THE SOUND OF A MOLECULE

MUMT 307 FINAL PROJECT

by Jaden Chong

Introduction and Objectives:

Throughout my studies in my biochemistry degree at McGill, I found one important area of chemistry particularly interesting. Nuclear Magnetic Resonance (NMR) Spectroscopy is a widely used technique to identify the shape and atomic composition of molecules, and I wanted to somehow bridge the gap between chemical and audio science for this project. To briefly introduce the technique, NMR analysis consists of subjecting the sample to a massive magnetic field of up to 24 Teslas (4800 times stronger than a fridge magnet), which aligns the quantum spins of the atoms. Then, the sample is given a radio-frequency electromagnetic pulse, and the tiny perturbations in the magnetic field are recorded. Analysis of the recorded data provides evidence for the shape and atomic composition of the molecule. A simpler way to think about this is with a relevant analogy; think of the molecules as "strings". When they are placed into the magnetic field, the "strings" are being tensed. The radio-frequency electromagnetic pulse is analogous to "plucking the string"; the vibrations are then recorded. NMR analysis of a molecule is like analysis of the sound from a plucked or struck instrument.

When we learned about Fourier transforms in MUMT 307, I was reminded of the fact that the analysis of NMR data also involves Fourier transforms. The different magnitudes and frequency components of NMR data are what allows the analysis of the molecule, just as the magnitudes and frequency components of sound data allow analysis of sounds. From this, I wanted to try to take NMR data and play it as a sound. To implement this, I wanted to use MATLAB.

Overall, my objectives for this project were as follows:

1. Acquire NMR data for various molecules.
2. Import the NMR data into MATLAB and play it as a sound.
3. Implement a MATLAB script that attempts to automatically recreate the generated molecule sound.

The Development Process

Firstly, I had to acquire raw NMR data. Usually what is analyzed in chemistry is the result of the Fourier transform, but I what I needed to play as a sound was the raw data of the NMR spectrometer, called the "free induction decay", or FID. I emailed the spectroscopy department of McGill, and they kindly referred me to an NMR simulation software called MestReNova (MNova). They also gave me one of ten existing McGill licenses to use this powerful software.

In MNova, you can draw any molecular structure you want, and using a massive database of known magnetic field responses of thousands of molecules, it will predict the NMR results of your drawn molecule to excellent accuracy.

The next step was to find a way to import it into MATLAB. The simplest method was to just save the FID as cartesian coordinates in an ASCII file, in a ".csv" format. For the purposes of this project, I only drew 5 molecules to test: Glutamate (MSG without the sodium), ethanol, THC, testosterone, and sugar. After saving them as .csv files, I could easily read them into MATLAB matrices using the function "readtable", which keeps the row and column organization of .csv files.

To play this data as a sound, I only needed the 2nd column – the one with the amplitudes – since they all corresponded to evenly-spaced values on the x-axis. Then, to play this sound, I could simply use MATLAB's "soundsc" function to play the amplitude values.

As a further challenge to myself, I decided to expand on the existing code so that it would automatically analyze the frequency spectrum of the molecule's sound, and then automatically re-synthesize the sound using additive synthesis.

The first step was to take the Fourier transform of the imported data, which I did using the "fft" function. Fourier transforms give a mirror-image result where only the first half is relevant, so I first had to discard the second half, which I did by setting all elements in the second half to zero. I didn't like that the Fourier transform matrix as it was in this state only had values in the first half, because that meant that during the collection of frequencies and amplitudes, the maximum frequency component would be limited to half of the actual frequency range. So, I decided to spread the values out evenly while maintaining the matrix size. The example below helps to visualize my intentions.

After cropping the Fourier transform, the data was something like this:
[1 2 3 4 5 0 0 0 0 0]
But I wanted it to look like this:
[1 0 2 0 3 0 4 0 5 0]

so that there could be a wider range of frequency components, instead of having to set the upper limit of the frequency to twice what you want the actual limit to be.

I achieved this by using the “interp1” function, after some trial and error.

The next challenge was to record the amplitudes and frequencies of all the major frequency components. This was broken down into two tasks: to pick all the local peaks, and then to filter out all the minor peaks to only leave the major ones.

For the first task, I took advantage of MATLAB’s “findpeaks” function, which returns all the local peaks into two matrices, one containing the x-values and the other with the y-values of each peak. One major downfall I realized after my first few tries was that since the peak picking only takes a few frequency components, I would lose the scaling and relative positions of the peaks relative to the frequency minimum and maximum; the lowest and highest frequency components would become the new limiting factor for the range of frequencies. Thus, to maintain the relative positions of all frequency components past the peak-picking, I had to guarantee that the maximum and minimum frequencies also passed the peak-picking. So, I set the second and penultimate amplitudes to 1, so that they would be considered as local maxima for the peak-picking function. Of course, this would result in the range being shrunk by 2 elements, as the first and last elements would not pass the peak-picking, but with the sound containing 32768 elements, this translates to a re-scaling of a factor of 2/32768, or 0.006%, which I found acceptable.

For the second task, I used MATLAB’s “find” function to comb through all the amplitudes in the picked peaks and removed all the amplitudes smaller than 10% of the maximum, to make the code more economical. After this peak-filtering process, I could then set the boundary amplitudes to zero again (The second and penultimate original indices become the first and last elements in the peak-picked matrix since they are the first and last peaks).

To scale the x-axis to the correct frequency range, I first normalized the matrix values so that everything was between 0 and 1, then multiplied every element by the user-specified upper frequency boundary.

After this, I set up a for loop that built the re-synthesized sound, one sinusoid per loop. The iterator would cycle through the values of the picked peaks to specify the amplitude and frequency of each sinusoid.

The synthesized sound was now complete, but now I had to try to recreate the amplitude envelope. To do this, I applied the chunking technique taught in the first few weeks of MUMT 307, where the original input matrix was reshaped into an N rows * M columns matrix where N is the number of samples per chunk and M is the total number of chunks. This way, every column represents one chunk. After taking the maximum value of each chunk, I could then use those points to interpolate an envelope function with the same matrix size as the resynthesized sound.

To keep the code elegant, I had to ensure that this chunking technique would work for any chunk size. Basically, the code ensures that if the number of samples in the input file is not evenly divisible by the number of chunks, it will append 0's to the end of the input matrix until it is evenly divisible by the chunk size. To do this, the number of samples in the input is divided by the desired number of chunks to find chunk size. Chunk size is rounded upwards to reach an integer value, and then the input matrix has zeroes appended until it is of size (chunk size * number of chunks). This allows any integer chunk size smaller than the input file size to be used that is greater than 1 (otherwise reshape and interp1 functions will not work).

The final output is then the additively-resynthesized sound multiplied by the envelope function.

The link below is a video I made with the 5 example molecules. Both the original NMR sound and the MATLAB-resynthesized sounds are played, along with the amplitude envelopes and frequency spectra.

Improvable Aspects:

In retrospect, I probably could have just scaled the x-axis to the correct frequencies before the peak-picking, which would have saved me all the trouble of having to preserve the scaling of the x-axis through the peak-picking. Also, I had expected the quality of the approximated amplitude envelope to increase with decreasing chunk size, but in fact for chunk sizes smaller than 5, the audio becomes noticeably distorted. Strangely, the accuracy of the envelope does not seem to increase with decreasing chunk sizes. This may have to do with the appending of zeroes, but I am not sure. Theoretically, having a chunk size of 1 would just mean the envelope function is equal to the original function itself, and I would simply be multiplying the original input with the resynthesized tone. I tried this, and somewhat unsurprisingly it did not sound good.

Conclusion

Overall, this was an extremely fun project. I gained a very good understanding of MATLAB processing and the details of analyzing, manipulating, and synthesizing sound data, and managed to hear what molecules sounded like in the process. There is always room for improvement, but this was a good start.