VRIJE
UNIVERSITEIT
BRUSSEL

SELF-ORGANIZATION IN VOWEL SYSTEMS FROM SMALL COMMUNITIES

Extending de Boer 2000 - EoS project

Lennert Bontinck

January, 2021-2022

Student number: 0568702

Computer Science: AI

Contents

1 Project goal and supplied code **1**

1.1 Project goal . . . 1

1.2 Important files . . . 2

2 Relevant literature **3**

2.1 Summary of de Boer (2000) . . . 3

2.2 Importance of network structure for emergence . . . 3

2.3 Other required background information . . . 3

3 Re-implementing de Boer (2000) **4**

3.1 Producing sounds . . . 4

3.2 Perceiving sounds . . . 5

3.3 Representing agents . . . 5

3.4 Playing and analysing games . . . 7

3.5 Improving de Boer (2000) . . . 7

4 Testing emergence for a small community **9**

4.1 TODO . . . 9

5 Results **10**

5.1 TODO . . . 10

6 Discussion **11**

6.1 TODO . . . 11

Extra figures **12**

References **13**

Project goal and supplied code

During the Evolution of Speech course taught at the VUB in 2021-2022 we, Computer Science students, were introduced to this multidisciplinary field by reviewing multiple important papers of the field. As the course was taught by Bart de Boer, who has an Oxford University Press published book on the origins of vowel systems and many papers in the field, we also reviewed some of de Boer's papers (de Boer, 2001; de Boer, 2000; de Boer and Thompson, 2018; de Boer and Zuidema, 2010). As a Computer Science student with limited linguistic knowledge, papers from de Boer using Agent-Based Modelling (ABM) techniques for studying phenomena in the field were found the most interesting (de Boer, 2000; de Boer and Zuidema, 2010). Because of this, we opted to extend upon de Boer (2000). The exact project goal and an overview of the supplied code are given in further detail here.

1.1 Project goal

As the ABM related papers were found most interesting, it was chosen to re-implement and extend the paper on self-organization in vowel systems by de Boer (2000) for this project. Whilst the original C++ source code of de Boer (2000) was provided to us, his students, it was dated and not so well documented. This was to be expected as the code was not originally meant for distribution. To further ground our understanding of the paper and make extensions on this work easier, we have chosen to do a re-implementation in Python. The written code is well documented, easily extendable and most importantly, publicly available under the GPL V3 license (Bontinck, 2021; "GNU General Public License", 2007). This enables readers to not only easily reproduce the results of de Boer (2000) and the extensions provided here but also gives them a great basis for future projects. The latter was something we felt was lacking and feel is an important contribution of this work. We also addressed some of the *ad hoc* decisions in the original version. To be more precise, this report provides an alternative way of converting to the bark scale and determining the effective second formant of a produced sound. This was found to not influence the results, as is further discussed in 3.5.

As computer science students, we know how important the used network structure is to models and ABMs in specific. In the original version of the imitation game proposed by de Boer (2000), agent pairs are picked at random. Whilst this is an understandable simplification for his work, it made us wonder if the findings hold for more complex structures. Initially, it was considered to use scale-free networks, as we thought this would better represent a human network. However, the actual realism of scale-free networks is debated and one of our colleagues wanted to go this route already (Broido & Clauset, 2019). Because of this, we opted to model a small community consisting of agents with different roles and influences. These agents also die and get replaced by new agents. It is thus more dynamic and varying than the setting used by de Boer (2000) and will test the original hypothesis of his paper further. This hypothesis is: "The structure of vowel systems is determined by self-organization in a population under constraints of perception and production." - de Boer (2000).

1.2 Important files

Accompanied by this report is a copy of the GitHub repository created for this project (Bontinck, 2021). It includes all files needed to reproduce the experiments, including saved versions of the games used for figures and statistics in this report. An overview of the most important files is given below:

- `README.md`
  - General information of the GitHub repository with hyperlinks to important files and documentation.
- `code-output`
  - All figures generated by the provided code, some of which are used in this report.
- `code/notebooks`
  - Jupyter notebooks and plain py files going over different aspects of the code for this project.
  - `1_implementing_de_boer_2000.ipynb`: step by step re-implementation of code by de Boer (2000).
  - `imitationGameClasses.py`: all classes needed to play imitation games as specified by de Boer (2000).
  - `2_recreating_de_boer_2000.ipynb`: step by step re-collection of results by de Boer (2000).
  - `3_alternative_bark_experiments.ipynb`: step by step re-collection of results by de Boer (2000) using a less ad hoc variant of the bark converter and effective second formant weighting function.
  - TODO TODO TODO TODO TODO TODO TODO TODO
- `code/html-exports` and `documentation/installation`
  - HTML export of the above discussed Jupyter notebooks, ideal for those who want to view the notebooks without installing the Anaconda environment.
  - Install instructions for the used Anaconda environment of this project (macOS and Ubuntu).

Relevant literature

TODO

2.1 Summary of de Boer (2000)

TODO

2.2 Importance of network structure for emergence

TODO

2.3 Other required background information

TODO

Re-implementing de Boer (2000)

The original C++ source code of de Boer (2000) was provided to us, de Boer's students. However, this code was dated and not meant for distribution, which made it difficult to be used or extended upon. Because we saw the value in a well documented and easy to extend implementation of this project, we decided to fully re-implement it in Python. This was done incrementally, with each step described in `1_implementing_de_boer_2000.ipynb`. The working of the code was validated by reproducing the results of the experiments by de Boer (2000) in `2_recreating_de_boer_2000.ipynb`. This chapter will summarise the development and findings of these two Jupyter notebooks. We stress that all of the discussed code in what follows is either derived from textual description in de Boer (2000) or the provided C++ code.

3.1 Producing sounds

de Boer (2000) system consists of agents who can produce, perceive, and remember speech sounds in a human-like way. The most important component for producing sounds in a human-like manner is the articulatory synthesizer. This synthesizer takes as input three vowel parameters: the tongue position (p), the tongue height (h) and the lip rounding (r). The outputs of the synthesizer are the first four formant frequencies of the corresponding vowel: F_1 to F_4 expressed in Hz. The conversion between input and output happens based on the synthesizer equations given by de Boer (2000), table 2. De Boer (2000) used interpolation from known data to create these equations. We used these known points for validating our F_1 to F_4 conversions from the p, h and r input and found them to match perfectly with the given data.

From the above-described synthesizer equations, we can make the `Synthesizer` class. To represent input and output formally, we make use of two helper classes: `Utterance` and `Phoneme`. The former stores an utterance consisting of the four formants F_1 to F_4. This is the output of the `synthesise` function. A phoneme is used as input for the same function and stores the p, h and r parameters. Two types of noise can be assigned to a `Synthesizer` object. The `max_noise_agent` and the `max_noise_ambient`. The agent noise is applied to the phoneme before utterance creation (Equation 3.1), the ambient noise is applied to the utterance after creation. The applied noise, λ, is uniformly picked from: $\frac{-\psi}{2} \leq \lambda \leq \frac{\psi}{2}$, with ψ being the provided parameter. ψ is one of many important parameters. In the experiments by de Boer (2000), only the ambient noise is used.

$$F_i^{agent}(p, h, r) = F_i(p + \lambda, h + \lambda, r + \lambda) \tag{3.1}$$

$$F_i^{ambient} = F_i * (1 + \lambda) \tag{3.2}$$

3.2 Perceiving sounds

With the `Synthesizer` and helper classes in place, an agent can produce a signal that represents sound in a human-like manner. For an agent to perceive these signals in a human-like manner, the `Bark Operator` class is created. This class is responsible for working with utterances. Remember that utterances were the first four formants of a generated sound, in Hertz. As the name of this class implies, the Bark scale is used by this class and is differing from the previously used Hertz scale. It represents frequencies in a manner that is closer to human perception. It goes from the four formant representation in Hertz to a two formant representation in Bark consisting of the first formant and the *effective second-formant* (F_2').

We took the conversion formulae and calculation formula for the effective second formant straight from de Boer (2000). The conversion from Hertz to Bark and back used by de Boer (2000) is again interpolated from data. It is also admitted by de Boer (2000) that his calculations for determining (F_2') are a bit ad hoc. The critical distance used for calculating (F_2') can be provided as an optional argument. Because of the interpolated conversion and ad hoc (F_2') calculations, we have also foreseen an `alternative_bark_conversion` parameter. If set to `True`, the Bark operator will use alternative methods for both of these functions. This is further discussed in section 3.5.

Having our Bark space configured, we can implement a distance measure between utterances as specified in Equation 3.3. This distance measure can be used as a way for the agent to determine the closest sound in his repertoire to the one it heard. This must happen in the Bark space as equal distances in bark correspond to roughly equal human-perceptual distances of sound. This is not the case for Hertz, as humans have a harder time differentiating higher frequencies. With this human-like distance measure, agents can now perceive sounds and compare them with their known sounds. This λ is again an important parameter for the simulations. It is set to 0.3 for all experiments in the project unless specified differently. This value is seen as realistic by de Boer (2000), Ladefoged (1985), Schwartz et al. (1997), and Vallée (1994).

$$D = \sqrt{(F_1^a - F_1^b)^2 + \lambda(F_2'^a - F_2'^b)^2} \tag{3.3}$$

3.3 Representing agents

The `Agent` class makes use of all previously discussed classes as well as the `Sound` helper class. The `Agent` class takes a synthesizer and Bark operator as arguments for initialising. It also has over ten optional parameters one of which is a logging capability handy for debugging purposes. The `Sound` class is used to store a known sound of an agent. It consists of the phoneme, utterance, usage count and success count of the sound. The utterance of this sound is determined by synthesising the provided phoneme in a noiseless environment.

To discuss the functions of an agent, it is easiest to present a typical imitation game flow. Algorithm 1 shows the actions performed by a randomly picked agent who starts an imitation game. If the agent known sound repertoire is empty a completely random vowel is inserted by picking random values between 0 and 1 for the Phoneme parameters. A different randomly picked agent plays the role of imitator and response to the heard utterance using the process shown in Algorithm 2. If the imitator's known sounds repertoire is empty, it will add a *similar sound* to the one it heard. It does this by checking eight *corner* sounds it can produce and picking the one which is closest in distance. Afterwards it improves this sound further by using its `improve_sound` function for the agent specific amount of times (`max_similar_sound_loops` parameter). The `improve_sound` function tries all possible permutation's of the phoneme parameters by either keeping the value or adding/substracting the agent specific step size (`phoneme_step_size` parameter).

Algorithm 1 The say_something function of an imitation game initiator agent

```
if No known sounds then
    Add random sound to known sounds
end if
S ← random known sound
Update usage count of S
Remember chose of S
Return utterance of S using own bark operator
```

Algorithm 2 The imitate_sound function of an imitator

```
Require: U_in: the heard utterance
if No known sounds then
    Add similar sound for U_in to known sounds
end if
Remember U_in
Find closest known sounds S to heard utterance
Update usage count of S
Remember chose of S
Return utterance of S using own bark operator
```

In the second phase of the game, the initiator validates the imitation it hears in a non-verbal manner. This process is shown in Algorithm 3. The agent validates if the closest known sound to the heard imitation utterance is the sound he used to start the game. He also communicates this to the imitating agent in a non-verbal manner. He updates the success count accordingly and prepares himself for the next round. The process of preparing for a new round is given in Algorithm 4. This consists of resetting the game variables such as the *last_spoken_sound* variable. The agent then updates its count of games played, ads well as the success and imitator/initiator count respectively. Based on the agent specific `cleanup_prob`, `new_sound_prob` and `merge_prob` the agent will potentially remove bad sounds, add a semi-random new sound or merge similar sounds. A sound is thus removed periodically if it's success rate is below the agent specific `sound_threshold_agent` and used at least `sound_minimum_tries`, which is also agent specific. A sound is also added semi-randomly on a periodic basis. We call this process semi-random as multiple random vowels will be tried based on the agent specific `max_semi_random_loop`, and the sound that had the greatest summed distance will be used as new sound. Finally, similar sounds are also merged on a periodic basis. The agent does this by validating if both the utterance or phonemes don't lie too close. Phonemes lie too close if their parameters differ less then 0.17 in total. Utterance are too close if they can't be distinguished taking into account the noise of the environment. Both of these calculations are taken from the code provided by de Boer (2000).

Algorithm 3 The validate_imitation function of an initiator

Require: U_{in}: the heard imitation utterance
  Retrieve last spoken sound S
  Find closest known sounds S' to heard utterance
  $success \leftarrow S = S'$?
  if *success* **then**
    Update success count of S
  end if
  Prepare for new game
  Return *success*

Algorithm 4 The prepare_for_new_game function of an agent

Require: *imitator*: whether or not the agent was an imitator in the played game
Require: *success*: whether or not the played game was a success
  Update agent games count
  if *success* **then**
    Update agent success count
  end if
  if *imitator* **then**
    Update agent imitator count
  else
    Update agent initiator count
  end if
  Remove bad sounds per agent specific odd
  Merge similar sounds per agent specific odd
  Add semi-random sound per agent specific odd
  Reset game variables

To end a game cycle, the imitator agent will process the non-verbal imitation success communication. It does using the process described in Algorithm 5. If the imitation was successful the agent will use the previously described `improve_sound` function once to make the spoken sound better match the heard utterance. If the imitation was not successful and the used sound has a success ratio lower then the agent specific `sound_threshold_game`, the sound is also improved as described before. However, if the success ratio of the sound is above this threshold it is assumed that the spoken sound is a correct imitation of other sounds in the network and thus a similar sound is added to the one heard as reaction. The process of adding this similar sound is identical as described when the sound repertoire of an imitator was empty.

3.4 Playing and analysing games

TODO

3.5 Improving de Boer (2000)

TODO

Algorithm 5 The process_non_verbal_imitation_confirmation function of an imitator

---

Require: *success*: whether or not the imitation was a success
  Update agent games count
  if *success* **then**
    Update agent success count
    Improve used sound to better match heard utterance
  else if Low success ratio of spoken sound **then**
    Improve used sound to better match heard utterance
  else
    Add a similar sound to the heard utterance
  end if
  Remove bad sounds per agent specific odd
  Merge similar sounds per agent specific odd
  Add semi-random sound per agent specific odd
  Reset game variables

Testing emergence for a small community

TODO

4.1 TODO

TODO

Results

TODO

5.1 TODO

TODO

Discussion

TODO

6.1 TODO

TODO

Extra figures

To make the report more readable some figures are not provided directly in the text. These figures are provided here.

References

Bontinck, L. (2021). *Eos project @ vub 2021 - 2022* [GitHub commit: TODO...]. Retrieved December 27, 2021, from https://github.com/pikawika/eos

Broido, A. D., & Clauset, A. (2019). Scale-free networks are rare. *Nature Communications*, *10*(1). https://doi.org/10.1038/s41467-019-08746-5

de Boer, B. (2001). *The origins of vowel systems.* [Oxford: Oxford University Press The book based on my PhD. thesis.]. Oxford University Press.

De Smet, R. (2020). *Vub latex huisstijl* [GitHub commit: d91f55...]. Retrieved November 2, 2020, from https://gitlab.com/rubdos/texlive-vub

de Boer, B. (2000). Self-organization in vowel systems. *Journal of Phonetics*, *28*(4), 441–465. https://doi.org/10.1006/jpho.2000.0125

de Boer, B., & Steels, L. (1999). Self-organization in vowel systems. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.32.5947&rep=rep1&type=pdf

de Boer, B., & Thompson, B. (2018). Biology-culture co-evolution in finite populations. *Scientific Reports*, *8*(1). https://doi.org/10.1038/s41598-017-18928-0

de Boer, B., & Zuidema, W. (2010). Multi-agent simulations of the evolution of combinatorial phonology. *Adaptive Behavior*, *18*(2), 141–154. https://doi.org/10.1177/1059712309345789

Gnu general public license. (2007, June 29). *Free Software Foundation.* http://www.gnu.org/licenses/gpl.html

Ladefoged, P. (1985). *Phonetic linguistics : Essays in honor of peter ladefoged.* Academic Press.

Schwartz, J.-L., Boë, L.-J., Vallée, N., & Abry, C. (1997). The dispersion-focalization theory of vowel systems. *Journal of Phonetics*, *25*(3), 255–286. https://doi.org/10.1006/jpho.1997.0043

Vallée, N. (1994). *Systèmes vocaliques : De la typologie aux prédictions* (Doctoral dissertation). Université Stendhal.