MATLAB Speech Synthesizer

Final Report

Lachlan Dow

2021

**Introduction**

This paper will describe the design, development and testing of a formant speech synthesizer built in MATLAB. This process took place between the 28th of October 2020 and the 27th of April 2021. Using MATLAB, the project has implemented a pre-processing unit and a speech synthesiser to take in user-inputted text and output the text as speech. The pre-processor implements rules to convert the text into the phonetic equivalents of the speech that is then inputted into the synthesiser which will produce a sound signal that represents said speech.

**Base Overview of the System –**

The MATLAB formant synthesiser (SpeakyThing9000) has been mainly developed based off the MITalk system, but it also draws from other synthesisers and techniques. The interface will take in text input and output the speech version of it. The system does this by first processing the inputted text through a series of functions into a form that is readable by the speech synthesiser. These stages include Text Reformatting, Text-to-Phoneme conversion, Text-to-Sound conversion, and sound duration calculations. This part of the system outputs a series of data frames that represent the settings for each 5ms of output for the speech synthesiser. This series of frames is inputted into the synthesiser, which will then use these frames to create an output array of the values of the sound that these data frames represent. Finally, the sound is played for the user.

**Background and History of Speech Synthesis Systems -**

For hundreds of years, people have been trying to mechanically produce human speech. Early models created in 1779 by Christian Kratzenstein (History and Development of Speech Synthesis, 2021) generated vowels by constructing models that resembled the human vocal tract (as shown in figure 1) by vibrating reeds like musical instruments.

Diagram

Description automatically generated

Figure 1. Katzenstein’s resonators (History and Development of Speech Synthesis, 2021)

In the subsequent century and up until the 1960s, multiple models for mechanical and semi-electrical analogues of vocal systems were created. The most notable of these systems is known as the VODER (Voice Operating Demonstrator), which was developed by Bell Laboratories in the mid-thirties (History and Development of Speech Synthesis, 2021). The device was designed to analyse speech into slowly varying acoustic parameters that could then drive a synthesizer to approximately reconstruct the original speech signal.

Diagram, schematic

Description automatically generated

Figure 2. The VODER speech synthesiser (Klatt, 1987)

The invention and demonstration of this system caused the scientific community to become increasingly interested in speech synthesis. In the coming decades, systems like the Pattern Playback synthesizer constructed by Franklin Cooper (Klatt 1987) reconstructed recorded spectrogram patterns into sounds. The first formant synthesizer, PAT (Parametric Artificial Talker), was constructed by Walter Lawrence in 1952. This system consisted of three electronic formant resonators connected in parallel. Moving glass slides where then used to convert painted patterns into six-time functions to control different parameters for the synthesiser. Around the same time, a synthesiser that processed sound using cascade resonators was introduced called OVE I (Orator Verbis Electris). This system would go on to be redesigned and improved over the next two decades in the OVE II and OVE III.

These two systems were combined by John Holmes in 1972 when he introduced his own parallel formant synthesiser. He was able to tune the synthesiser to say “I enjoy the simple life” well enough that most people couldn’t tell between the synthesiser and a real voice.

In 1979, the MITalk system was developed by Allen, Hunnicutt and Klatt. This system used all the advancements up to this point including the parallel/cascade synthesis method to produce extremely realistic speech sounds. This system would later go on to be adapted to be used in DECtalk – a commercial text-to-speech device.

Text-to-speech synthesis has continued to advance to the stage it is at today, with complex unit synthesis systems that use POLSA technology to blend pre-recorded text together in an extremely realistic way. These systems are getting to the stage where it can sometimes be difficult to differentiate between them and a real human voice.

**Design –**

The design process of the SpeakyThing9000 synthesiser began with investigation into speech synthesizers, their history, and current state of the art synthesizers. This stage uncovered multiple different types of synthesis, namely:

* Unit selections synthesis - this uses a large database of recorded speech where each recorded utterance is segmented in different ways using a specially designed speech recognition system. These segments are indexed based off defining factors e.g., fundamental frequency, duration etc.
* Diphone synthesis - this uses a database containing all the diphones occurring in a language. Only one example of each diphone is contained in the speech database, then at runtime the prosody of a sentence is superimposed on the units, usually through a linear predictive coder (LPC).
* Domain-specific synthesis- this method concatenates pre-recorded words and phrases to create complete utterances, this method is generally only used in speech variance as it is very limited, for example a train announcement system or weather reports.
* Formant synthesis- this method uses an electronic speech production model where different parameters are varied to create speech sounds, these values can then be interpolated to create diphthongs or stopped to create plosive bursts. This method is said to create a sound that is very easily understood but sounds very robotic.

Until recently, this was the best option for synthesis due to its customisability and adaptability.

In the initial stages of design, creation of a unit selection synthesis system was investigated as a strong possible solution. In the experimentation stage of this, MATLAB was used to take speech segments and separate them into their respective sounds, and then reassemble them into the sounds that were needed to create other words. This was found to not be very effective for a.), the scope of the project and b.), MATLABs features.   
After investigation into other options and reading speech synthesis and recognition(J & W Holmes. 2002) and MITalk(Jonathan Allen, M. Sharon Hunnicutt, Dennis H. Klatt, Robert C. Armstrong, and David B. Pisoni. 1987.), these sources had more information on the use of formant synthesis as a speech production method, and coupled with advice from the project advisor, a decision was made to transfer the project to a formant synthesis system.

The design for the MATLAB version was split into two parts which were built one after the other – first: the formant synthesizer, based off the parallel/cascade design (Klatt. D, 1980) and second: the text pre-processor for input into the synthesizer.

**Diagram

Description automatically generated**

Figure .Class diagram for cascade/parallel synthesiser based off Klatt synthesiser (Klatt. D.1980)

**The Formant Synthesizer – Design and Development**

The design of the formant synthesizer was adapted from the design found in the paper by Klatt himself and also from other adapted versions that are more suited for modern systems, most notably the system developed by d'Heureuse (2019). The synthesizer was broken down into different sections as objects in MATLAB, and each object would have step functions to replicate a signal passing through them and the effect that it would have on that signal.

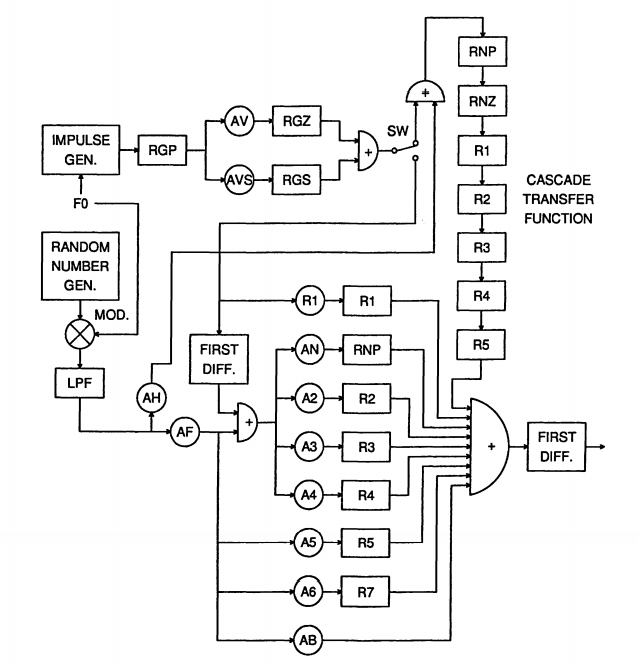


Figure 4. Block diagram of Klatt cascade/parallel formant synthesizer

Pictured above is the class diagram for the synthesizer in MATLAB. This has been built as an object-oriented application. The main function for this system is the generateSound() function. It is responsible for creating the sound waveforms from the parameters inputted. The function takes a data frame as input as well as the main parameters for the synthesizer, and then will process the values for the sound through each branch to output an array of values that can be used to produce a sound.

**More details on the objects -**

GeneralParameters - this class represents the main parameters for the synthesiser that will not change during the sound production process.

FrameParms - this class represents the values of the data frames that will be used by the synthesiser to generate the speech waveform. The synthesiser will pass a new set of frame parameters for every 5ms of sound produced.

CurrentFrameParameters – this class represents the current parameters being used by the synthesiser in their linear form.

PeriodState – this represents the variables currently active at a given fundamental frequency period.

Diagram

Description automatically generated

Figure 5. digital resonator increasing output value from frequency and bandwidth domain

Resonator – This class represents the resonator from the Klatt formant synthesiser (Klatt, D., 1980). The digital resonator will use a frequency and bandwidth to increase the ratio of output for a given input signal at the mentioned frequency and bandwidth. This is done with a calculation.

Text

Description automatically generated

Figure 6. Filter coefficients

In the Figure 3 shown a, b and c represent the filter coefficients for the calculation of the next output value from the resonator, r is used as a temporary step (-) for the c and b coefficient calculations. y1 and y2 represent the last and second last output value of the resonator. The object contains functions to either set the resonator as a passthrough object, set it to a muted state, adjust the gain of the signal or to calculate the next output value. The step function performs the calculation to calculate the next output value:



Figure 7. Digital resonator calculation

Where A, B and C represent the filter coefficients, represents the number passed into the step function, is the previous output value of the resonator and is the second to previous output of the resonator. The function will then update the previous and second-to-previous values and output the newly calculated value.

The Anti-Resonator – This class represents the anti-resonator or anti-formant (Klatt,D. 1980). This class is designed to do the opposite of the resonator. A slight modification to the resonator equation is done to invoke the correct response.



Figure 8. Digital Antiresonator calculation (Jonathan Allen, M. Sharon Hunnicutt, Dennis H. Klatt, Robert C. Armstrong, and David B. Pisoni. 1987)

Where represents the current sample, represents the last output sample and represents the second-to-last output sample. A` B` and C` are defined by the calculations in Figure 8, where A, B and C are represented by the calculations in Figure 5.

Text

Description automatically generated

Figure 9. Anti-resonator filter coefficients calculation (Jonathan Allen, M. Sharon Hunnicutt, Dennis H. Klatt, Robert C. Armstrong, and David B. Pisoni. 1987)

**The Differencing Filter –**

This represents a high pass filter that is used to remove low frequency energy from the higher frequency formants.

LpFilter1 – This represents a first order infinite input response filter which will take an input value and produce an output value using 2 filter coefficients, a and b, where

Where G is the gain set of the filter, f is the frequency of the filter, T is the sample rate of the synthesiser, and EG is the additional gain added to the filter – set as standard to 1.

The filter calculates the next step in the calculation with the equation.

Where represents the current input value and represents the previous input value. a and b represent the filter coefficients mentioned above.

This filter is used to generate the glottal source pulse for the system, and the tilt filter to implement a spectral roll-off slope.

**The Low Pass Noise Source –**

This class implements a glottal source generator that is passed through a low pass filter. The class stores a low pass filter and will initialise it with the necessary parameters to filter a signal with a 1000Hz cut off and a gain value of 0.75. When the getNext() function is called, the object uses the getWhiteNoise() function which will create a random value between -1 and 1, then input this as the number in the step function in the low pass filter. The output value will then be returned.

**The Impulsive Glottal Source –**

This object is used as a generator of a glottal pulse filtered through a resonator configured as a low pass filter. The resonator is configured based off the frequency target of 0, as that allows it to function as a low pass filter only allowing values within the bandwidth specified. The sample rate of the synthesiser is divided by the open phase length to determine the bandwidth necessary for a low pass filter of this noise generation object. The step function can be used to compute the next value of the pulse signal by passing a value, either 1, -1 or 0, based off the position in the period of the impulse into the step function of the resonator.

**The Natural Glottal Source –**

This class generates a glottal pulse based off the KLGLOTT88 model (D.H. & L.C. Klatt (1990). This generator calculates the pulse for the synthesiser using the current (a) and second (b) derivative of the signal. The calculations for this are as follows:

Using these values, the current output for the glottal pulse is found by first adding *a* onto *b* then adding *a* onto the output value and returning the output value.

**The Generator Class –**

This is the main class in the synthesiser part of the system. This class combines all the previous classes into a series of functioning components. The generator object will initialise instances of classes that represent the parts of the synthesiser pictured in Klatt’s original diagram.

Something that is important is the outputLpFilter, which is an instance of a low pass filter that is used to filter the output signal from the synthesiser. The aspiration sources for the cascade and parallel branch are also of note – these are instances of low pass noise sources to create an aspiration sound in the synthesiser. Another part that is essential is the nasal formant resonators and anti-resonator that are responsible for the creation of nasalised sounds. The oral formant variables for the cascade and parallel branches are both arrays of resonators used to amplify the formant sections of the signal. These resonators are passed signals in different ways for both the cascade and parallel sections of the system.

When a generator object is created, it initialises all the variables that are needed in the constructor. After initialising these variables and objects that make up the class, the generateFrame() function can be called. The function will start a new output period for a given set of frame parameters, then for the given length needed of the output signal, it will loop to produce an array of output values by triggering the function computeNextOutputSignalSample().

The function design of this code and the functions featured in the generator object were designed using activity diagrams.

Diagram

Description automatically generated

Figure 10. Activity Diagram for computing the next output signal

The generate glottal pulse section of the function is created by getting the next output value from the chosen method of glottal source generation. Both the cascade and parallel branch calculations can be broken down further.

Diagram

Description automatically generated

Figure 11. compute cascade branch activity diagram

The diagram above lays out the procedure for calculating the voiced value for the cascade branch. The aspect that differentiates itself most from the parallel calculation is the way it calculates the voicing value. After each formant or nasalisation is performed, the voiced value is passed into the next resonator for the formant.

Diagram

Description automatically generated

Figure 12. Compute parallel branch activity diagram

The parallel branch calculation, as featured in its respective activity diagram, features calculations for the aspiration value of the branch but also the frication value. This branch is heavily used to create the phonemes that require frication, such as RR or FF and the plosive bursts as the parallel branch does not produce the same jumps and burps if you change the output values quickly.

The values produced from these functions are then returned to their respective values in the computeNextOutputSignalSample() function. These values are then summed and passed through the output low pass filter. The compute next output signal returns these values to the generateFrame() function, which will add this value to the output array, then loop until the end of said output array. The final array will be returned to the program to be outputted as sound.

**Implementation difficulties –**

When constructing the synthesizer, a waterfall method of production was used, as generally, to have a system that produced any sound at all, the whole base structure of the synthesizer would be needed. Because off this method, after laying out the base structure of the synthesizer, roughly 4 weeks was spent debugging; stepping through the program and ensuring variables were correct at each stage, as well as fixing errors that were caused by putting all the different classes together. In the future, it would be good to construct a predefined testing plan for each function and class to ensure that they would work correctly with the synthesizer. This would positively impact the production of the generator class, as that was where a large portion of the errors were found.

Generating white noise - this problem was discovered as the random number generation in MATLAB has a very wide spread of differences that tend to gravitate towards the more extreme ends of the scale that has been given. This caused a lot of confusion in the initial development of vowel sounds. Multiple options were tried including another random number generation method, but in the end, simply reducing the variance in the numbers was enough to make the sound understandable.

Amplification in the low pass noise source – the low pass noise source had a very high amplification level, and because of this, there was a large amount of buzzing produced. To remedy this, the amplification level was adjusted to the stage that there was less buzzing to the extent where the output signal was recognisable as a formant.

Nasalisation – when attempting to create nasalisation in the synthesizer, a buzzing sound was produced that made the important section of the sound inaudible. This was fixed by adjusting the nasal anti-formant resonator to remove a larger section of the non-nasal sound. This fix came with the effect of the sound’s quality seeming slightly hollow, but it increased legibility to a large degree.

**Diagram

Description automatically generated**

Figure 13. Flow of the pre-processing unit

**Pre-Processing unit**

The pre-processing unit was developed using an agile approach. Each component was developed and functional in a deployable way. This section of the system was broken down as:

1. Text reformatting
2. Text-to-phoneme conversion
3. Phoneme-to-sound frame conversion
4. Sound duration computation

More detail on these sections is provided below.

**Text Reformatting**

In this section, numbers were broken down into their word equivalent with a series of functions, acronyms were translated into individual letter segments to get the right sound from the system, and some abbreviations were translated into their full words.

**Pseudocode for Text Reformatting:**

1. findAbbreviations()
2. split array into words
3. loop for each word
   1. use regular expression to find acronyms
   2. regular expression to check for numbers
   3. if the word is a acronym
      1. replace the acronym in the array
   4. end if
   5. if the word is a number
      1. replace the number with the word equivalent
   6. end if
4. end for loop
5. change all words into the lowercase versions for the next stage of pre-processing

**Text-to-Phoneme Conversion**

This function’s purpose was to break down a word into its different phoneme segments. In an overarching method, the function word2Phone will take a word input, a map of key-value pairs for the different letters, and their position in an array of letter-to-phoneme rules. The function then loops through every letter of the word, referencing the word conversion with the rule list, and creates a new array with the phoneme conversion.

**Pseudocode for Text-to-Phonene conversion**

1. while loop for every letter
   1. get index position of rule in array
   2. loop until match found
      1. extract the character matches for rule segment
      2. check if the segment matches the section of text from current letter
      3. if this section matches extract the conditional rules for match before letter rules
      4. check that the string preceding the rule set matches the extracted rules
      5. extract conditional rules after the matched letter set
      6. check the string after the rule set matches the extracted rules
      7. assign Boolean value to matchFound
      8. if match is found push phonemes equivalent from the rule to array of phonemes
      9. increase rule index value by one and return to start of array
   3. end loop
2. end while

**Phoneme-to-Sound Frame Conversion**

This function will convert the phonemes into their sound frame equivalents. The function will loop through the array of phonemes and then convert them, using a map of the phoneme symbol, to an index representing the sound frame for that phoneme.

**Pseudocode for Phoneme-to-Sound conversion**

1. loop for the length of the array of phonemes
2. if sound doesn’t contain diphthong
   1. if sound is a \_ for HH sound
      1. get next phoneme and convert to HH equivalent
      2. add sound to sound array
      3. move onto phoneme after adjusted one
   2. end else
      1. convert to sound
      2. move onto next phoneme
   3. end if
3. if sound does contain a diphthong
   1. split diphthong into segments
   2. convert phoneme into sound frames
   3. assign diphthong attribute to these sounds
   4. more onto next sound in the array
   5. end if
4. end loop

**Sound Duration Computation**

This function is used to calculate the duration of sounds and how to blend them. The parameters inputs for this function are a list of sound objects to be converted. The function will then process these sounds into 5ms sound frames from durations based off a map of phonemes and duration characteristics, then output these sounds as an array to be passed into the generateSound() function in the Generator class.

**Pseudocode for Sound Duration Computation**

1. check if array length is one
2. if the array length is one
   1. assume that the item is a letter and adjust the interpolation accordingly
   2. function return
3. end if
4. loop for length of sound array
5. if sound is last in array or plosive
   1. generate sound frames with no blending
   2. else
   3. generate frames for current sound frame generate blended sound for next frame
6. end if
   1. end if
7. end loop

After these functions are run, the text is ready to be sent to the synthesizer.

Pictured in Figure 12, is the diagram that was used to construct these functions and how they fit together to be input into the synthesizer.

**Difficulties in Implementation of the Pre-processing system -**

One of the notable roadblocks encountered while implementing the pre-processor section of the synthesiser was interpolating between the different frames for the different phonemes. The aim was to interpolate between these data frames in a way that made sense and didn’t make the system sound like it was just jumping between the values too quickly. Yet, the transition could not be drawn out for too long or it would sound forced. This meant that values had to shift on a curve. To do this, weights were applied to the specific values and then the values were interpolated between each of these including the weights (with a given transition period), as the value is closer to the start, it is more strongly weighted to that value than the other, and as it travels to the next value it will become more strongly weighted to the next one.

Another notable difficulty was translating a number into words. The function for this was lengthy and complex. The way this was implemented in the end was with ‘switch’ and ‘if’ statements to represent the different stages of the number.

**Pseudocode for conversion from number to word equivalent**

1. if number contains a decimal point, split at decimal
2. find modulus 3 of number length
3. loop over number
   1. if there is a remainder
      1. extract the section of the start of the number
      2. numberToWord()
      3. test the remaining length of the string to determine what denomination should be added to the other strings(million ect)
   2. move across the string for the length of the remainder
   3. else loop for the other section of string
      1. extract the section of the start of the number
      2. convert the number into the word equivalent
      3. test the remaining length of the string to determine what denomination should be added to the other strings(million ect)
   4. move across the string for length 3
   5. end if
   6. if there is a decimal point
   7. add the word point
   8. loop for every number after decimal point
   9. get number equivalent for the number
4. end if
5. join all the word into one string

The code pictured above could have been simplified in the initial stages with a key-value map to cut down on unnecessary code and increase maintainability and readability.

In more detail, the code in the number2Word() function where a three-digit number is translated into its word equivalent. This code was very large and contained a lot of switch statements with cases for each number in each level of the number e.g. [one hundred and], [twenty], [one]. Looking back, it would have been far simpler to break the number down into its respective components then, using key-value maps again, to test the components and simply insert strings accordingly. Maps could also be reused for the hundreds and unit sections of code, making it more reusable and maintainable.

The pseudocode for an updated version of the code can be seen below, which implements an early return if the number was in the teens, increasing efficiency.

**Updated Pseudocode for conversion from number to word equivalent**

1. split number into its different digits
2. loop for length
   1. if length == 2 and number >=10 AND < 20
      1. map digit to teens digit
      2. return
   2. end if
   3. if remaingLength = 3
      1. get value from map of digit
      2. add “hundred and” to the text
   4. else if remainingLength = 2
      1. get value map of tens digit eg “thirty”
   5. else if Remaininglength == 1
      1. get value from map digit
   6. end if
3. return combined number string

**The User Interface**

The user interface was originally designed in the online UI wireframing tool wireframepro, and a design was sketched out to create the initial component configuration that would be needed for a user to interact easily with the synthesiser.

Letter

Description automatically generated with low confidence

Figure . Intital wireframe of user interface

The design was then implemented in MATLAB with the app designer tool.

Graphical user interface, application

Description automatically generated

Figure . First functioning prototype of user interface

This version of the app was adjusted after consulting potential users of the system. The design using a black background and large blue text boxes was found to be popular. Additionally, the high contrast text is perfect for users who may have a visual impairment.

The UI was later redesigned after more consultancy with users that pointed out that more information about the sound produced would improve the experience of using the system. Subsequently, a waveform analyser and a frequency domain plot were added to the system. Again, the initial design was made using wareframepro.

Diagram, shape

Description automatically generated

Figure . Wireframe of redesign of user interface

As can be seen in the wireframe of the redesigned UI above, the input area has been made larger for users who wish to input larger sections of text. More details about the sound have been added using the frequency domain and waveform analyser plots. Additionally, a new button has been added so that a user could input text synthesis and replay it.

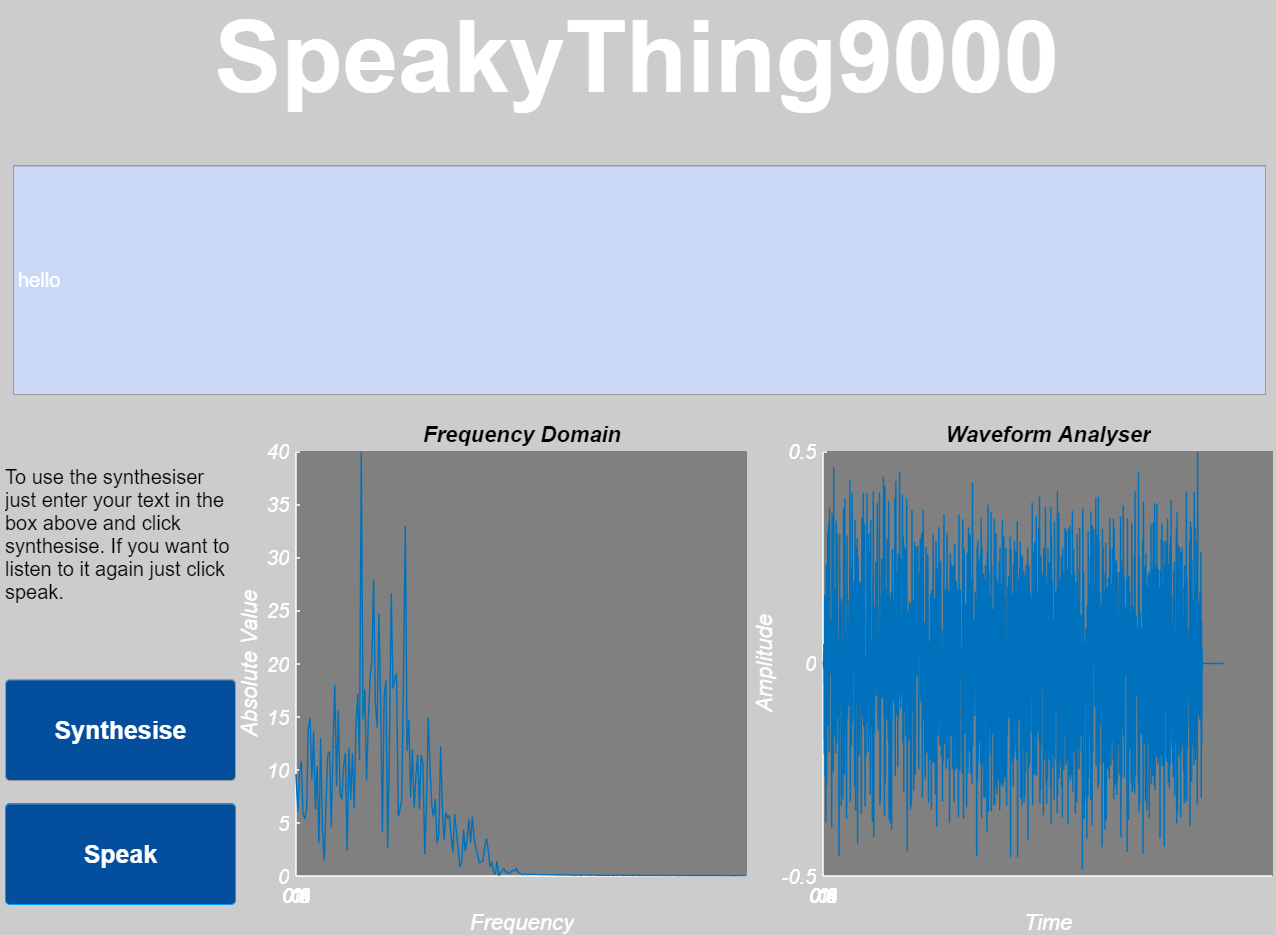


Figure . final implementation of user interface

The final UI design in MATLAB consisted of a waveform analyser and a frequency domain graph that is created every time a new sound is made.

From testing with users, this was found to be an excellent setup and every test subject was able to easily navigate the system.

**Testing -**

**Functionality Testing-**

In order to test the different sections of the system, using test cases for the different methods and classes was required. Testing the pre-processor section consisted of running individual methods with expected input.

**Testing of the synthesiser section –**

Each class had a selection of methods that needed to be tested. For basic functionality testing, the different methods have been tested to ensure that their methods return the expected values from calculations and changes to variables according to what is expected from the design.

|  |  |  |
| --- | --- | --- |
| **Class** | **Test** | **PASS/FAIL** |
| Resonator | Constructor – generates new object with specified parameters | PASS |
|  | set – sets the values of the object | PASS |
|  | setPassthough - sets passthrough to true and mute to false | PASS |
|  | setMute-sets mute to tre passthrough to false | PASS |
|  | adjustImpulseGain - adjusts the gain of the resonator | PASS |
|  | step - adjusts values in the anti-resonator from calulation | PASS |
| AntiResonator | Constructor – generates new object with specified parameters | PASS |
|  | set – sets the values of the object | PASS |
|  | setPassthough - sets passthrough to true and mute to false | PASS |
|  | setMute-sets mute to tre passthrough to false | PASS |
|  | adjustImpulseGain - adjusts the gain of the anti-resonator | PASS |
|  | step - adjusts values in the anti-resonator from calulation | PASS |
| DifferencingFilter | Constructor – generates new object with specified parameters | PASS |
|  | step - adjusts values in the differencing filter from calulation | PASS |
| FrameParms | Constructor – generates new object with specified parameters | PASS |
| GeneralParameters | Constructor – generates new object with specified parameters | PASS |
| LpFilter1 | Constructor – generates new object with specified parameters | PASS |
|  | set – sets the values of the object | PASS |
|  | setPassthough - sets passthrough to true and mute to false | PASS |
|  | setMute-sets mute to tre passthrough to false | PASS |
|  | adjustImpulseGain - adjusts the gain of the resonator | PASS |
|  | step - adjusts values in the anti-resonator from calulation | PASS |
| LpNoiseSource | Constructor – generates new object with specified parameters | PASS |
|  | getNext - Calculates the next outputvalue from the synthesiser | PASS |
| NaturalGlottalSource | Constructor – generates new object with specified parameters | PASS |
|  | startPeriod - assigns values to the NGS upon creation | PASS |
|  | getNext - Calculates the next outputvalue from the synthesiser | PASS |
| Generator | Constructor – generates new object with specified parameters | PASS |
|  | generateFrame - generates output array for sound production | PASS |
|  | computNextOutputSignal - outputs generator object updated to next step | PASS |
|  | computeCascadeBranch - returns value from cascade branch comutation | PASS |
|  | computeParallelBranch - returns value from parallel branch computation | PASS |
|  | startNewPeriod - adjusts object with new frame parameter and periodLength | PASS |
|  | startUsingNewFrameParameters - called by startNewPeriod, changes to new Frame parameters in object | PASS |
|  | initGlottalSource - initialises the glottal source wave generator | PASS |
|  | startGlottalSourcePeriod - used by initGlottalSOurce to run start period function of a glottalsource object | PASS |
|  | setTiltFilter - sets the value of the tilt filter | PASS |
|  | setNasalFormantCasc - sets the nasal formant values for the cascade branch | PASS |
|  | setNasalAntiFormantCasc - sets the nasal antiformant values for the cascade branch | PASS |
|  | setOralFormantCasc - sets the oral formant values for the cascade branch | PASS |
|  | setOralFromantPar - sets the oral formantvalues for the parallel branch | PASS |
|  | setNasalFormantPar - sets the nasalFormant values for the parallel branch# | PASS |
|  | adjustsignalgain - adjust signal so that the gain isn't too high by caluculating rms | PASS |
|  | setFstate - sets the parameters for the fState object | PASS |

Table 1. functionality testing for the formant synthesiser section of the program

In Table 1, the testing for the methods can be seen. This is the basic level view showing the different methods passing in normal circumstances. Edge case and extreme test cases have not been included due to time constraints.

**Testing the Pre-processing section –**

The pre-processing section of the synthesiser testing was conducted by first testing the output of individual functions and then the output of the whole section on its own.

This section of the system was tested with normal, edge and extreme cases.

**Testing of the text-to-phoneme conversion and Text Reformatting-**

These sections of code were tested together as they are closely related and have a strong effect on the success of the system. These test cases consisted of:

Normal cases – strings consisting of words with no numbers, abbreviations, or acronyms.

Edge cases – strings with words, numbers, abbreviations, and acronyms,

Extreme cases – strings consisting of incorrect punctuation and characters.

Normal test cases were able to return a sound in most cases initially, but in some cases the synthesiser would struggle with phoneme conversion cases and the way that they were split up.

Edge cases also returned sound in most test cases, but again had the same problem as above.

In extreme cases, strings returned with an alert that the string didn’t follow the rules of conversion and that there was an error.

Based off the testing above, improvements could be made to the storage of phonetic sounds to stop errors. This could be implemented before the phoneme-to-sound conversion and would actually cut down the length of code needed in the phoneme conversion, as the section to handle diphthongs would not be needed.

Better handling of extreme cases could also be implemented. In the cases of unpredicted punctuation, additional regular expressions could be used to remove these so that the system has the best chance of playing the correct sound.

**Testing of the phoneme-to-sound section** -

This section of function was again tested with multiple types of test cases:

Normal – The phonemes that are passed in are all contained in the conversion map.

Edge – The phonemes are contained in the phoneme map, but there are no sound parameters contained for it.

Extreme – the phonemes passed in are not contained in the map.

Normal edge cases function as expected – sound frames are produced.

Edge cases - edge cases are impossible when the synthesiser is in practical use. These were still included for future developments on the system though, and changes to the rule set. Currently there is no handling for these cases, but implementation of this would be essential if the synthesiser were to be expanded upon. Currently when edge cases are found the system still returns the same value as would be for the extreme cases.

Again, extreme cases are not handled at this level due to the nature of the system, but they should be. In the future, this should be implemented.

**Measures in comprehensibility –**

To calculate the comprehensibility of the system, tests were performed based off some of the testing conducted in the MITalk development (Jonathan Allen, M. Sharon Hunnicutt, Dennis H. Klatt, Robert C. Armstrong, and David B. Pisoni. 1987), the most notable of which was the rhyme test. Due to the current situation of the COVID-19 pandemic, this test was conducted on a limited sample size as the conditions of the testing needed to be correctly controlled to ensure continuity between participants.

**Procedure –**

Using the Rhyme Test (Fairbanks 1958), a procedure was created to test the recognition of different phonetic segments. This testing procedure was chosen as it meant that people who had no knowledge of phonetic notation could participate. The test consists of 20 questions with 6 answers for each question. The tester would be played the sound as many times as they wished before selecting the word that they believed to be the one that they heard. The test was conducted in the same space to limit any disturbances to the test, the different test subjects would wear the same set of headphones and would answer questions through a Google form on their own smartphone or tablet device.

**Results and discussions –**

The focus of this testing was to investigate the phoneme recognition of different words. Although the amount of test data acquired is limited, the data can still be of use in making adjustments to the system.

Table 2. Percentage of successful recognition of beginning phonemes

Looking at the results in Table 2, it is visible that the main areas that test subjects struggled with were the DD and CC sounds.

Upon further investigation of the DD sound, the difficulty in some cases was that the test subject would miss this part of the word or confuse the sound with GG. Possible remedies to this instance are increasing the gain and extending the DD sound for longer.

Investigating the KK sounds found that test subjects were confusing this with a either a DD or SS sound. Further investigation would be needed to produce definitive results, but confusion between the DD and KK sounds would make sense as they are similar.

Looking at results from the AA sounds, it seems that vowel sounds have been successful, but again, more investigation would be necessary to make a 100% confident claim.

Within these tests it was found that:

1. More research should be conducted into the sounds that contain stops, e.g., KK, DD, BB and GG as they are clearly the lower performing sections of the synthesiser.
2. More testing should be conducted into the different words that contain these types of sounds to get a clearer picture of the issues in these cases.

**Improvements –**

From the results found, it can be hypothesised that the production of phonetic components from the synthesiser is of a high standard especially while producing vowel sounds.

This test would need to be conducted again when a global pandemic isn’t present. Because of the restrictions in place and the need to have a consistent environment free from background noise and with specific equipment, the sample size is small. Tests would need to be conducted with a larger sample size to:

1. Confirm findings.
2. Investigate in greater details what changes and adjustments need to be made to the synthesiser to improve phonetic differentiability.

**Testing word recognition in sentences -**

**Procedure –**

Based of testing conducted in MITalk(Jonathan Allen, M. Sharon Hunnicutt, Dennis H. Klatt, Robert C. Armstrong, and David B. Pisoni. 1987.) tests were conducted where a user would listen to sentences recorded and then be asked to identify certain words that were played. For example, the sentence played could be “hello, how are you?” and the test subject would be asked to identify the 2nd word. The subject would write down what they think the word is and then proceed to the next question. The sentence would only be played 3 times for the test subject before they had to give their answer.

Figure 19. Test results for word in sentence recognition

**Results and discussion –**

This testing was less successful than the first set of tests conducted, only 10% of the sentences tested found the words understood correctly.

While conducting the tests, test subjects were asked if they were struggling to understand the sentences spoken. All subjects said that they were struggling, and identified the problem was the pronunciation and intonation of the sentences.

From these results, it can be hypothesised that the main problem with the comprehension of the system is the pre-processing section, as from the rhyme tests the recognition of different phonetic segments was adequate. In the future, a way to improve this would be to redesign the pre-processing section from the ground up by incorporating additional stages for analysis of words, so that the correct pronunciation can be found. This would involve more investigation into the prosodic and morphological components of speech which were out of the scope of this project.

**Evaluation -**

**Usability –**

From the testing conducted, it can be seen that the system developed is usable and understanding words (in isolation) is possible, but not in larger sentences. With further testing and analysis, a more specific set of improvement could be formed.

Tests should also be conducted to see how the system performs reading larger chunks of text and also measure participants’ ability to interpret their meaning.

**Non-functional Requirements** –

A requirements document was made for the project – some non-functional requirements were created, and some have not been implemented in the system – for example letting the user change the intonation of words, add stress to different sections of words of their choosing, and change the style of the speech that the system is producing.   
Other requirements have been met, including producing speech in under 5 seconds (for a sentence), a simple interface, accessibility and ease of use, and words produced (in isolation) are able to be understood by an English speaker.

**Survey –**

In the survey that was conducted on the same participants as in the Rhyme test and word-in-sentence-identification-test, it was found that many of the participants thought that the synthesiser was difficult to listen to. They said that the voice didn’t just sound robotic, but also had a very unnatural feeling to it. To improve the sound, a different glottal source wave form was used - this produced improved user feedback.

**Description of final synthesiser –**

The final synthesiser consists of a user interface that will take user inputted text, which will then be processed first by the pre-processor section of the program which will then pass an array of data frames representing each 5 milliseconds of sound parameters for the synthesiser. The synthesiser will use these data frames to produce a sound signal resembling English speech. This speech is then outputted, and the resultant array of values are transformed to produce graph plots of the sound waveform and the frequency domain of the sound.

**Summary and Conclusions -**

To conclude, the process of designing, developing and testing a formant speech synthesiser in MATLAB has been completed. The system has still not reached its full potential, and a redesign of the pre-processing unit would be beneficial to the system as a whole to improve the comprehension of sentences of text. From the testing of the recognition of different phonemes, it is seen that the functionality of the synthesiser part of the program is effective and allows adequate translation. This project has had to close without satisfactory completion of all the requirements, but this report lays out future improvements and adjustments that will turn it into a fully functioning system.

**Appendix –**

For setup and use of the current synthesiser program, there are two routes a prospective user could take.

1. Running the MATLAB application – to do this the user needs to navigate to the app file “SpeakyThing” in the projects folder and double click. Taking this action will navigate the user to MATLAB where the user will be prompted to install the application. After successful installation, navigate to the “APPS” tab open the drop down menu containing the different MATLAB applications. Find the app SpeakyThing and click it. The application will then run and you will have full use of the application.
2. Using folder setup with direct access to source code – Open the folder containing the different files of the synthesiser. From here, the application with the user interface can be accessed by clicking on the “app1” file and navigating to the “Run” button. The synthesiser can also be interacted with directly through the command line. To begin using the command line, some additional steps need to be followed. First, the variable set needs to be loaded in. This is stored in the file “recentVars.mat” to load these, use the command load(“recentVars.mat”) and observe that the workspace tab is now populated with the variables that the program needs to function.

**References**

Cs.mcgill.ca. 2021. *Speech Synthesis*. [online] Available at: <https://www.cs.mcgill.ca/~rwest/wikispeedia/wpcd/wp/s/Speech\_synthesis.htm#:~:text=The%20first%20computer%2Dbased%20speech,the%20history%20of%20Bell%20Labs.> [Accessed 26 January 2021].

Klatt, D., 1980. *Software For A Cascade/Parallel Formant Synthesizer*. [online] Fon.hum.uva.nl. Available at: <https://www.fon.hum.uva.nl/david/ma\_ssp/doc/Klatt-1980-JAS000971.pdf> [Accessed 20 January 2021].

Wiki.inf.ed.ac.uk. 2021. [online] Available at: <https://wiki.inf.ed.ac.uk/twiki/pub/CSTR/Speak14To15/evaluation.pdf> [Accessed 27 January 2021].

J & W Holmes. 2002. Speech Synthesis and Recognition (2nd. ed.). Taylor & Francis, Inc., USA.

Jonathan Allen, M. Sharon Hunnicutt, Dennis H. Klatt, Robert C. Armstrong, and David B. Pisoni. 1987. *From text to speech: the MITalk system*. Cambridge University Press, USA.

d'Heureuse, Christian. "Chdh/Klatt-Syn". *Github*, 2019, [https://github.com/chdh/klatt-syn. Accessed 4 Feb 2021](https://github.com/chdh/klatt-syn.%20Accessed%204%20Feb%202021).

Research.spa.aalto.fi. 2021. *History and Development of Speech Synthesis*. [online] Available at: <http://research.spa.aalto.fi/publications/theses/lemmetty\_mst/chap2.html> [Accessed 12 April 2021].

Klatt D. (1987) Review of Text-to-Speech Conversion for English.*Journal of the Acoustical Society of America, JASA* vol. 82 (3), pp.737-793.

[*Fairbanks*](https://asa.scitation.org/author/Fairbanks%2C+Grant) *1958.* [The Journal of the Acoustical Society of America](https://asa.scitation.org/journal/jas) **30**, 596 (1958); <https://doi.org/10.1121/1.1909702>

D.H. & L.C. Klatt (1990), "Analysis, synthesis and perception of voice quality variations among male and female talkers"(http://www.source-code.biz/klattSyn/Klatt-1990.pdf))

|  |
| --- |
| . |
|  |