The application begins by importing the necessary libraries and saves the data of the given sound files as bytes objects. The rice code algorithm is then implemented in the form of a function and another function is also implemented to write the encoded information to a file.

There was a major flaw in the way I initially implemented these two functions. The major flaw was I saved the rice encoded information and wrote it to the file as a series of strings. I wrote each rice code as a string and appended a “\n” to each string so the file would be more human readable. What I didn’t realize at the time was that a “0” or a “1” written as a string is ASCII encoded and therefore takes up 8 bits. Not to mention the “\n” is also ASCII encoded and the rice codes for large numbers are very long. This led to my compression rates for the initial part of the project being terrible. For the case where K = 4, the compression rate was on the order of 1300% and for the case where K = 2, it was on the order of 3300%. I dealt with these issues in my further development of the rice coding algorithm.

Next, there are several functions written that convert a binary string to a decimal integer as well as vice versa. Then there are two more functions designed to read the encoded files, convert the strings to bytes objects and then write them back to the required .wav files.

Finally, we’ve reached the exciting part: the ideas for further development. So in the ideas for further development I figured out how to actually write the binary strings directly to a binary .bnr file. This immediately had a massive impact but still didn’t initially actually compress either of the files. What I noticed from the encoded files from the initial part of the project was that the rice encoded files contained a lot of large numbers which led to a very large number of “1”’s at the beginning of each rice code. I got around this by using run length encoding to save the number of “1”’s at the start of each rice code. I saved the maximum possible quotient for the given value of K being used and made sure the quotient was exactly that long by appending “0”’s to the front of the quotient string if need be. So, for example, when K = 4, M = 16 which means the maximum quotient for a byte would be 255 / 16 = 4. So the quotient was always length four for K = 4. The remainder stayed exactly the same as before, however, and if there was a quotient of 0, this wouldn’t be included in the code. So every code was either of length 9 or length 5. Each code started with either a 0 or 1 indicating the quotient was either 0 or greater than 0. If the quotient was greater than zero, this would be saved into the next four or six digits (depending on K) of the code. This would then be followed by the remainder in binary which was either 4 or 2 digits (also depending on K). Since the start and end of each quotient and remainder was now completely determined, the need for the barrier 0 was eliminated.

	Original Size	Rice (K = 4 bits)	Rice (K = 2 bits)	% Compression (K = 4 bits)	% Compression (K = 2 bits)
Sound1.wav	1.002 MB	0.935 MB	0.919 MB	93.3%	91.7%
Sound2.wav	1.008 MB	1.044 MB	1.088 MB	103.6%	107.9%