PIC 16, Winter 2018 – Preparation 7F

Assigned 2/16/2018. To be completed by class 2/23/2018.

Intended Learning Outcomes

By the end of this preparatory assignment, students should be able to:

- load/save data of several types, especially audio, using `scipy.io`,
- learn about `scipy.special` math functions as needed,
- solve linear systems of equations and perform other linear algebra operations using `scipy.linalg`,
- choose a specialized linear equation solver for linear systems with special properties, and
- compute the FFT and IFFT of signals using `scipy.fftpack`.

Tasks

- ☐ Read the introduction to Scipy Lectures 1.5.
- ☐ Unless you're a Matlab or Octave user, the most important part of 1.5.1 is the "See also" portion at the end and the link to the scipy.io reference, because we're going to be loading audio.
- ☐ Sound is pressure waves in air. If you want to know more, watch the first minute of this.
- ☐ Any wave can be represented as a sum of sines and cosines. For more, watch this.
- ☐ We represent sound waves digitally by sampling the pressure of the air many times per second. For more, watch this.
- ☐ To prepare for Assignment 7F, get familiar with loading `.wav` file (audio) data.
  - Download AMajor.wav, which I generated using SciPy. Play it using any media player on your computer (you might want to turn the sound down first and calibrate once it's playing).
  - Load the data in Python. Plot the first 1000 samples (data points).
  - Using the amplitude, sample rate, and *data type* of that sound file for reference, generate two seconds of the note A440 ($y = A\sin(440 \cdot 2\pi \cdot t)$, where is A the desired amplitude) and save the data to `A440.wav`. You should be able to play your file in a media player, and it should sound like `A440.wav`. It should *not* sound like `A440_bad.wav`.[1]
- ☐ The special functions from 1.5.2 can come in handy for scientists and engineers. If you've ever wondered what a Bessel function looks like, try plotting one using the `jn` function. The first argument is the order (any integer) and the second is an array of `x` values. Try different (low) orders.
- ☐ Follow 1.5.3.
- ☐ Take a look at the documentation for the `scipy.linalg.solve` function. Generate a random linear system of equations $Ax = b$ by generating a 10×10 random NumPy array `A` and a random 10×1 `b`, then use the solve function to find the solution `x`. Verify that your solution is correct by checking that the Euclidean norm of $Ax - b$ is small. You've seen the functions for creating random matrices and the method for taking a matrix product before. You may need to look up the function for taking the matrix or vector norm, or you could just guess…
- ☐ Now try solving the same equation with the formula $x = A^{-1}b$ where A^{-1} is the inverse of A. Verify that your two solutions are the same (to reasonable accuracy; there may be a small difference).

---

[1] If it does, you didn't pay attention to the data type. Shame!

- Randomly generate a system of 800 linear equations (instead of just 10) and measure the time it takes to solve using each of the previous two methods. Do this several times (using a loop) and take the minimum for each solution technique. You will find that the `solve` function is significantly faster because it implements techniques for solving linear equations that are faster than inverting a matrix. Check out all the functions for solving special linear systems in the `scipy.linalg` documentation. If your system meets certain requirements (e.g. the matrix is positive definite or triangular), solution can be much faster!
- Never solve a linear equation by taking the inverse of the matrix again. In fact, avoid taking the inverse of a matrix ever again (as you can probably get around it)!
- Fourier had a great idea: functions can be represented as the sum of sines and cosines of different frequencies. For more, watch this. The second half of the PIC 16 Track A Math Review Part II (starting at 22:30) might also be helpful. I hope that this primer I wrote helps if you want a deeper understanding of the FFT than the course requires.
- Follow 1.5.8. The Fast Fourier Transform does essentially the same thing as the Fourier Transform, but for discrete data like our audio signal. For technical reasons, it returns an array of *complex* numbers, so pay attention to how the tutorial takes the *magnitude* of each of these complex numbers (yielding the `power`) and generates the list of frequencies that each power corresponds with (using `fftfreq`). When it's done, the tutorial has extracted from the original signal the "power", or amount, of the sinusoids at each of the given frequencies. Assignment 7M will involve removing noise from a real audio signal, and is very similar to removing the high-frequency noise as in the tutorial.
- You do not need to review the worked examples unless you are interested. The exercise is much more difficult than the assignment will be, so you don't need to do that, either.