**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 other adapted versions that weremore 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.

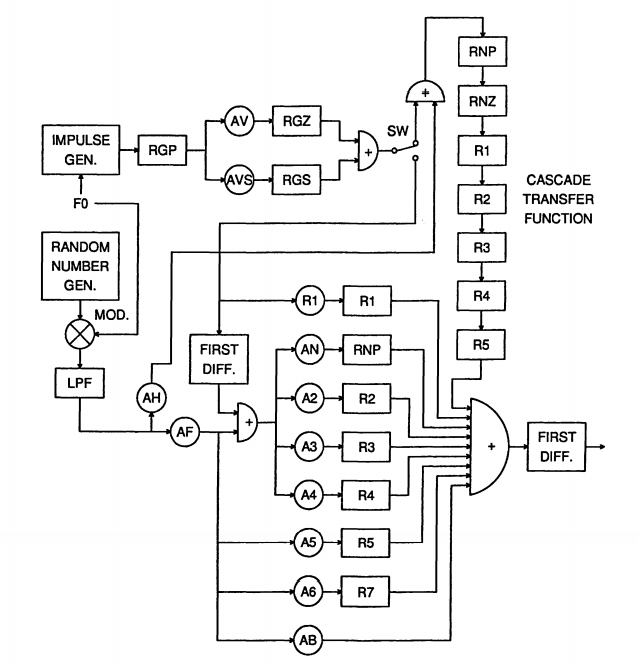


Figure 4. Block diagram of Klatt cascade/parallel formant synthesizer.

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 generator object, which is responsible for creating the sound waveforms from the parameters inputted. The main function is the generateSound() function. 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.

**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**

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 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

**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.

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

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.

**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

**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.

Line chart

Description automatically generated

Another notable difficulty was in 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:

**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 code necessary and increase maintainability and readability.

In more detail the code in the numberToWord() function where a three-digit number is translated into it’s 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 code 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