Iuliu Teodor Radu

“One man barbershop quartet”
Realtime voice harmonisation
for the BeagleBone Black and Bela

Introduction

Voice harmonisation is the process of adding consonant notes to a voice, thus creating a homophonic texture. In other words, given one musical tone, others are added on top or below that sound ‘in harmony’, thus creating a chord. From a music theory point of view, there are horizontal (temporal) implications to finding ‘good’ chord progressions as well as vertical (harmonic). The real time voice harmoniser I have built attempts to lay the foundations for thinking out ways of intuitively navigating harmonies with just a voice and the help of a user friendly state machine. The motivation for this idea came from various videos of people overdubbing themselves in an attempt to create a one man barbershop quartet. In an attempt to make this system musically coherent, I have taken into consideration the diatonic implications of singing or playing along with a pitch shifter for up to four concurrent voices (including the input). Since major and minor keys comprise unequally spaced intervals, I have had to implement different ways of shifting between major and minor intervals. At a high level, the system has four different states, each exploring an added level of complexity for each added voice.

I have encountered a number of difficulties throughout the project, especially as far as the pitch shifter is concerned. I will attempt to document any imperfections in the system as I go along.

The pitch shifter

One of the biggest challenges of this project was to pitch shift a signal in a harmonically coherent way. Phase implications in the frequency domain as well as windowing of the Fast Fourier Transform had to be considered.

Since the input signal will be one that changes over time and is of indefinite length, it is not feasible to take the FFT of an entire signal. To keep the pitch shifting real time, the input signal has to be divided into manageable chunks or windows on which the Short Time Fourier Transform (STFT) is performed. I have applied a Hann function to the windows in order to overlap them. Once these windows get added back together, it is necessary to scale them down as their total amplitude will exceed that of the input amplitude. The scale factor can be seen at line 217 of render.cpp and the Hann window implementation can be seen at lines 256-8 of render.cpp.

The size of the sample being passed to the FFT also adds latency to the real time system. A larger FFT window allows for better quality pitch shift, yet adds more latency. Having experimented with this tradeoff, I have decided to set the FFT window length to 8192 samples, and to overlap adjacent windows by 75%. Therefore, the oversampling (osamp) variable is set to 4.

Once the FFT is run on a window using the ne10_fft_c2c_1d_float32_neon function, the frequency domain bins are stored within the frequencyDomain array. Although this array contains the same amount of variables as timeDomainIn, I am only going to use the first half of these, as they represent all frequency bins up to the Nyquist limit, which is half the sampling rate (44.1khz). In my current setup, each frequency bin is approximately 5.38hz wide, allowing for a good frequency resolution.

If every frequency we are trying to shift would be aligned perfectly with a frequency bin, then this would be an easy task. However, since I will try to pitch shift a voice, there will more than likely be lots of frequencies that lie between frequency bins. These frequencies will contribute to the adjacent frequency bins in a phenomenon known as smearing. In order to deal with frequencies 'smearing' between bins, it is important to consider the difference between FFT windows in terms of phase for each frequency bin. Each frequency bin holds information about the magnitude and phase at that frequency. In order to find a true frequency that is not aligned with a bin, it is necessary to find the difference in phase offset of a bin between the current and previous FFT windows.

The process of pitch shifting a signal in the frequency domain is sandwiched between an analysis stage and a synthesis stage.

In the analysis stage, the real and imaginary values of each frequency are converted into magnitude and phase by pythagorean rules:

Magnitude, $\sqrt{\mathrm{Im}^2+\mathrm{Re}^2}$ is implemented at line 301 of render.cpp.

The x2 multiplier at the beginning of the equation adds the magnitude present on the other side of the Nyquist limit.

Phase, $\arctan\left(\frac{\mathrm{Im}}{\mathrm{Re}}\right)$ is implemented at line 302 of render.cpp.

A gLastPhase global array is used to keep track of the phases from the previous FFT window. This is necessary in order to find out the phase difference between the current and past window. The resulting difference has to be compared against the expected difference between two windows. The difference between the expected and actual differences can often result in a phase outside the +/- pi interval. If this is the case, multiples of 2pi have to be subtracted (if result is positive) or added (if result is negative) until the angle lands into the +/- pi or 0-2pi interval. This is called phase unwrapping. Finally, getting the frequency deviation from the nth bin is gotten by dividing the above difference by 2pi, multiplying it by the oversampling rate (considering we need to let the phase run for an entire window, yet the difference collected was between windows 1/4 of a window apart) and finally multiplying it by the nth bin's order. The resulting adjustment may be negative or positive, thus it is either added to or subtracted from the nth bin's frequency to calculate the 'true frequency'. Lines 317-21 of render.cpp reflect an implementation of the above. The tmp variable holds the above long winded string of differences until the final computation is stored in a gAnaFreq array for later use.
The pitch shifting stage involves a pitch_shift_index variable that polices the shifted frequencies to not exceed the upper bound of the frequency spectrum allowed in the FFT window. In other words, frequencies which end up being higher than 22050hz get pushed

out, while maintaining target shifted frequencies within the 0-22050hz limit. Shifted analysis frequencies get inserted into a synthesis array, gSynFreq which now contains the corresponding new true frequencies. Implementation details can be found between lines 335 and 351 of render.cpp.

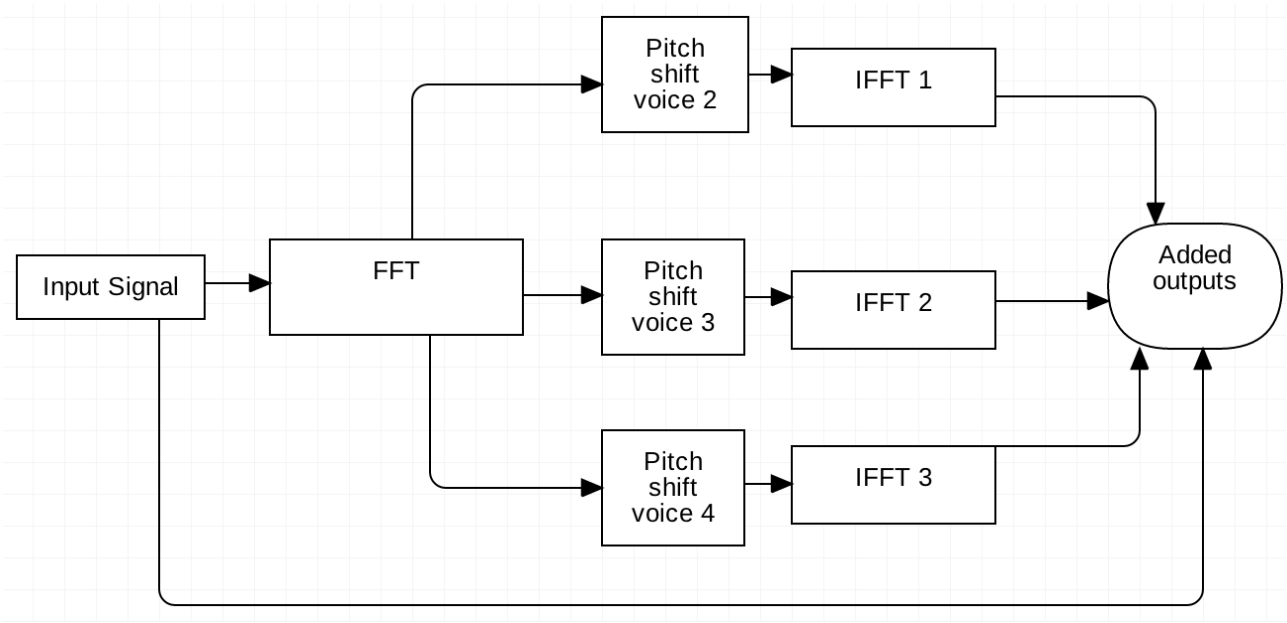
The final synthesis phase is essentially the analysis phase in reverse.

To get the real and imaginary parts back into the time domain, it is necessary to convert the phase and magnitude information by the following formulas:

$$\mathrm{Im} = Magnitude \times \cos(\phi)$$
$$\mathrm{Re} = Magnitude \times \sin(\phi)$$

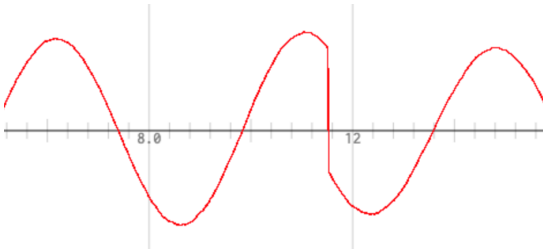
It seems that performing these calculations inline and storing them back into the frequencyDomain array is computationally expensive. When trying to do three concurrent calculations, the Bela was stressed to a CPU usage of greater than 90% and froze, causing a lot of dropped packets as a result of buffer underrun. However, this was easily mitigated by calculating and storing the cos and sin of the phase separately and then multiplying the result by the magnitude. Perhaps it is difficult for the Bela to cast a float into a ne10_fft_cpx_float32_t.

As seen in the code between lines 335 and 351 of render.cpp, I have introduced three pitch shifts in total. This is because I would like to have up to three concurrent voices playing simultaneously in my harmonising system. Therefore, for every FFT window, there will be one FFT and three inverse FFTs, outputting three separate pitch shifted signals into output buffers. The following diagram will clarify this system.



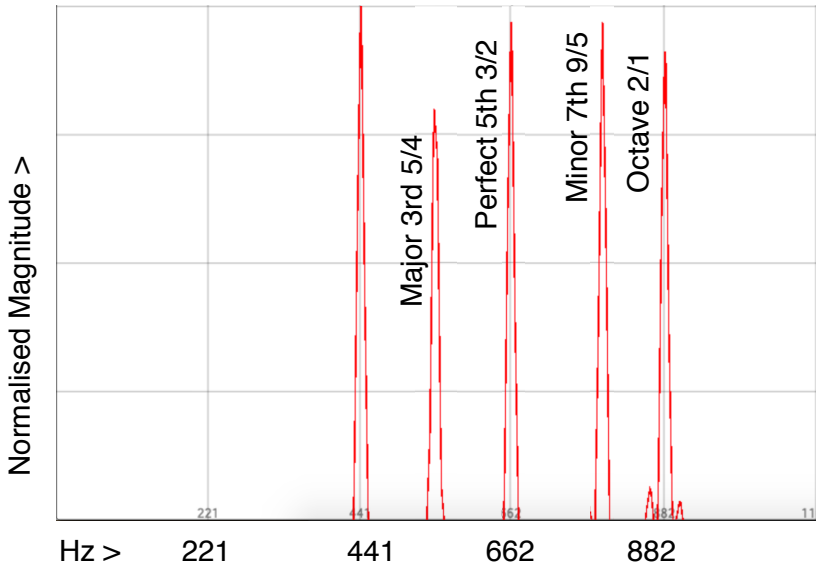
Evaluation of Pitch Shifter

Having tested the pitch shifter after implementation, I have observed that certain frequency sine waves wrap about three quarters of a cycle. To start with, when just passing sound through the three stages above with a pitch shift coefficient of 1, there is a sort of din created in the output which seems to be proportional to the FFT window size. When pitch shifting a constant sine wave of 310hz up by a major third, there is a recurrent click in the sound that seems to be proportional to the FFT window length. I am convinced that something is wrong in the phase unwrapping or wrapping stages. The phenomenon is caused by the phase of the sine wave wrapping about three quarters of a cycle as shown below in an intermittent fashion. In addition, this comes with a slight, periodic scaling down of the magnitude of the wave. This only happens with certain frequencies. As this is much less noticeable with an irregular input signal, I have decided to keep implementing the pitch harmoniser regardless of these imperfections.



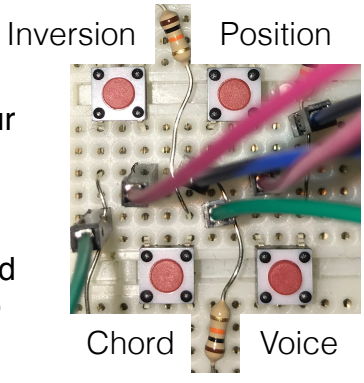
To prove that the pitch shifter does indeed work as anticipated, I have put a 441hz sine wave through the input and measured the output after the FFT with the Bela scope. As expected, with a pitch shift coefficient of 1, the frequency appeared correctly.

As can be seen in this figure on the right, the intervals have been correctly shifted to the expected frequencies. The 441hz reference tone has been shifted to 551.25hz for the major 3rd, 661.5hz for the perfect 5th, 793.8hz for the minor 7th, and 882hz for the octave.



Usage and logic

The harmoniser is essentially a state machine with different features for each of three concurrent voice modes. There are four buttons on the breadboard, labelled ‘Voice’, ‘Chord’, ‘Inversion’, and ‘Position’. Each of them facilitates the passing between four different states or modes. The desired state of each of the functionalities can be selected by clicking a button between 1 and 4 number of times in quick succession. For example, clicking the



‘Voice’ button 3 times in quick succession sets the ‘Voice’ mode to 3. There is a 1 second timer that allows the user to navigate to the required mode before it takes effect. Once the new state comes into effect, there is a message on the console that notifies the user of the new state selected for the relevant button clicked. In addition to the buttons, there is a potentiometer that works when the 2nd voice state is selected. I will discuss more about functionalities and states in subsequent sections.

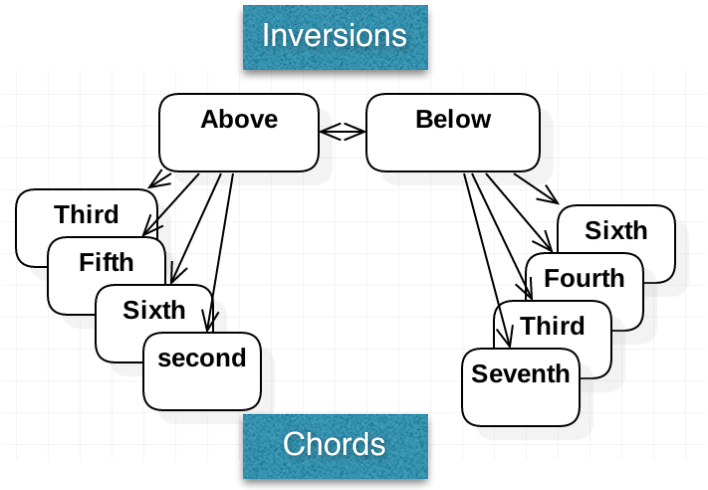
A quick and relevant music theory primer

A scale is a sequence of seven notes in succession. A diatonic scale tells us about the intervals or ‘space’ between these notes. Namely, a diatonic scale includes five tones (full steps) and two semitones (half steps). Focusing specifically on the major diatonic scale, the intervals that make it up are as follows: step (Do-Re), step (Re-Mi), half step (Mi-Fa), step (Fa-Soh), step (Soh-La), step (La-Ti), half step (Ti-Do). The complication this presents when harmonising a voice is not immediately obvious. If we were to pitch shift an accompaniment a major third above, the singer would experience an unpleasant dissonance should he/she ascend a diatonic major scale. For example, if we were in the key of C major and a singer would start ascending the major scale, the correct accompaniment ‘a third above’ should be C-E, D-F, E-G, F-A, G-B, A-C, B-D, C-E (The lower voice/singer’s voice being the left hand side of each pair, and the accompaniment/ computer pitch shifted voice being the right hand side of each pair). However, the result of having a rigid pitch shifter in major thirds would be C-E, D-F#, E-G#, F-A, G-B, A-C#, B-D#, C-E. The sharps observed would give this ascending scale an unpleasant, jabbing, dissonant sound. To correct for the ‘sharp’ intervals, I have used the potentiometer mentioned above to ‘tighten’ or ‘loosen’ the musical interval. For example, it gives the user the option of shifting a major third into a minor third (tighten) or into a perfect fourth (loosen) in real time.

When more than one accompaniment is played in real time, the ‘diatonic shifter’ does not work anymore as the chords become too complex. Hence, thinking about pleasant harmonic progressions comes in handy. Good harmonic progressions involve juxtaposing certain chords in such a way that they sound good in succession. In addition to the choice of chord, the inversion of that chord is also important. Every basic chord is composed of the root, third, and fifth (intervals) of a diatonic scale. The root provides the essential ‘grounding’ for the chord, while the third provides the ‘flavour’ of the chord, whether it sounds happy or sad, and the fifth provides a sense of space and ‘openness’ of a chord. Each chord’s character depends on the configuration of each of its components. If the chord is voiced with the root as the bass, this generally gives it a grounded and stable character; if the third is voiced as the bass, it gives the chord a sense that the progression is not yet finished; while if the fifth is voiced as the bass, it gives the chord an unstable feeling. These choices of voicing are called inversions i.e. ‘toppling’ the chord back over itself to create a different character of sound out of its different configurations.

Keeping the above oversimplified definitions in mind, I will explain the functionality of each state of my harmoniser. Because the system only has four buttons, certain harmonic constraints had to be set, and a lot of practice is needed to perform well with the system.

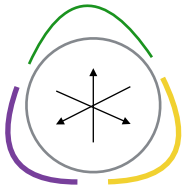
Functionality of 2 voice mode



In order to select 2 voice mode, the voice button has to be pressed twice in quick succession. It is then possible to select whether the accompaniment to the input should be above or below by using the inversion button. Then the desired chord should be selected. Changing the inversion while singing a certain interval will select the inverted interval in the opposite direction. For example, if the accompaniment is set to a third above and the inversion is changed, the interval will change to a sixth below. The 'Above' and 'Below' states indicated above are controlled by a variable called 'inversion_mode' which is toggled by an 'inversion_trigger'. Each of the interval states is controlled by setting the pitch shifting variable 'pitchShift2' to one of four values held in the 'twoVoiceIntervals' array based on which 'chord_mode' is selected.

In addition to the interval functionality, as mentioned above, the potentiometer allows to tweak the pitch shifted voice by a semitone above or below. For example, if one is singing with the accompaniment in perfect fifths above, the potentiometer allows to shift the interval to a minor sixth, or an augmented fourth. In the case of thirds, this is a useful tool for staying in tune when moving around in major or minor keys.

The potentiometer is set to detect three states as shown on the right. It makes it easy to handle during performance since setting its general position, the correct state takes effect and the 'diatonic offset' is changed. The figure on the right highlights the regions of the potentiometer, with the green region indicating no diatonic offset.

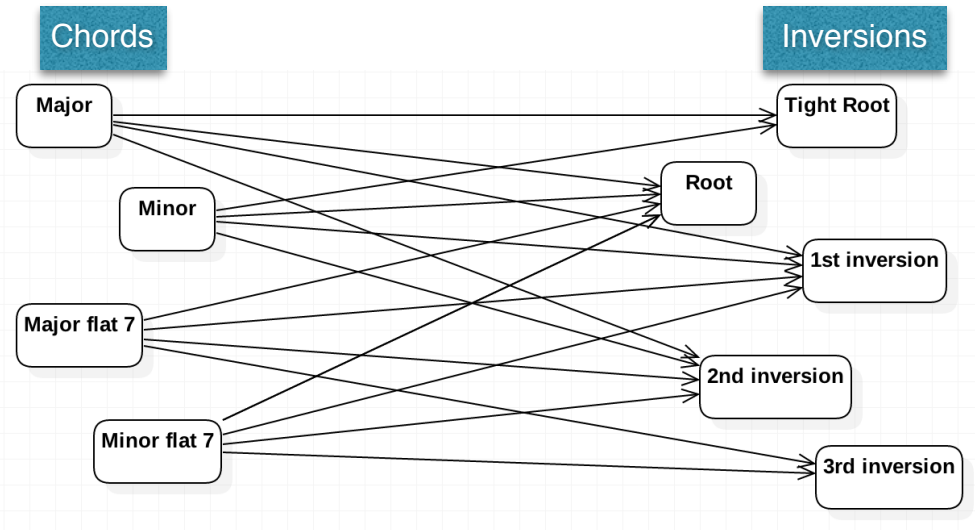


The 2 voice mode is a simple harmoniser based on the basic building blocks of harmony: intervals. To facilitate expressively coding musical intervals, I have created an 'Intervals' class that allows to refer to the intervals directly rather than to the arithmetic harmonic ratios. For example I can therefore refer to a Perfect 5th above the input as Intervals::per5th_u instead of an abstract '3/2'. This allowed me to quickly think harmonically rather than mathematically about the musical design. All of the variables in the Intervals class are static, and therefore the class does not require instantiation before use. In addition to the interval variables, I have provided two arrays for housing these intervals, one for the intervals above and another for the intervals below. Since all intervals are in relation to where the voice input is, these two arrays would provide the arithmetical means by which to invert the intervals when the user changes the interval state. The 'up' array includes the higher intervals in order from smallest to largest, while the 'down' array includes the lower intervals in order from largest to smallest. Hence, it is possible to flip from a major second above to a minor seventh below etc.

Functionality of 3 voice mode

In a similar fashion as above, the 3 voice mode is selected by triple clicking the voice button in quick succession and waiting for the state to take effect. This time, the pitch shifter adds two voices. Because of the added complexity of navigating harmony in three voices, I had to do away with the idea of just thinking of intervals, and introduced chords with inversions instead. This also sees the potentiometer deactivated from this point onwards.

The Inversions class serves the purpose of implementing three-voice chords that sound good. Given I have four states for the chord button, I chose to provide four fundamental chord types for use. In a diatonic sense, the most useful chords are major, minor, major flat seventh, and minor flat seventh chords. The minor flat seventh chords are least used in classical harmony, but became more popular during the 20th century, especially as far as the advent of jazz and pop subculture is concerned. Considering the chords in terms of their most frequent usage, I have assigned major chords to chord state 1, minor chords to chord state 2, major flat seventh chords to chord state 3, and minor flat seventh chords to chord state 4. These are selected in the standard way using the chord button. Since the Intervals class provides intervals of up to an octave above and down to an octave below, the range of every possible chord in this 3rd voice mode will be of two octaves. I have made some detailed decisions as to the inner voicing of each chord using these intervals that I have documented in the implementation code. The singer or input always represents the middle voice of these chords, and will always have an interval sounding below and above itself. Therefore, it takes a good ear and practice to position oneself in each of these chords and to be able to navigate between them. It might be easier to play an instrument into the harmoniser rather than to sing into it at first.



Within each of the four chord states, I have designed four inversions. Therefore, different ‘flavours’ of chords can be obtained for more sophisticated progressions and harmonisation. The inversions can be selected by using the inversion button in the standard way. Once in a chord state, it is possible to flip the chord so that the third or fifth is on the bottom, given the input is moved to the appropriate part of the chord indicated in the code. Chords are in root position by default. As an example, in the minor flat seventh chord state, it is possible to put the chord in second inversion, i.e. fifth in the bass, by clicking the inversion button three times, and singing a major second higher than before. Singing a major second higher is necessary due to the fact that initially, the input represented the flat seventh of that chord, while after

the change, the input represents the root of that chord. Further details are included as documentation in the code.

The major and minor chords can have a root position, 1st inversion, and 2nd inversion. However, I also defined a ‘Tight Root’ inversion for the major and minor chords as these cannot have a 3rd inversion. This is a compact triad in root position in comparison to the ‘Root’ chord which is spread over two octaves. Since we only have three total voices including the input to play with, the flat seventh chords do not have a ‘Tight Root’. The ‘Tight Root’ feature might be a good place to start when getting to grips with the 3 voice mode as the upper and lower parts are very close to the input and easier to hear the intervals clearly.

Functionality of 4 voice mode

This is where it gets really difficult to keep singing in tune. In order to implement four voice chords, I had to make decisions for each individual 3 part chord whether the extra voice should go on top or below of the input. As the singer, one would need to be very aware whether they are singing the tenor or the alto parts of each of those chords. The voicing for each chord is documented in the Inversions class implementation. In addition, for the simpler major and minor chords, since there are only three parts, I had to decide which one to double as the fourth voice. I had to take into consideration spacing between the intervals and the harmonic function of each of the chord inversions. For example, a first inversion chord with the third on the bottom would sound quite weak if it had the third doubled. Indeed, these chords are used rarely and for special effect. Hence, I decided to double the fifth for first inversion chords, which makes them sound more ‘open’.

Unfortunately for the singer or any poor soul using this four voice harmoniser, changing inversion states can now have you jumping between tenor and alto parts of the chord voicing in addition to changing the interval you represent in the chord. For example, if one changes from a root position major chord to a first inversion major chord, they have to go between singing the fifth of the chord as a tenor to singing the root of the chord as an alto. The documentation shows these combinations more clearly. Fortunately for the user who may not be happy ending up singing the alto part, I have implemented position shifters that allow to ‘swap places’ with the other middle part. These swap_to_alto and swap_to_tenor static functions implemented in the Intervals class calculate the new pitch shift of the surrounding voices from where one would be if they were to swap with the other middle part. For instance, if a chord were composed of a root, third, fifth, and root while I were singing the tenor (third), then calling the swap_to_alto function would allow me to instantly sing the fifth as the alto, while the other parts would remain sounding the same.

Given that one knows their position in a chord and the interval for the target other middle part, the position button is used to fire the relevant function for swapping voices between middle parts. It has one of two states since this allows for the singer to be either middle part in all of the chords and inversions available.

Control flow

There are four functions inside each audio frame inside render() being called to control the flow of the program: a checkButtons(), set_diatonic_offset(), selectListener(), and voiceLogic() function. The less obvious selectListener() handles the logic of selecting states while prompting the user what is clicked and what is selected. The voiceLogic() function acts as a router that passes control to the appropriate twoVoiceLogic(), threeVoiceLogic(), or fourVoiceLogic() functions. Within each of these voice logic functions, support for navigating between states is provided by extensive usage of switch statements and keeping track of global variables. Appendix 1 is a more detailed attempt at documenting the system.

As all states of this system are impossible to represent succinctly, this diagram is a compromise that shows passage between states by means of 'control interactions' which are represented by the red bubbles. Each control interaction is associated with a button or potentiometer listener that are described above. These red bubbles also describe the switching possibilities between states, hence the double sided arrows. This is instead of showing messy interactions between states directly, as each state could switch to any other state.

The green bubbles represent the voice states. In order to interpret the correct control flow for one of the voice states, it is important to assume that only the signal processed as part of the selected voice state is outputted. Therefore, logically, only the arrow paths from one green bubble should be followed to the output at a time for a correct description of the control flow for each state in the system.

The yellow bubbles represent the logical representation of how the pitch shifting will be performed at a high level i.e. as intervals above or below (as is the case for two voice pitch shifting), or as three or four part chords.

The blue bubbles represent sets of either UP intervals or DOWN intervals, which are the low level implementation of the pitch shifting.

The purple bubbles represent the pitch shift variables that are manipulated based on the rules preceding them.

Conclusion

Albeit a bit complex to wrap one's mind around the functionality of the harmoniser, it could be used as a practice tool for ear training, trying to navigate between the different voices. If used as a composition tool, it would be possible to get someone to sing a line without thinking of the complexity of the harmoniser, while automating its functions at different time intervals. The system could also be used to try and explore nice harmonic progressions intuitively and see if it is possible to improvise along with it. There are currently obvious user interface limitations to the system which would prevent it from becoming a live performance device. One way of improving the experience of the system is to write up a big chart with all the possible combinations of chords and inversions and which part the input is supposed to play at any given time. An attempt to map out voices in musical notation can be seen in Appendix 2. The red highlighted notes represent the parts of the chord to be sung/played into the input in order to achieve the chords depicted. Another improvement would be to make the selection of states much quicker. The one second time interval that one has to wait between pressing a button and switching states is inconveniently long if trying to sing faster than 60bpm. In order to make it a 'true' one man barbershop system, a pitch recognition algorithm and further intelligent automation has to be implemented which may be beyond the scope of the Bela's processing power.

References

Allain, Alex, *Bitwise Operators in C and C++: A Turorial,* https://www.cprogramming.com/tutorial/bitwise_operators.html, accessed 09/03/18

Bernsee, Stephan, *Pitch Shifting Using The Fourier Transform*, http://blogs.zynaptiq.com/bernsee/pitch-shifting-using-the-ft/, accessed 12/04/18

Dattorro, John, *Phase Response,* https://ccrma.stanford.edu/~dattorro/phaserules.pdf, accessed 12/04/18

Mathworks*, Unwrap,* https://uk.mathworks.com/help/dsp/ref/unwrap.html, accessed 13/04/18

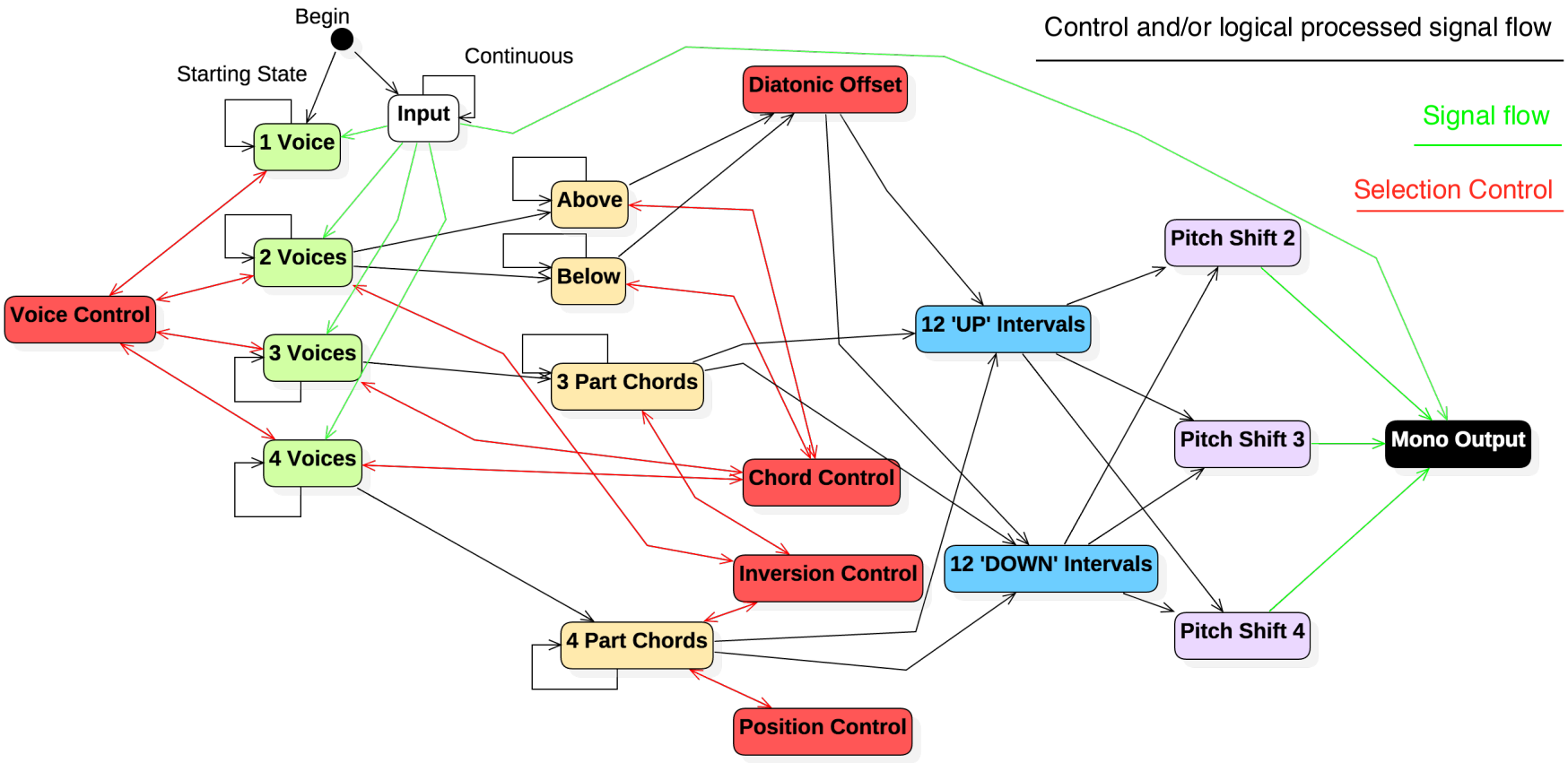
National Instruments, *Understanding FFTs and Windowing,* http://download.ni.com/evaluation/pxi/Understanding%20FFTs%20and%20Windowing.pdf, accessed 13/04/18

Parviainen, Olli, *Time and pitch scaling in audio processing,* https://www.surina.net/article/time-and-pitch-scaling.html, accessed 12/04/18

StarUML, *Working with Statechart Diagram*, http://docs.staruml.io/en/latest/modeling-with-uml/working-with-statechart-diagram.html#uml-state, accessed 09/03/18

Stewart, Rebecca (2018), Lecture notes

Appendix 1 - Diagram of harmoniser control flow



Iuliu Teodor Radu

Appendix 2 - Chord chart depicting voice location - example in the key of C.