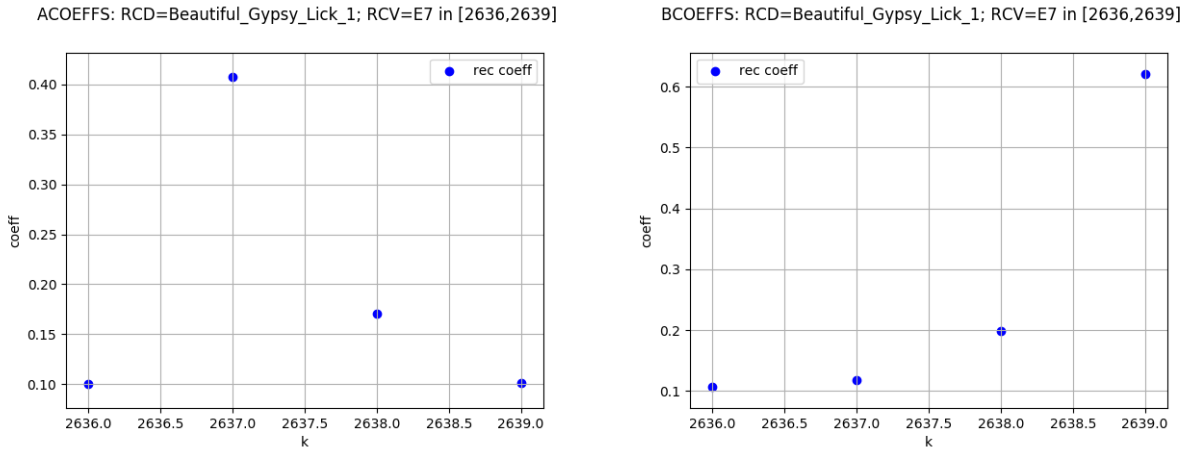

Figure 4: E7 Harmonics in Lick 1

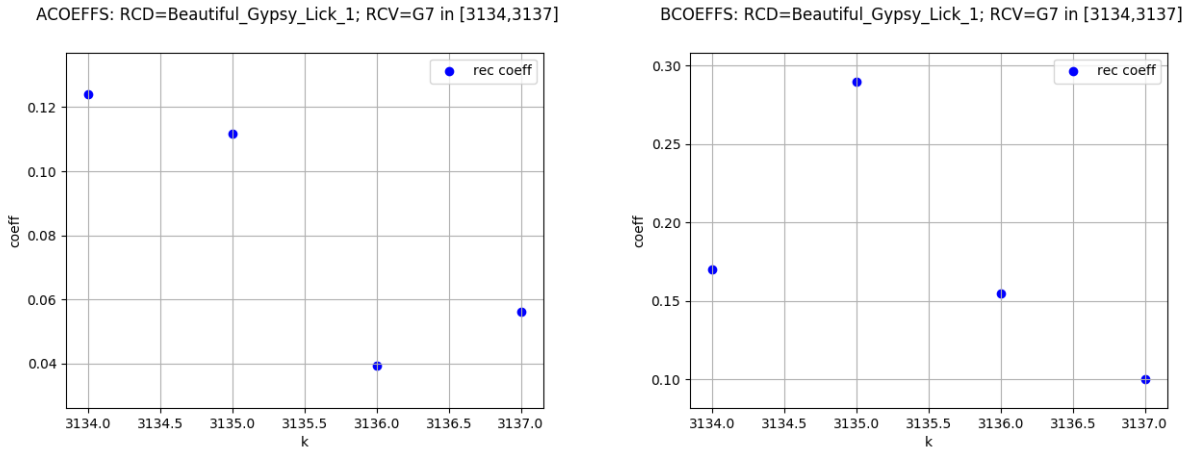

Figure 5: G7 Harmonics in Lick 1

from `Beautiful_Gypsy_Lick_1.wav`. From these plots, we can tell that A7 is weakly present, D7

is present somewhat stronger, and E7 and G7 are strongest.

Analyze the presence/abscence of the A7, D7, E7, and G7 piano harmonics in `Beautiful_Gypsy_Lick_2.wav` and `Violin_A.wav` by filling out the appropriate stubs in `cs3430_s24_hw09_uts.py`.

What To Submit

Submit your code in `cs3430_s24_hw09.py`, `cs3430_s24_hw09_uts.py`, `rmb.py` and any other files from your previous assignments for me to run your code the unit tests. You do not have to save any plots. I will look at them when I run the unit tests. Zip your files into `hw09.zip`, and upload the zip in Canvas.

Happy Hacking!