

csce2143
assignment four
pitch correction

John C. Lusth

Revision Date: December 10, 2005

Your assignment is to take the next step in turning an off-key singer into the next singing sensation! You will do this by correcting the pitch to the nearest semi-tone.

Introduction

In Western Music, there are 12 semi-tones named:

tonic

diminished supertonic

supertonic

diminished mediant

mediant

subdominant

diminished dominant

dominant

diminished superdominant

superdominant

diminished subtonic

subtonic

For this program, you may assume *equal temperament* (see http://en.wikipedia.org/wiki/Equal_temperament), in which the frequency of each successive semi-tone is calculated by multiplying the frequency of the previous semi-tone by the twelfth root of two. Thus if the tonic is middle C (261.62556 Hertz), then the superdominant (the note A) would have a frequency of

$$261.62556 * 2^{\frac{9}{12}}$$

or 440 Hertz. The power $\frac{9}{12}$ is used because the superdominant is the ninth semi-tone above the tonic and there are a total of twelve semi-tones. Note that this is not the only way to calculate superdominant-like notes. Under

just intonation (see http://en.wikipedia.org/wiki/Just_intonation), the frequency of note A (with the given note C frequency) would be

$$261.62556 * \frac{5}{3}$$

or 436 Hertz. See if you can tell the difference. Here are two files, one playing a C and an A using equal temperament and one using just intonation. The file with just intonation should sound a tiny bit fuller and rounder, with fewer high overtones. Tell me which one is which: a.wav and b.wav.

Methodology

If the pitch needs to decrease (i.e., the frequency needs to go down), the method for pitch correction is quite simple: one simply resamples the data to spread it out. For example, suppose the original data starts out...

```
0   1   2   3   4   5   6   7   8   9   ...
0 627 1254 1879 2501 3120 3734 4342 4945 5539 ...
```

...where the first row is the sample number and the second row the corresponding sample. Suppose further that you determine f_0 to be 269 Hz and thus the closest semi-tone has a frequency of 261.62556 Hz. We will call the frequency of the closest semi-tone the *desired frequency* or f_d . If we let each sample constitute a unit of time (sample 0 is at time 0, sample 1 is at time 1, and so on), the original samples should appear at the following times in the new data stream:

```
0 1.03 2.06 3.08 4.11 5.14 6.17 7.20 8.22 9.25 ...
0 627 1254 1879 2501 3120 3734 4342 4945 5539 ...
```

We calculate the new times by shifting the old times; we shift by multiplying the original time by the ratio of the original frequency to the desired frequency (we will call this ratio z).

Unfortunately, these new times are not integral and we need to find the sample values at $t = 0, 1, 2, \dots$. We accomplish that task through a process called *interpolation*. For example, suppose we wish to find the sample value at $t = 5$ in the new spectrum. Looking at the results above, we see that the desired value should lie between the values at 4.11 (2501) and 5.14 (3120). We call these two times the *encompassing shifted time window*. Intuitively, the desired value should be nearer to 3120 than 2501 because 5 is closer to 5.14 than 4.11. A *linear interpolation* is a simple way to capture this intuition. The desired value s can be found using the following formula (which implements a linear interpolation)...

```
s = (int) (data[a] + (data[a+1] - data[a]) / z * (t - az));
```

...where t is the integral time of interest in the new spectrum, z is the frequency ratio described above, az is lower edge of the shifted time window encompassing t , and a is the original unshifted time corresponding to az .

Using this formula, we can define a function that shifts a spectrum:

```
function shift(data,f_0,f_d)
{
  var t;
  var a;
  var z = f_0 / f_d;
```

```

t = 0;

while (t < size)
{
    a = (int) (t / z); /* find the unshifted lower edge */
    output((int) (data[a] + (data[a+1] - data[a]) / z * (t - a * z)));
    t = t + 1;
}
}

```

Note, as before, z is the original frequency divided by the desired frequency. If we used this idea on the original example data, we would produce the following output, where, again, the first row indicates the sample number (and the new time):

```

0   1   2   3   4   5   6   7   8   9 ...
0 609 1219 1827 2432 3035 3633 4225 4812 5392 ...

```

Notice how the samples are growing more slowly in the new spectrum. This is what we would expect since we "slowed" the spectrum down, reducing the frequency and increasing the peak to peak distance.

The given function works in both directions, but there is a problem when the frequency needs to increase ($f_d > f_0$). When reducing the frequency, there are more than enough samples in the original spectrum since we don't use the samples at the end (think about stretching a Slinky (tm) and then cutting it off in its stretched state to the original length). However, when increasing the frequency, we compress the spectrum so that the original number of samples doesn't fill up the original duration of the spectrum. How do we generate the remaining samples? We do this by making the original spectrum just long enough by copying a sufficiently sized portion of the middle and inserting it after the copied portion.

How big a portion do we need to copy? Simply

```
# of cycles to copy = (int) ((z - 1) * s / spc)
```

where z is defined as above, s is the total number of samples, and spc is the number of samples per cycle.

Input/Output

Your program should handle a few options:

pitch ;filename; report the YIN estimate

pitch -z ;filename; report the ZCR estimate

pitch -r ;filename; report the sample rate

pitch -w NNN ;filename; set the window size to NNN samples

pitch -c ;filename; correct the pitch of the sample to the nearest semitone - assume a frequency of 440 Hz for the superdominant and equal tempering - call the corrected sample out.wav

where ;filename; is a .wav file.

Grading

Points will be deducted for not adhering to the specifications given in this document and in the grading rubric. Points will be deducted for bad style, especially unreasonable amounts of duplicated code, as well as for sloppy formatting, insufficient or overly verbose documentation, compiler warnings, run-time crashes, and other such transgressions. You will receive no credit if your program does not compile.

You will be demonstrating your program to me and I will ask you to modify your program as you demonstrate it. If you cannot sufficiently explain your program to me, you will receive no credit and you may be subjected to a charge of plagiarism and be dismissed from class with a failing grade.

Submitting the assignment

To submit your assignment, delete the Java class files from your working directory. Then, while in your working directory, type the command

```
submit csce2143 lusth assign4
```

The submit program will bundle up all the files in your current directory and ship them to me. Thus it is very important to delete any extraneous files in your directory. This includes subdirectories as well since all the files in any subdirectories will also be shipped to me. I will deduct points for extraneous files, so be careful.

You must supply a makefile that responds to the command

```
make
```

which should compile your implementation. You may submit as many times as you want; new submissions replace old submissions. I should receive a makefile, your source files, and a shell-script named *pitch* to run your implementation. You may also submit any test cases you used.

You must submit from turing.csce.uark.edu!