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.

**Overview of the System –**

The MATLAB formant synthesiser (MATalk) has been developed off of the MITalk system. 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 calculation. 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 in inputted into the synthesiser, which will then use these frames to create an array output of the sound that these data frames represent. Finally, the sound is outputted 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 talking.

//TODO: add more details about modern synthesis technology

**Design –**

The design process 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, which uses a large database of recorded speech when 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, which 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, which concatenates pre-recorded words and phrases to create complete utterances, is generally only used in speech variance as it will be 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, 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.   
After more investigation into other options and reading speech synthesis and recognition (John and Wendy Holmes) and MITalk, this method 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 synthesizer.

Predesign for the MATLAB program was split into two parts-

1. The formant synthesizer, based off the parallel/cascade design (Klatt. D, 1980)
2. The text pre-processor for input into the synthesizer

**Diagram

Description automatically generated**

**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.

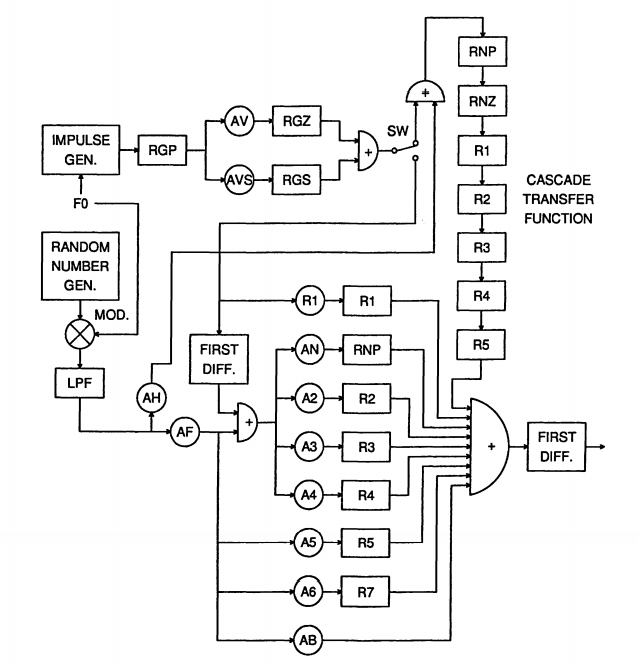


Figure 3. Block diagram of Klatt cascade/parallel formant synthesizer.

Figure 2 shows the model of the Klatt formant synthesizer proposed in his paper (Klatt, D., 1980.)

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’s the generator object, which is responsible for creating the sound waveforms from the parameters inputted. The main function is the. 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 doubles that can be used to produce a sound using the sound values.

**More details on the objects-**

FrameParms- This class represents the values of the dataframes that will be used by the synthesiser to generate the speech wavefrom. The synthesiser will be passes a new set of frame parametersfor every 5ms of sound produced.

Resonator – This class represents the Resonantor, from the klatt formant synthesiser(kKlatt, D., 1980) the resonators. In the class diagram shown a,b and c represent the filter coefficients for the calculation of the next output value, r is used as a tempcalucualtion for the c and b coefficient calculations. Y1 and y2 represent the last and second last output value of the resonator. The Object contins functions to use the resonator as a passthrough, muted, adjust the gain of the system and step to calculate the next value in the synthesiser. The step value performs the calculation:



Figure . Digital resonator calculation

Where A,B and C represent the fiter coefficients, nT represents the number passed into the step function (nT-T) is the previous output value of the resonator and (nT-2T) 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 AntiResonator (Klatt,D. 1980)

**Difficulties in creation –**

When constructing the synthesizer, a waterfall method of production was used. Generally, to have a system that produced any sound at all, the whole base structure of the synthesizer would be needed. Because of 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.

Nasalisation – when attempting to create nasalisation in the synthesizer, a buzzing sound is produced that made the important section of the sound inaudible. This was fixed by adjusting the nasal antiformant 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**

**Pre-Processing unit**

The pre-processing unit was developed using an agile approach. Each part 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 and 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**

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

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.

**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.

**Sound Duration Computation**

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

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.

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

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

Pictured in figure 3, is the diagram that was used to construct these functions and how thy 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.

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

Line chart

Description automatically generated

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. The pseudocode below shows this process:

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

In more detail, the code in the numberToWord() 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

**Designing 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

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

Graphical user interface, application

Description automatically generated

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 . New UI design

As can be seen in Figure 4. New UI design, 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.

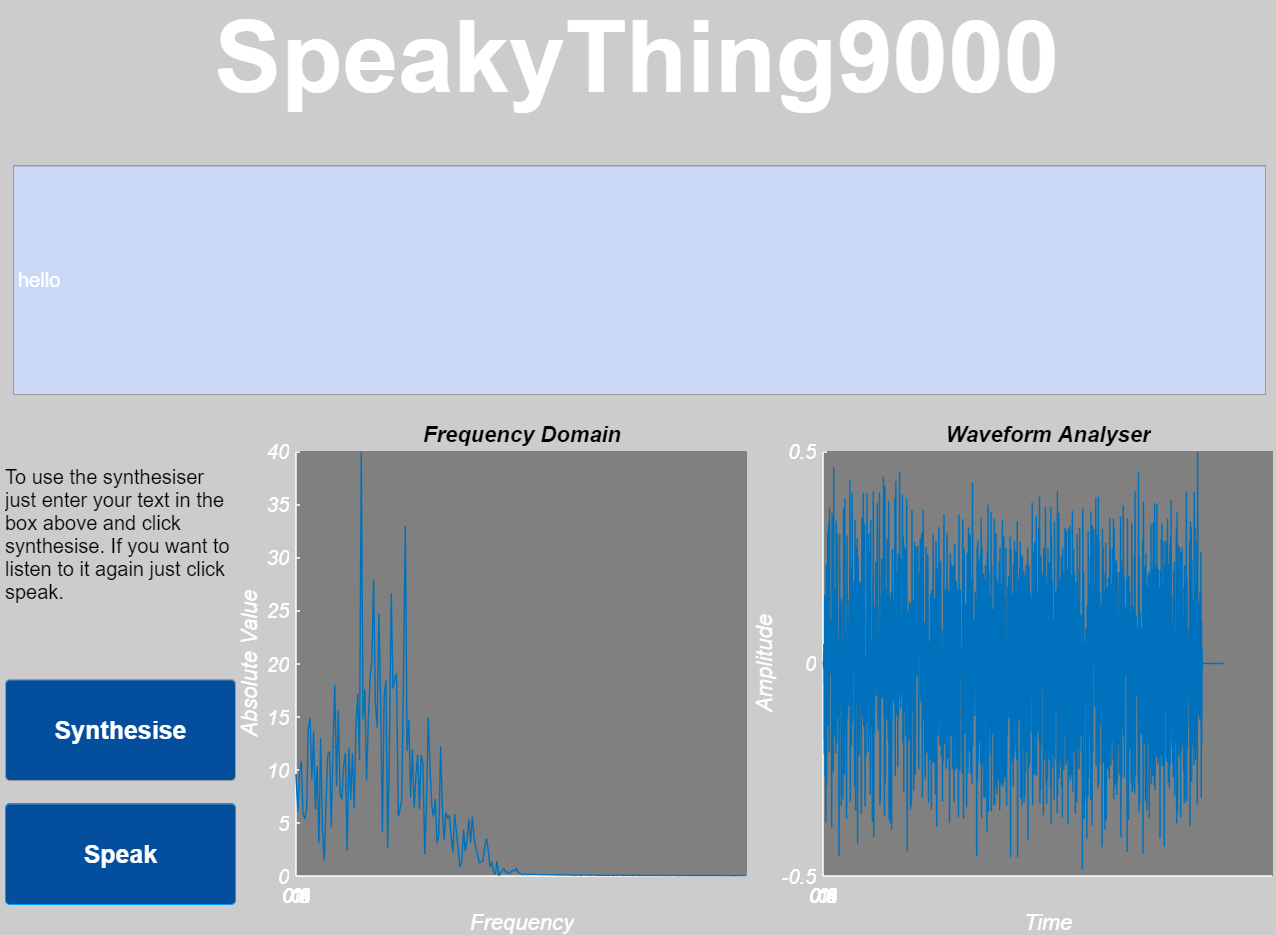


Figure . final UI design

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.

**Functionality testing of the system -**

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 that their methods return the expected values from calculations and changes 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 on 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 actually impossible when the synthesiser is in practical use. These were included for future developments on the system and changes to the rule set. Currently there is no handling for these cases, but implementation of this would be essential if the synthesiser was to be expanded upon. And edge cases if it was caught in the program would simply tell the user that they could use a certain word.

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 correct one. 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 from 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 confused 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.

Improvements –

The test itself would have been improved greatly with a larger sample size to get more accurate averages for the testing of the different sounds, understand better what sounds people struggled with and how to best adjust the system to suit a wider audience. 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. 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.

**Testing word recognition in sentences-**

To get a better idea for the recognition and understandability of words in sentences, tests were constructed 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 could 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.

This testing was less successful than the first set of tests conducted, as very little of what was played for the users was identified correctly.

When asking test subjects what they were struggling with consistently the problem was the pronunciation and intonation of the sentences. From these results it can be concluded that to improve the results for this it would be a good option to create more functions for calculating durations of sounds so that the synthesiser can follow the rules for the given accent that it is using. It would also be advantageous to use phoneme conversion rules for British English as the speakers who were participating in these tests were a majority British English speaking individuals.

**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 more in depth analysis of the improvements

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

**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), having the text in a processed form so that the user read along with the speech, a simple interface, accessibility and ease of use, and words produced can 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 produce improved user feedback.

Appendix –

Include pass over manual for someone to pick up the project in the future.

**Description of final synthesiser –**

The final synthesiser consists of a user interface that will take user inputted speech, which will then be processed by first the Pre-processor section of the program which will pass an array of data frames representing each 5 millisecond of sound parameters for the synthesiser. The synthesiser will then 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 this text-to-speech synthesis application will take in user inputted text and output the text as spoken word.

The project was implemented between 28th of October 2020 and the 27th of April 2021.

**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].

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>

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