Linguistic Portfolios

Volume 13 Article 6

2024

# SYNTHESIZERS DEMONSTRATED ON BETINE

CLARE MILLER

ISABELLA RANDOLPH

Linguistic Portfolios – ISSN 2472-5102 –Volume 13, 2024 | 80

# DIGITALIZED SPEECH SYNTHESIS: A COMPARATIVE ANALYSIS OF SPEECH SYNTHESIZERS DEMONSTRATED ON BETINE

CLARE MILLER AND ISABELLA RANDOLPH[1]

## ABSTRACT

*This paper provides basic information on developing speech synthesis to help preserve and revitalize critically endangered languages. It uses Betine, (ISO 639-3:eot), as a model. The death of minority languages is escalating globally, and 90% are predicted to die by 2100, which results in the loss of cultural heritage and knowledge. Synthesizing speech in dying and near-extinct languages can preserve and even revitalize them. The paper includes spectrographs and waveforms of the given Beti name <Adjo> transcribed phonetically as [a:ɟo]. It also contains voice component measurements and the synthesizing programs used for comparison. The measurements taken from the data include F0/pitch, formants (F1, F2, F3, F4, F5), amplitudes (A1, A2, A3, A4), and A5, intensity, duration, and bandwidths (B1, B2, B3, B4, and B5). The paper notes that chunking longer phonemes into multiple short samples can give better synthesized results. Klatt, KlattGrid, and WORLD synthesizers are used for synthesizing the Betine speech segment. The paper proposes that vocoder synthesizers such as WORLD may be worth greater research concerning their capabilities in artificial speech synthesis.*

**Keywords:** Betine, Beti, Speech Synthesis, Formant Synthesis, Klatt Synthesizer, KlattGid, Tdklatt, WORLD Synthesizer, Formant Extraction, Formant Bandwidths, Endangered Languages

## 1.0 Introduction

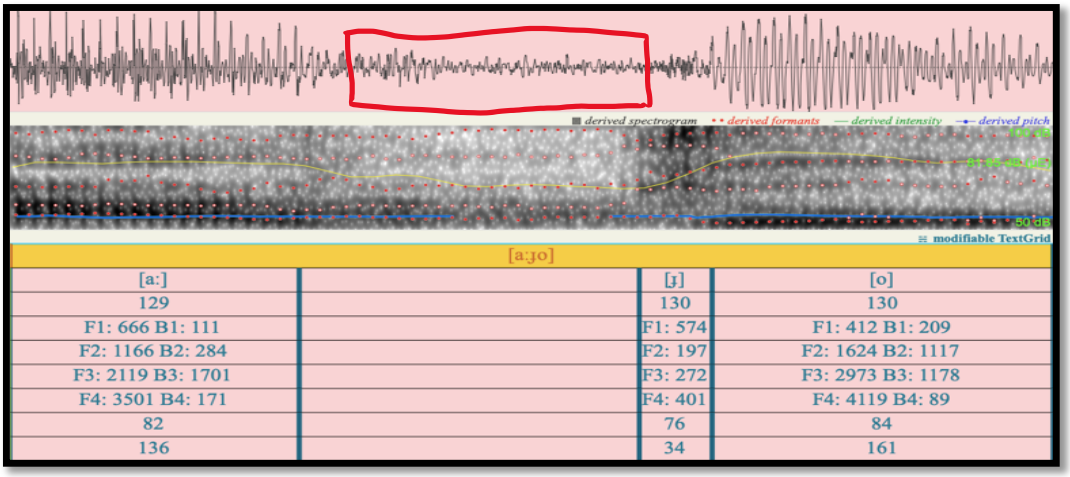
While the death of languages is not a new phenomenon, it is quickly escalating on a global scale. The United Nations Educational, Scientific, and Cultural Organization (UNESCO) estimates that by the year 2100, 90% of the world's languages will have died, as cited by Koffi (2021:23). As a language dies, the accumulated knowledge of all the generations who spoke it vanishes, not to mention the loss of cultural heritage. With this gravity in mind, the paper attempts to find an efficient and effective way to synthesize speech, using the critically endangered language Betine (ISO 639-3:eot) as a model. It is spoken in the lagoon areas of Côte d'Ivoire, West Africa. Successfully synthesized speech for dying languages should be interactive and have the same capabilities as systems such as Siri and Alexa. If it does, it could indefinitely preserve and even revitalize the languages that will otherwise disappear. For this demonstration, our final class project focused on an audio sample of the female Beti name **<Adjo>** transcribed phonetically as **[a:ɟo]**. The paper includes codes and synthesizing programs that made the synthesis possible, as well as a comparison of our results from each of those programs.

## 2.0 Spectrograms and Measurements

There are many instruments used in speech synthesis. Spectrographs, visual representations of sound, are helpful for viewing pitch and formants, among other speech components. Spectrographs

[1] **Recommendation:** This paper was recommended for publication by **Dr. Ettien Koffi** and **Dr. Mark Petzold** who taught the Speech Signal Processing and Coding course from which the paper originated. Dr. Koffi verified the accuracy of the acoustic phonetic information and Dr. Petzold did the same for the accuracy of the scripts and codes.

give information about frequency over time, while waveforms display amplitude over time. All the spectrographs shown in this paper are generated by Praat, an acoustic speech signal processing program that is user-friendly and widely accepted for various uses involving speech. Below is a spectrograph and waveform of our given word, [à:ɟó]. It also includes the measurements taken. There is a notable gap between the segments [à:] and [ɟ]. This is because the space was determined to have a drawn-out and echo-like quality exhibited by the speaker, likely due to an exaggerated and drawn-out pronunciation, which would otherwise not be the case in running speech. This area is indicated by the red rectangle in Figure 1:




| [a:] | | [ɟ] | [o] |
|---|---|---|---|
| 129 | | 130 | 130 |
| F1: 666 B1: 111 | | F1: 574 | F1: 412 B1: 209 |
| F2: 1166 B2: 284 | | F2: 197 | F2: 1624 B2: 1117 |
| F3: 2119 B3: 1701 | | F3: 272 | F3: 2973 B3: 1178 |
| F4: 3501 B4: 171 | | F4: 401 | F4: 4119 B4: 89 |
| 82 | | 76 | 84 |
| 136 | | 34 | 161 |

Figure 1: Spectrograph of [à:ɟó]

**3.0 Measurements**

The measurements taken from our data and used in synthesis are as follows: F0/pitch, also known as fundamental frequency, formants (F1, F2, F3, F4, F5), amplitude (A1, A2, A3, A4, A5), intensity, duration, and bandwidth (B1, B2, B3, B4, and B5). F0 is the baseline pitch in a sound. It changes depending on the sound uttered and the characteristics of the speaker. Upon F0 are built other frequency elements such as formants, which are sound qualities formed by resonances in the vocal tract and produce a filter effect. F1 corresponds to how wide the mouth is (mouth aperture), F2 to the length of the oral cavity (depending on horizontal tongue movement), F3 to the rounding of the lips, and F4 to head size. F5 was also collected from Praat but considered a possibly inaccurate result due to the abstract definition of Formant 5 mentioned in Koffi and Petzold (2022). B1 to B5 refers to the bandwidth measurements corresponding to F1 to F5. A1 to A5 refers to the amplitude of the corresponding formant (F1 to F5). This is obtained by converting the sound segment to a spectrum and finding the nearest maximum intensity for a given frequency. The equivalent method in Praat would be to convert the sound to a spectrum and call the query function "Get sound pressure level of nearest maximum…"

The sound file was also downsampled from 44100 Hz to 10000 Hz, as recommended by Koffi and Petzold (2022). These measurements were taken by hand after setting boundaries in Praat and using simple "get __" functions. We would like to note that the gap previously mentioned between [à:] and [ɟ] similar measurements to the first segment, indicating that it might be an extension of the vowel. However, the intensity decreases noticeably as indicated by the faint yellow line found in the spectrograph (Figure 1), and synthesis of the gap results in a distorted sound.

Originally, we tried to synthesize speech by plugging in values obtained from Praat (see Figure 10). However, due to the time it takes to manually get each value and the decision to "chunk" longer phonemes into shorter ones, we decided to use Parselmouth to dynamically find the average values. Previous projects showed that these values are slightly different from what one would find through Praat - likely due to some subtle difference in the algorithm Praat uses to calculate mean values - but are nonetheless close enough to use without worry. We are including the measurements from Praat for the sake of comparison, even though they were not used for synthesis. We note in passing that the only measurements we used from Praat are the start time and end time of a speech segment. This is only used for the Klatt synthesizer and not the KlattGrid or WORLD synthesizers. As for chunking phonemes, the Klatt synthesizer does not sound correct when we are playing a single phoneme over a long duration. As such, we decided to split longer phonemes into shorter ones. Our first version with Praat synthesis chunks the first and last phoneme into two separate speech segments for a total of 5 chunks. A revised version chunks each speech segment into even smaller chunks around 10 msec long, for a total of 33 chunks.[2] Below are the code samples that can be used to obtain each measurement.

```
def getPitch(sound, startT, endT):
    pitch = call(sound, "To Pitch", 0.0, 75, 600)
    pitch_avg2 = call(pitch, "Get mean", startT, endT, "Hertz")

    return pitch_avg2
```

Figure 2: Pitch Extraction Code

GetPitch() returns F0/pitch measurements for a given time range (Figure 2). Note that we call the Praat method via Parselmouth rather than getting each pitch data frame and averaging it. We do not use this method for KlattGrid measurements.

```
def getIntensity(sound, startT, endT):
    intensity = sound.to_intensity()

    return intensity.get_average(from_time=startT, to_time = endT)
```

Figure 3: Intensity Extraction Code

GetIntensity() returns the average intensity for a given time range (Figure 3). GetFormant() also returns the average formant *and* bandwidth for a given formant number (Figure 4).

[2] Editor's note: In speech signal processing, 20 msecs are usually preferred because it is deemed that speech signals are invariant within 20 msec frames.

```
def getFormant(sound, fn = 1, startT=0, endT=999):
    formants = sound.to_formant_burg()
    formant_values = []
    bandwidth_values = []
    num_frames = formants.get_number_of_frames()
    for u in range(num_frames):
        t = formants.get_time_from_frame_number(u+1)
        if t >= startT and t <= endT:
            v = formants.get_value_at_time(fn, t)
            bw = formants.get_bandwidth_at_time(fn, t)
            if not np.isnan(v):
                formant_values.append(v)
            if not np.isnan(bw):
                bandwidth_values.append(bw)

    if len(formant_values) == 0:
        formant_values.append(0)
    if len(bandwidth_values) == 0:
        bandwidth_values.append(0)

    return np.average(formant_values), np.average(bandwidth_values)
```

Figure 4: Formant Extraction Code

GetFormantAmplitude() converts the sound to a spectrum and gets the intensity for a given formant number (Figure 5). We were unsure whether or not this needed to be pre-emphasized first.

```
def getFormantAmplitude(sound, formant_freq=300, startT=0, endT=0):
    # NOTE: Should we pre-emphasize this first?
    if formant_freq <= 0 or formant_freq >= 6000:
        return 0

    sound = sound.extract_part(from_time=startT, to_time=endT)
    spectrum = sound.to_spectrum()

    amplitude = call(spectrum, "Get sound pressure level of nearest maximum...", formant_freq)

    return amplitude
```

Figure 5: Amplitude Extraction Code

For our Klatt synthesizer, this is the function we used to fill every parameter we want to obtain (Figure 6).

```
def getAllData(sound, startT, endT, parameters):
    data = {}
    ff = []
    bw = []
    for u in range(5):
        fn, bwn = getFormant(sound, u+1, startT, endT)
        ff.append(fn)
        bw.append(bwn)
    for i in range(len(parameters)):
        p = parameters[i]
        if p == "F0":
            data[p] = getPitch(sound, startT, endT)
        elif p == "FF":
            data[p] = ff
        elif p == "AV":
            data[p] = getIntensity(sound, startT, endT)
        elif p == "BW":
            data[p] = bw
        elif p == "DUR":
            data[p] = getDuration(startT, endT)
        elif len(p) == 2 and p[0] == "A" and p[1].isdigit():
            n = int(p[1])
            if len(ff) >= n:
                data[p] = getFormantAmplitude(sound, ff[n-1], startT, endT)
            else:
                data[p] = 0
```

Figure 6: Extraction Codes in Klatt Synthesizer

The method used to get the value for each pitch data frame and add it to the KlattGrid object (Figure 7). Note that unlike in Klatt, we obtain the measurement by obtaining the pitch value for every frame. This is because we do not need an average for a given time value, just what the value is at that time.

```
pitch_frames = pitch.get_number_of_frames()
for i in range(pitch_frames):
    pitch_value = pitch.get_value_in_frame(i+1)
    if not np.isnan(pitch_value):
        cur_t = pitch.get_time_from_frame_number(i+1)
        call(klattgrid, "Add pitch point...", cur_t, pitch_value)
```

Figure 7: Pitch Codes for KlattGrid

The method used to get the value for each intensity data frame and add it to the KlattGrid object (Figure 8).

```
intensity_frames = intensity.get_number_of_frames()
for i in range(intensity_frames):
    cur_t = intensity.get_time_from_frame_number(i+1)
    i_val = intensity.get_value(cur_t)
    call(klattgrid, "Add voicing amplitude point...", cur_t, i_val)
```

Figure 8: Intensity Codes for KlattGrid

The method used to get each formant, alongside corresponding bandwidth and amplitude (Figure 9). It is then added to the KlattGrid object. For the Klatt synthesizer and the KlattGrid synthesizer, we obtain data roughly the same way through Parselmouth. However, the WORLD synthesizer uses a different set of measurements. It uses new algorithms to estimate various measurements as quickly as possible. These measurements include the F0 contour at each time point, the harmonic spectral envelope, and the aperiodic spectral envelope. These measurements may be obtainable through Praat, but if they are, the method is unknown to us.

```
formant_frames = formants.get_number_of_frames()
for i in range(formant_frames):
    cur_t = formants.get_time_from_frame_number(i+1)
    for n in range(5):
        fn = formants.get_value_at_time(n+1, cur_t)
        bw = formants.get_bandwidth_at_time(n+1, cur_t)
        if not np.isnan(fn):
            amplitude = call(spectrum, "Get sound pressure level of nearest maximum...", fn)
            call(klattgrid, "Add oral formant frequency point...", n+1, cur_t, fn)
            call(klattgrid, "Add oral formant bandwidth point...", n+1, cur_t, bw)
            call(klattgrid, "Add oral formant amplitude point...", n+1, cur_t, amplitude)
```

Figure 9: Formant Codes for KlattGrid

The F0 contour represents the pitch values at a given time frame. In this regard, it should be roughly the same as what Praat would give us. However, the method used is different. WORLD uses an algorithm called DIO (Distributed Inline-Filter Operation), as explained by Morise et al. (2009). According to Boesma (1993), Praat, on the other hand, uses an autocorrelation algorithm by default. In order to estimate F0, we also looked at using the Harvest algorithm (Morise, 2017). This gives better results than DIO but is slower. F0 is also refined using the Stonemask algorithm (Morise, 2017). Harmonic spectral envelope, or timbre, is roughly the spectrum of a sound - that is, the intensity value at each frequency value. WORLD uses the CheapTrick algorithm (Morise, 2014).

The Aperiodic spectral envelope is calculated (relative to the harmonic spectral envelope) using the Harvest algorithm. This is used in order to make for a more natural sound synthesis. Without it, the synthesized sound ends up having several robotic-sounding sections. However, we noticed that it had very little impact on our sound example. We are also using D4C (Definitive Decomposition Derived Dirt-Cheap) that estimates "band-aperiodicity."

Following the steps outlined above, we moved on to our measurements themselves. The tables below contain data extracted manually from Praat.

| Segment | [aː] | [ɻ] | [o] |
|---|---|---|---|
| F0 | 129 | 130 | 130 |
| F1 | 666 | 574 | 412 |
| F2 | 1166 | 1976 | 1624 |
| F3 | 2119 | 2724 | 2973 |
| F4 | 3501 | 4015 | 4119 |
| F5 | 4552 | 4550 | 4572 |
| Intensity | 82 | 76 | 84 |

| Duration | 136 | 34 | 161 |
|---|---|---|---|
| B1 | 111 | 508 | 209 |
| B2 | 284 | 525 | 1117 |
| B3 | 1701 | 563 | 1178 |
| B4 | 171 | 586 | 89 |
| B5 | 913 | 178 | 270 |

Table 1: Data Extracted Manually from Praat

Here, we would like to note the bandwidth measurements from Praat. Bandwidth is infamously difficult to measure, and many of our first attempts included values that were, for each formant respectively, 20% of the formant itself (Koffi, 2023).[3] As indicated by the B4 measurement of the third segment [o], the results from Praat do not always align with the parameters we would expect to see. However, for our later attempts, we used values extracted directly from the sound, as we detected no difference in the sounds produced, so they were kept in the final syntheses.

In our Klatt synthesis, we attempted two different methods: synthesis with almost as few chunks as possible and synthesis with a chunk size of 10 msec. The latter resulted in 33 chunks. Our first attempt was done using five chunks, effectively splitting the first and last vowels into two sections. We have listed the measurements for the first segment in the word to give the reader an idea as to the accuracy of measurements generated generated by code using Parselmouth. Our Parselmouth data was used for synthesis in KlattGrid, while our manual data from Pratt was used for our first attempt with five chunks. Data from Parselmouth was used in our attempt with more chunks, but the duration values from Praat were still imported. The measurements from Parselmouth are as follows:

| Segment [a:] | Chunk 1 | Chunk 2 | Average |
|---|---|---|---|
| F0 | 127 | 130 | 128 |
| F1 | 776 | 587 | 681 |
| F2 | 1176 | 1182 | 1179 |
| F3 | 2075 | 1839 | 1957 |
| F4 | 3595 | 3490 | 3542 |
| F5 | 4708 | 4380 | 4544 |
| Intensity | 82 | 82 | 82 |
| Chunk size | 0.068415 | 0.068415 | - |
| B1 | 156 | 100 | 128 |
| B2 | 361 | 202 | 281 |
| B3 | 542 | 808 | 675 |
| B4 | 184 | 121 | 152 |
| B5 | 617 | 342 | 479 |

[3] Editor's note: Bandwidth estimates mentioned here are actually based on Rabiner and Juang (1993:152). Full citation: Rabiner, Lawrence and Bin-Hwang Juang. 1993. *Fundamentals of Speech Recognition*. Prentice Hall, Englewood Cliffs: New Jersey.

Table 2: Measurements from Parselmouth

## 4.0 Klatt Synthesizer

Klatt is a formant speech synthesizer that uses a grid approach (pre-set speech parameters) for its data. It relies on manual tuning and measurement input to produce natural-sounding speech. As a result, the output depends almost entirely upon what measurements are being used. The first paper published by Klatt on voice synthesis in 1980 lists 39 parameters that may be used (Klatt, 1980). We utilized 16 of those, including F0, F1-F5, AV (amplitude of voicing), A1-A5, and B1-B5. Due to limited information on how to obtain the data needed to use other measurements, we focused on other areas as well as the parameters for our synthesis. For example, the chunk size mentioned earlier.

Formant synthesizers view voice linearly, which allows for a direct correlation for programming. Klatt's formula is as follows:

$$P(f) = S(f) * T(f) * R(f)$$

Here, (f) in each symbol indicates frequency. S(f) is the source volume velocity, T(f) is a ratio of lip-plus-nose volume velocity, U(f) is lip volume velocity, and R(f) is the radiation from the head (Klatt, 1980). According to Klatt, sound energy is activated by lung pressure, then as it travels through the vocal tract, is excited by the natural resonances of the body (formants). Voicing, aspiration, and frication are all part of the initialization of sound, which then goes through a filtering system that results in radiated sound pressure (P(f)), or the final sound. The Klatt synthesizer mimics this theory by placing frequency (F0) through filters that imitate the physiological qualities of the head and vocal tract.

For our project, we used tdklatt to synthesize each sound using the previously mentioned parameters (Guest, 2018). With the way it works, we had to create several different sound samples, play them, store the buffer data for each sound played, and then combine all of the sounds together in order to create a single sound. So, this is the code for this step-by-step.

```
parameters = ["F0", "FF", "BW", "AV", "DUR", "A1", "A2", "A3", "A4", "A5"]

soundData = {
    "a" : {
        "START_T" : 0.345405,
        "END_T" : 0.482235,
        "CUSTOM_PARAMETERS" : {
            "SW" : 1
        }
    },
    "f" : {
        "START_T" : 0.63,
        "END_T" : 0.683,
        "CUSTOM_PARAMETERS" : {
            "SW" : 1
        }
    },
    "o" : {
        "START_T" : 0.683,
        "END_T" : 0.838632,
        "CUSTOM_PARAMETERS" : {
            "SW" : 1
        }
    }
}
```

Figure 11: Codes Used for tdklatt

The first step is defining the parameters for which we wanted to obtain measurements. Thereafter, we needed to define the start and end time for each phoneme. We added the custom parameter SW for each sound segment to have the Klatt synthesizer use parallel resonators rather than cascade.

```
soundParams = []
for name, syllable in soundData.items():
    startT = syllable["START_T"]
    endT = syllable["END_T"]
    chunk_size = math.floor((endT-startT)/MIN_CHUNK_SIZE)
    time_step = (endT-startT)/chunk_size
    for i in range(chunk_size):
        newStartT = startT + i * time_step
        newEndT = startT + (i+1)*time_step
        data = getAllData(sound, newStartT, newEndT, parameters)
        if "CUSTOM_PARAMETERS" in syllable:
            for key,val in syllable["CUSTOM_PARAMETERS"].items():
                data[key] = val
        s = tdklatt.KlattParam1980(**data)

        soundParams.append(s)
```

Figure 12: Sound Parameters

The next step was to separate the phonemes previously defined into chunks. We found the measurement data and created a KlattParam1980 object using tdklatt. This information was appended into a list.

```
sounds = []
for i in range(len(soundParams)):
    s = tdklatt.klatt_make(soundParams[i])
    s.run()
    sounds.append(s)
```

Figure 13: Klatt Parameter Object

With the list of each of the parameters, we had tdklatt create the actual object that synthesizes the sound. We ran it and appended it to a list to play each one later.

```
s_data = []
for i in range(len(sounds)):
    buffer, y = sounds[i].play()
    buffer.wait_done()
    s_data.append(y)
```

Figure 14: Sound Parameters

```
def play(self):
    """
    Plays output waveform.
    """
    y = self._get_int16at16K()
    return sa.play_buffer(y, num_channels=1, bytes_per_sample=2, sample_rate=16_000), y
```

Figure 15: Sound Output

We went through each sound and played them one by one. For this, we modified tdklatt's play() method in order to return the buffer object of the sound played, as well as the data used to play the sound.

```
def combineAudio(data1, data2, window_size = 5):
    new_data = np.concatenate([data1, data2])

    w_max_int = window_size-1
    w_max = float(w_max_int)
    new_data_start = len(data1) - int(window_size/2)
    for i in range(window_size):
        alpha = float(i) / w_max
        val = data1[-i] * (1-alpha) + data2[w_max_int-i] * alpha
        new_data[new_data_start+i] = val
    return new_data

sound_buffer = s_data[0]
for i in range(1,len(s_data)):
    d2 = s_data[i]
    sound_buffer = combineAudio(sound_buffer, d2)

sa.play_buffer(sound_buffer, num_channels=1, bytes_per_sample=2, sample_rate=16_000)
```

Figure 16: Modified tdklatt

Finally, with a list of sound chunks, we wanted to combine it all into a single sound file. Since we did not want to simply list each sound right next to each other, we interpolated sound chunks that were right next to each other. At the end, we used Simpleaudio to play a test version of the sound.

```
with wave.open("test4.wav", "wb") as out_f:
    out_f.setnchannels(1)
    out_f.setsampwidth(2) # number of bytes
    out_f.setframerate(16000)
    out_f.writeframesraw(sound_buffer)
```

Figure 17: Test Version Code

## 5.0 KlattGrid Synthesizer

The KlattGrid synthesizer works in effectively the same way as the Klatt synthesizer. The key difference is that it uses a graphical user interface to input values. We did not do this in our example, partially due to the different API (Application Programming Interface), but also because of another difference. The KlattGrid synthesizer does not rely on discrete data points for each time period. Instead, it works on a continuous set of data over time. Of course, in our example, we input

a discrete set of data points. This gives us the most accurate results. However, in theory, we can get away with using a lot less - the synthesizer will simply draw a line between sequential data points. The result of this difference is a more natural-sounding voice, as it does not contain as many sudden jumps in frequency or intensity. Previously, we showed how data points are input into a KlattGrid. Below is the Python code that uses Parselmouth to create the KlattGrid object, and then the code to save it.

```
klattgrid = call(sound, "Create KlattGrid...", "Name", 0.0, 1.40, 6,
1, 1, 6, 0, 0, 0)

call(klattgrid, "Save as text file",
"C:/Users/User/Documents/1_S23_Classes/Comp.
Linguistics/Scripts/test.KlattGrid")
```

Figure 18: Parselmouth for KlattGrid

After doing so, we could open the file in Praat and convert it to the sound. It should also be possible to automate that step in Parselmouth. However, it was not necessary for our purposes. During experimentation with KlattGrid, we were surprised to find that bandwidth had a dramatic effect on synthesis. After entering data for F0, F1-F5, duration, and intensity, the sound produced was nothing like a voice, instead monotone and unintelligible. Inputting bandwidth data, however, transformed the sound into speech.[4] We were previously unaware of the dynamic power that bandwidth gives to speech, and thus consider this is a benefit of grid approaches such as Klatt and KlattGrid that allow the user to experiment with the effects of speech components (Koffi & Petzold, 2022).

## 6.0 World Vocoder Synthesizer

The WORLD Vocoder Synthesizer is a modern method developed by a team led by Masanori Morise at the Nation Institute of Information and Communications Technology (NICT) in Japan in 2016 (Kawahara et al., 1999). Its main application is speech analysis, manipulation, and synthetization with very little processing time but a high degree of accuracy. Using algorithms to estimate F0, the harmonic spectral envelope, and the aperiodic spectral envelope, it is able to synthesize a very realistic-sounding voice in a short amount of time. The algorithm it uses to do this is based on a similar one called STRAIGHT (Kawahara et al., 1999). There are several variations of STRAIGHT, such as TANDEM-STRAIGHT (Kawahara et al., 1997). WORLD, however, uses a convolution of both the harmonic and aperiodic spectral envelopes for a more natural-sounding voice. We did not write any code for WORLD. Instead, we used PyWORLD and the included demo to see what sort of results we would obtain. We are including this in the paper as a demonstration that formant-based synthesis is not the only method – or, indeed, the best. It is also useful for comparing our results to what modern methods can achieve.

[4] Editor's comment: This is the reason why bandwidth data has to be produced in order for speech synthesis to work.

## 7.0 Comparison

Below, we have listed each of the results obtained from our methods.

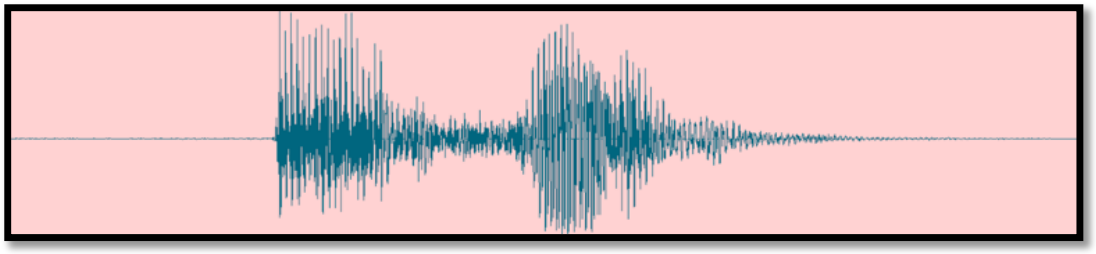


Figure 19a: Waveform of the original sound sample.

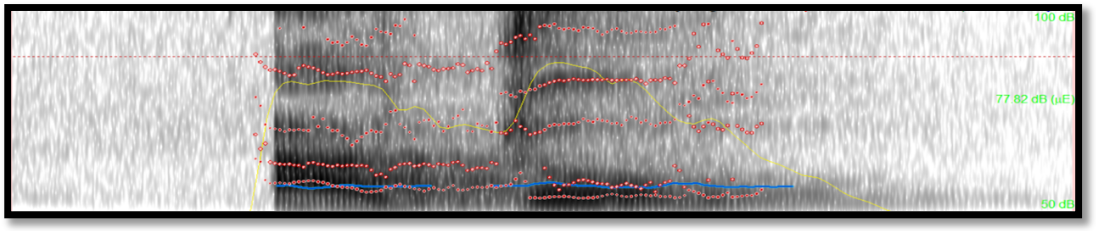


Figure 19b: Spectrogram of original sound sample

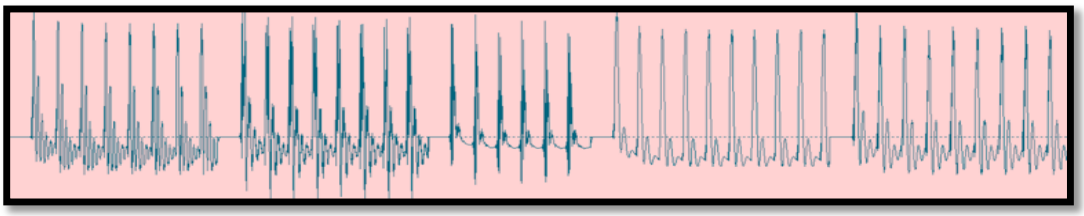


Figure 20a: Waveform of Klatt Synthesis with 5 chunks

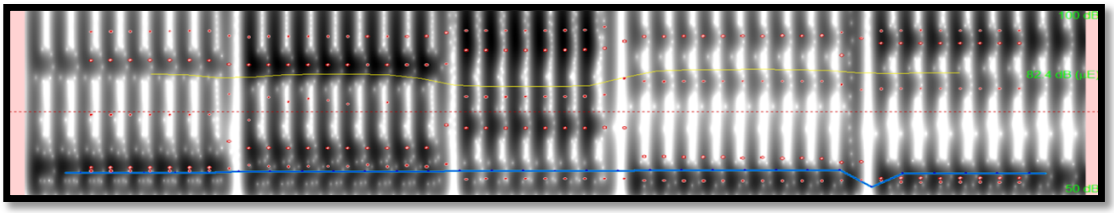


Figure 20b: Spectrogram of Klatt synthesis with 5 chunks.

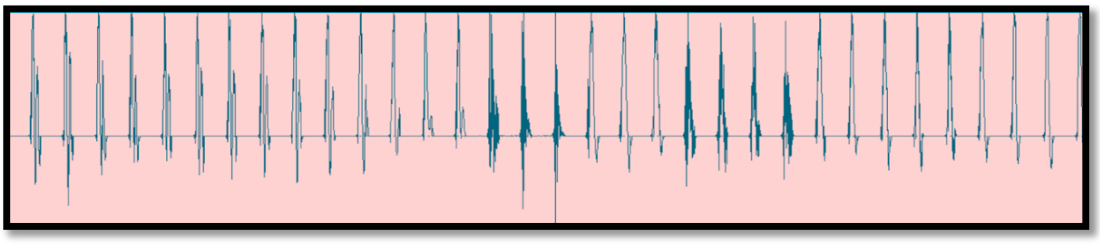


Figure 21a: Waveform of Klatt synthesis with 33 chunks.

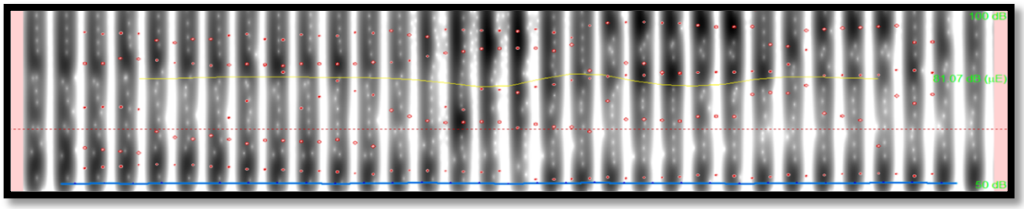


Figure 21b: Spectrogram of Klatt Synthesis with 33 chunks.

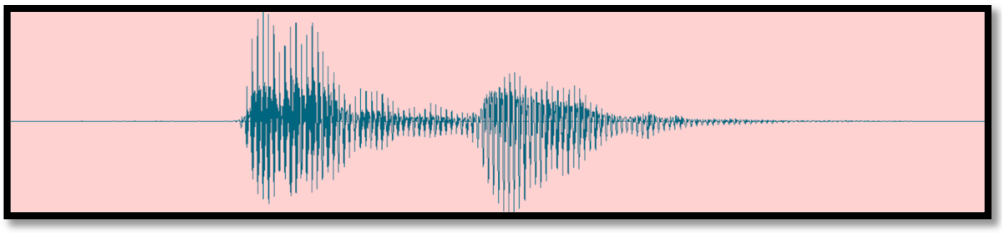


Figure 22a: Waveform of KlattGrid Synthesis.

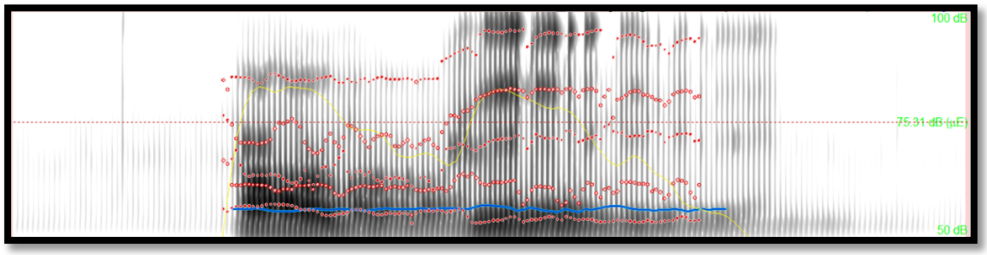



Figure 22b: Spectrogram of KlattGrid Synthesis.

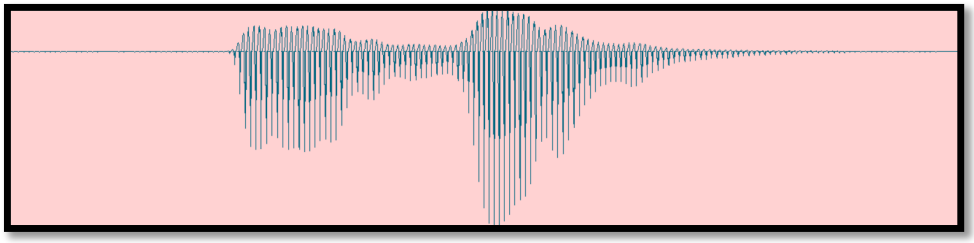


Figure 23a: Waveform of KlattGrid Synthesis without Bandwidths

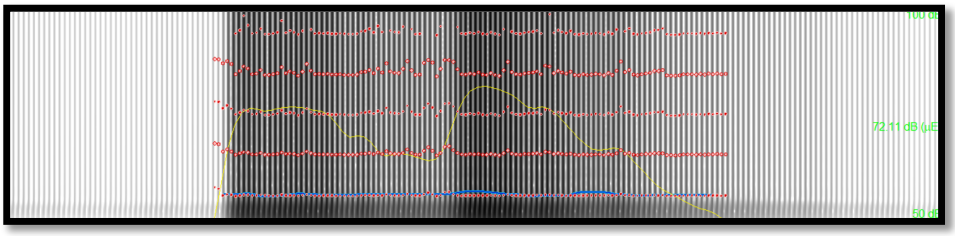



Figure 23b: Spectrogram of KlattGrid synthesis without Bandwidths

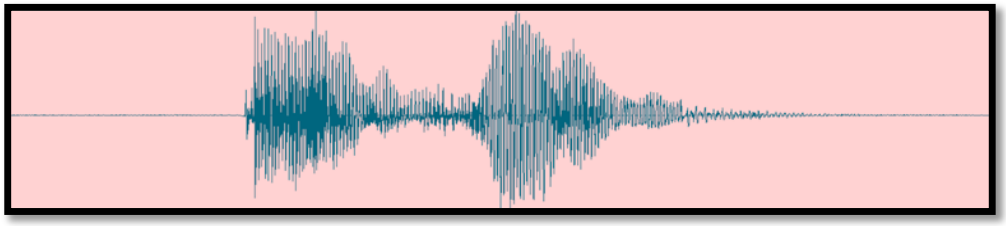


Figure 24a: Waveform of WORLD synthesis.

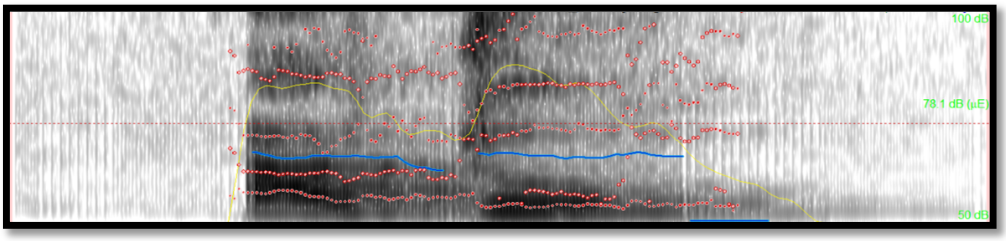



Figure 24b: Spectrogram of WORLD synthesis.

Based on these results, the most accurate synthesizer that sounded closest to the original audio sample was WORLD. In both the spectrographs and waveforms (Figures 19a and 24b), WORLD produced a speech sample that compared favorably to the original audio. As can be seen, our Klatt sample with five chunks resulted in an extremely choppy spectrograph whereas our Klatt sample with 33 chunks produced a more fluid-looking spectrograph and waveform. However, in both, a lot of information regarding intensity is missing. Neither of them looks at all natural when compared to the original sample. With more development, our Klatt sample could replicate the original speaker's voice to an extent, but it would be difficult to make it indistinguishable from her real voice. This only poses a problem if the ultimate goal is obtaining a synthesized voice that matches human speakers. For the purposes of Betine speech synthesis, an automated-sounding voice is not an issue for native speakers. However, given that this population is rapidly decreasing, in order for people in the future to be able to listen and interact with a synthesizer, we want the speech to be as close to a naturally-sounding Betine voice as possible. The WORLD synthesizer holds the most promise in this area.

KlattGrid produced speech closer to the original sample, but as seen in the spectrograph, much of the noise data is different. The formants are remarkably similar, which is a win for formant-based synthesis. Our formants produced in Klatt have little variation comparatively. We are optimistic that with a larger sample size, Klatt may produce a better replication of the formants.

When speech synthesis results from using Klatt, KlattGrid, and WORLD are compared, we notice that the latter has far better results. The noise gap between [à:] and [ɟ] that was mentioned early in the paper is also replicated faithfully in WORLD, suggesting that its synthesis has one of the most effective methods of filtering audio. It relies on more than formants to create the most natural sound. It is worth noting that WORLD relies more heavily on obtaining measurements from a sample sound. As a result, it is likely more difficult to synthesize new sounds based on a collection of other sounds. It is possible that machine learning could be used to find and extract patterns from sounds, but those patterns may not be understandable to human minds. Klatt, on the other hand, has relatively intuitive parameters for its synthesis. This being said, if WORLD can be harnessed to replicate a voice bank and rearrange phonemes in an effective way, which it seems to

possess, we believe that using vocoder synthesizers may be a viable route in the future of artificial speech synthesis.

## 8.0 Conclusion

We are hopeful that the work done in this course can bring technology one step closer to help dying languages. It can also breathe new life into disappearing cultures and people groups. Our research and experience have shown us that there is a wealth of knowledge and programming available, but the process of channeling it into a relevant program that can be applied is difficult and in need of much more research. The program developed by Klatt paved the way for much of what speech synthesis has accomplished in the last fifty years, but there are new synthesizers and programs constantly being developed that might bring the lightbulb solution for synthesizing dying languages. In the process of working on this project, we have gained a greater appreciation and understanding of the massive scope of this field. We have only just barely touched the surface on what is possible and what the future holds.

## ABOUT THE AUTHORS

**Clare Miller** (she/her) is an undergraduate student at Saint Cloud State University, majoring in Computer Science. She has experience with Lua, C++, and Java. This was her first large project using Python. Until recently, she has had little formal education on the topic of acoustic phonetics. She is interested in ways speech analysis and machine learning can be used to assist with language learning, particularly phoneme identification. She can be reached at crmiller@go.stcloudstate.edu or miller.clare@outlook.com.

**Isabella Randolph** is an undergraduate student at Saint Cloud State University, majoring in Linguistics with an emphasis in Communication Sciences and Disorders (CSD). She has focused on the acoustic phonetics area of linguistics in her studies and is currently involved in research with acoustic analyses of English as a Foreign Language (EFL) speech. She plans to pursue research at a graduate level in speech pathology. She can be reached at Randolph, Isabella J isabella.randolph@go.stcloudstate.edu or isabella.randolph21@gmail.com.

## References

Boesma, P. (1993): "Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound." *Proceedings of the Institute of Phonetic Sciences 17:97-110.* University of Amsterdam. Available on http://www.fon.hum.uva.nl/paul/.

Guest, D. (2018). tdklatt, Github. Accessed: 4/26/2023. [Online]. Available: https://github.com/guestdaniel/tdklatt.

Hsu, J. (2016). PyWORLD, Github. Accessed: 4/26/2023. [Online]. Available: https://github.com/JeremyCCHsu/Python-Wrapper-for-World-Vocoder.

Kawahara, H., Masuda-Katsuse, I., and de Cheveigné, A. (1999). “Restructuring speech representations using a pitch-adaptive time–frequency smoothing and an instantaneous-frequency-based F0 extraction: Possible role of a repetitive structure in sounds,” *Speech Communication*, vol. 27, no. 3-4, pp. 187–207.

Kawahara, H., Estill, J., and Fujimura, O. (1997). "Speech representation and transformation using adaptive interpolation of weighted spectrum: vocoder revisiter," in Proc ICASSP1997, pp.1303-1306.

Klatt, D. H. (1980). “Software for a cascade/parallel formant synthesizer,” *The Journal of the Acoustical Society of America*, vol. 67, no. 3, pp. 971–995.

Koffi, E. (2021). “Language endangerment threatens Phonetic diversity,” *Acoustics Today*, vol. 17, no. 2, p. 23.

Koffi, E. (April 2023). Personal Communication [In-person].

Koffi, E. and Petzold, M. (2022). "A Tutorial on Formant-based Speech Synthesis for the Documentation of Critically Endangered Languages," *Linguistic Portfolios*: Vol. 11, Article 3.

Morise, M. (2014). “Cheaptrick, a spectral envelope estimator for high-quality speech synthesis,” *Speech Communication*, vol. 67, pp. 1–7.

Morise, M., Kawahara, H., and Katayose, H. (2009). "Fast and reliable F0 estimation method based on the period extraction of vocal fold vibration of singing voice and speech," *Journal of the Audio Engineering Society*.

Morise, M. (2017). Harvest: A high-performance fundamental frequency estimator from speech signals, in Proc. INTERSPEECH 2017, pp. 2321–2325, 2017. http://www.isca-speech.org/archive/Interspeech_2017/abstracts/0068.html.

Morise, M. (2012). “Platinum: A method to extract excitation signals for voice synthesis system,” *Acoustical Science and Technology*, vol. 33, no. 2, pp. 123–125, Oct. 2012.

Morise, M. (2016). D4C: a band-aperiodicity estimator for high-quality speech synthesis, Speech Communication, vol. 84, pp. 57-65, Nov. 2016. http://www.sciencedirect.com/science/article/pii/S0167639316300413

Morise, M., Yokomori, F., and Ozawa, K.: WORLD: a vocoder-based high-quality speech synthesis system for real-time applications, IEICE transactions on information and systems, vol. E99-D, no. 7, pp. 1877-1884, 2016. https://www.jstage.jst.go.jp/article/transinf/E99.D/7/E99.D_2015EDP7457/_article

Petzold, M. (2023). Personal Communication [In-person].