Behavior Research Methods
https://doi.org/10.3758/s13428-017-1012-5

TISK 1.0: An easy-to-use Python implementation of the time-invariant string kernel model of spoken word recognition

Heejo You[1,2] · James S. Magnuson[1]

Abstract
This article describes a new Python distribution of TISK, the *time-invariant string kernel* model of spoken word recognition (Hannagan et al. in *Frontiers in Psychology*, *4*, 563, 2013). TISK is an interactive-activation model similar to the TRACE model (McClelland & Elman in *Cognitive Psychology*, *18*, 1–86, 1986), but TISK replaces most of TRACE's reduplicated, time-specific nodes with theoretically motivated time-invariant, open-diphone nodes. We discuss the utility of computational models as theory development tools, the relative merits of TISK as compared to other models, and the ways in which researchers might use this implementation to guide their own research and theory development. We describe a TISK model that includes features that facilitate in-line graphing of simulation results, integration with standard Python data formats, and graph and data export. The distribution can be downloaded from https://github.com/maglab-uconn/TISK1.0.

Keywords Spoken word recognition · Neural networks · Computational models

In this article, we introduce an easy-to-use, freely available, and extensible implementation of the *time-invariant string kernel* (TISK) model of spoken word recognition (Hannagan, Magnuson, & Grainger, 2013). Our motivations in making this model available follow from our strong belief that models are essential tools for developing and testing theories of even moderate complexity, and our strong commitment to making models freely available to promote the replication of simulations, the comparison of models, and (therefore) the exploration of model predictions. We briefly discuss these motivations in the following two sections, and then we turn to the details of the TISK model and of the freely available distribution of the TISK code described below.

The utility of computational models in the brain and cognitive sciences

As Farrell and Lewandowsky (2010) have discussed, replication in scientific reasoning is a crucial and often overlooked aspect of the scientific process. They suggest that when theories are specified verbally, scientists risk engaging in a game of "telephone," in which different scientists may construe the same description of a theory in qualitatively different ways, via disparate mental models. They argue that formal theory specifications, such as implemented simulation models, promote replicability of scientific reasoning; rather than intuiting behavior implied by a theory, it can be observed via simulation, potentially confirming or disconfirming intuitive predictions or producing unexpected behavior. Furthermore, they point out that implementing a theory requires a high level of precision regarding not just theoretical assumptions, but concrete details such as rates of learning or activation spread. Note that a theory with just a few "moving parts" (e.g., forward activation flow between levels and lateral inhibition within levels) quickly becomes "intuitively intractable" (i.e., one cannot predict its behavior beyond a small number of steps), or even analytically intractable (i.e., depending on the dynamics assumed or implied, a model's category- or item-specific behavior may not be derivable by equation). Magnuson, Mirman, and Harris (2012) have discussed examples from the spoken word recognition literature in which seemingly logical predictions about what a model would do were not confirmed when simulations were actually conducted. Thus, the behavior implied by a theory may often be derivable *only* by simulation, making models *essential* tools for theory development and theory testing.

✉ James S. Magnuson
james.magnuson@uconn.edu

[1] Department of Psychological Sciences and Connecticut Institute for the Brain and Cognitive Sciences, University of Connecticut, Storrs, CT, USA

[2] Korea University, Seoul, Korea

Published online: 30 April 2018

There are also pitfalls that come with implementing models, however. Magnuson et al. (2012) pointed out that implementing a model also requires grappling with details that may be outside the scope of the theory, including simplifying assumptions. For example, in the domain of speech and spoken word recognition (SWR), most current models do not take real speech as input. Instead, a *temporary* simplifying assumption is accepted: Rather than waiting for all problems at the level of the speech signal to be solved, assume the input is the output of a preprocessing stage corresponding to a string of abstract phonemic or phonetic features extracted from speech. As Magnuson (2008) discussed, such simplifying assumptions can paradoxically *complicate* the problem under study (e.g., a focus on phonemic inputs hides potential subphonemic constraints on spoken word recognition, as was exquisitely demonstrated by Salverda, Dahan, & McQueen, 2003) while eventually acquiring a functional status in theoretical reasoning similar to that of the true theoretical commitments, highlighting some questions (e.g., the *embedding problem* in SWR—that most words have multiple words embedded within them, such as CAT in CATALOG; McQueen, Cutler, Briscoe, & Norris, 1995), and masking some others (e.g., sub- and supraphonemic cues to word length that potentially mitigate embedding; Salverda et al., 2003; see also Davis, Marslen-Wilson, & Gaskell, 2002).

Even when the simplifying assumptions are clearly identified as such, they pose challenges for model interpretation, even (or potentially especially) when they appear to be outside the scope of a theory (Lewandowsky, 1993). Magnuson et al. (2012) discussed four decreasingly interesting levels at which models can fail (or succeed): theory (core predictions, independent of details of implementation), implementation (aspects of the architecture, such as representation of the input, output, or model-internal levels), parameters (specific values governing, e.g., the strength of connections between or within levels), and linking hypotheses (operational definitions relating model and human behavior to stimulus properties).

Ideally, experimental simulations have the potential to identify instances where a core *theoretical* commitment is falsified (e.g., [hypothetically] a demonstration that feedback or lateral inhibition in an interactive-activation model generates predictions that run counter to human performance).

Implementational details (choices of representations and model architecture) are a crucial aspect of translating a theory to a model, but failures (or successes) attributable to this level cannot falsify a theory (unless all reasonable implementational choices have been exhausted). For example, evidence for human sensitivity to subphonemic coarticulatory detail does not falsify a theory when its corresponding model has been implemented with phonemic-grain inputs; rather, this demonstrates a limit of a particular simplifying assumption.

Failures (or successes) due to *parameter* choices (e.g., balance of bottom-up vs. top-down gain) provide weak evidence at best against a theory. Demonstrating poor model fit with a particular set of parameters is virtually meaningless without some exploration of the parameter space (see Pitt, Kim, Navarro, & Myung, 2006, for *parameter space partitioning*, a formal approach to model comparison). Conversely, a model success that can only be observed with a very precise set of parameters must be treated with caution. Robust tests of parameter dependence require both exploration of the parameter space and exploration of the attested performance space. That is, models should be tested on a broad set of attested phenomena in the biological domain they apply to. It is particularly problematic when model parameters are adjusted to simulate a new aspect of performance without simulations confirming that the model can still simulate other phenomena to which it was applied with the previous parameter settings.

Finally, model failures due to poor linking hypotheses are not even wrong; they are simply invalid. Magnuson et al. (2012) reviewed examples from SWR in which simulations were invalidated by the use of model input manipulations that were poor analogues to the materials used with human subjects.

Exploring any of these levels requires implemented models. Again, crucially, precise predictions that follow from a theory of even moderate complexity can in many cases only be derived from simulation with a model. This highlights the pressing need for freely available, reasonably easy-to-use implementations of models that implement competing theories.

As we have just discussed, the process of implementing models to accompany theories pushes scientists to a higher level of precision, requiring them to separate theoretical commitments from convenient simplifications, while grappling with details that might never have been considered if the theory had not moved beyond verbal specification (Farrell & Lewandowsky, 2010; Lewandowsky, 1993; Magnuson et al., 2012). Ideally, new empirical findings, new theories or extensions of extant theories would be accompanied by simulations (ideally with multiple competing models corresponding to competing theories). Replication is also an important concern, not just with respect to confirming reported simulations, but also with respect to scientific reasoning; that is, understanding how and why a model makes the predictions it does (Farrell & Lewandowsky, 2010). Observing model simulations is a crucial way to gain understanding of a model, and therefore its corresponding theory (though see the caveats discussed above regarding levels of model success and failure). It is not reasonable, however, to expect every scientist to have the skills, time, and financial resources required to implement every (or even any) model they wish to test or simulate on a new empirical finding. Indeed, despite the complexity of our theories, simulations are exceedingly rare throughout the brain and cognitive sciences, and in the domain of SWR in particular.

One partial remedy to this situation is to make models freely available in formats easy enough for programming novices to use (requiring that nonprogrammers acquire minimal programming fundamentals to work with models). In SWR, only a few key models are available. The original C code for

the TRACE model (McClelland & Elman, 1986) has been available from McClelland's lab for decades. A GUI-based Java reimplementation of TRACE (no programming required) that includes facilities for graphing and batch processing was described a decade ago in this journal (Strauss, Harris, & Magnuson, 2007). Shortlist B (Norris and McQueen, 2008), the Bayesian refinement of Shortlist (Norris, 1994), is freely available on the web from the Norris lab. Other models may be or may have been available by request to authors, but we are aware of several models that are not available (due to author choice [based, e.g., on reluctance to distribute uncommented code, or even loss of code]). In this article, we report on a refined implementation of a recent model that we are making freely available in order to promote its use for replication, comparison with other models, and possible theory and model extension by other researchers.

The TISK model of spoken word recognition: A brief overview

Hannagan et al. (2013) introduced the *time-invariant string kernel* (TISK) model of spoken word recognition (SWR). Like TRACE (McClelland & Elman, 1986), the gold-standard for simulation models of SWR (cf. Magnuson et al., 2012), TISK is an interactive activation model. One of the great challenges in simulating SWR is representing sequences. A model that simply activated phonemes as they occurred would not have a way of representing the order in which they occurred; the phoneme series corresponding to CAT, TACK, and ACT would all result in /k/, /æ/, and /t/ being activated. Other strategies include creating templates that are sensitive to order by differentially activating items in sequences based on order (e.g., Grossberg & Kazerounian, 2011), but such schemes by themselves cannot encode items with repeated elements (e.g., /rili/ ["really"] could not be distinguished from /ril/ ["real"]). TRACE utilizes a unique strategy for encoding sequences of arbitrary length and with repeated elements: it reduplicates time-specific detectors for features, phonemes, and words in a spatial memory. The memory "trace" is aligned with the inputs such that at each processing step, a new input is applied to the "right" of the previous input (see Fig. 1). Phoneme detectors aligned with the inputs become activated, as do word detectors aligned with the appropriate phoneme nodes that have become activated. This allows TRACE to represent sequences, including sequences with repeated elements (e.g., the two instances of /d/ in /dæd/ ["dad"] would be encoded by independent /d/ detectors, as would the two instances of "dog" in "dog eat dog").

The reduplication strategy has often been critiqued, beginning with McClelland and Elman themselves (1986, p. 77). Hannagan et al. (2013) review some of the arguments, and Magnuson (2015) makes a case for TRACE as a model building

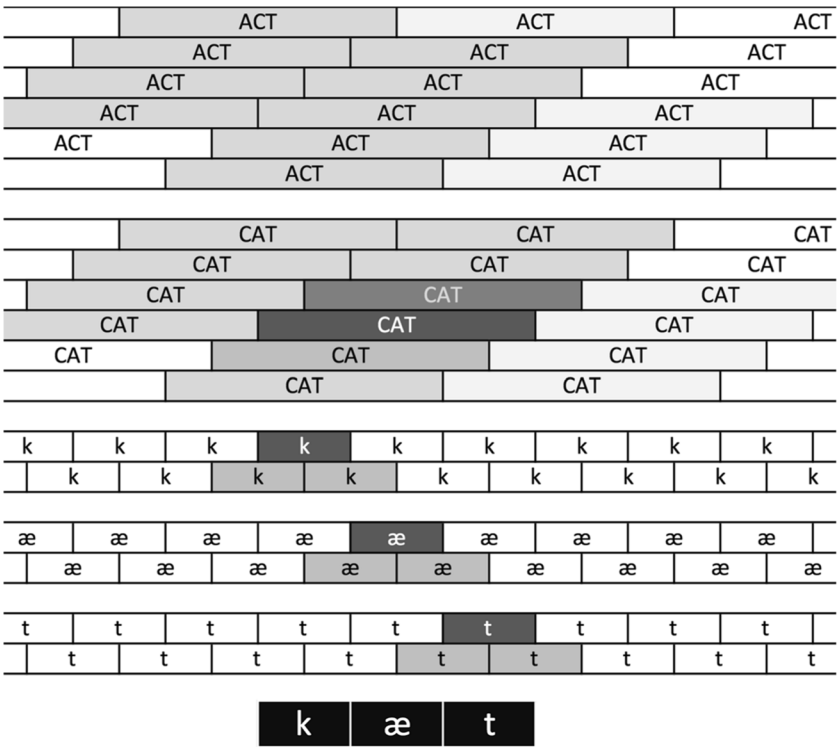

Fig. 1 TRACE's reduplication strategy (McClelland & Elman, 1986). The black squares at bottom indicate the input patterns (which would be features over time in the actual model), presented in sequence (corresponding to CAT). The nodes above are part of the "trace," aligned with specific time slices. Shading of the cells indicates their degrees of activation, with black indicating a high level and white a low level. The inputs activate phoneme detectors that align with them even partially, and the activated phoneme detectors activate the word detectors aligned with them.

on echoic memory. As Hannagan et al. (2013) concluded, no matter one's opinion on the plausibility of the TRACE reduplication strategy, its computational cost (scaling to a realistic phoneme and lexical inventory would require ~1.3 million nodes and more than 40 billion connections) raises the question of whether a more efficient solution might be possible. Hannagan et al. (2013) proposed a new model that replaced most of the time-specific nodes in TRACE with time-invariant nodes (single instances rather than reduplicated copies). The key innovation that allows for this (and that motivates TISK's name) is a kind of string kernel representation for sequences.

TISK uses a kind of "open-diphone" coding. To illustrate this, let us start with a full open, ordered diphone code. For simplicity, we will use orthographic bigrams rather than phonemic diphone examples. The idea is that all ordered pairs of letters that occur in a string are part of that string's representation, no matter the size of the gap between two letters. So for CAT, the bigram pairs are CA, CT, and AT. For DAD, they would be DA, DD, and AD. For CATALOG, they would be CA × 2, CT, CL, CO, CG, AT, AA, AL × 2, AO × 2, AG × 2, TA, TL, TO, TG, LO, LG, and OG. The kernel aspect of this is that we can implement the encoding as a matrix of all letters by all letters, and then represent any string as the count of each letter–letter pair that occurs in the string. Then the operations we wish to perform on the strings (e.g., comparing similarity between any pair of strings) can be performed on the string matrices, with any operation taking the same amount of time, independent of the original string length (see Hannagan et al., 2013, for more discussion).

Before discussing how TISK implements diphone coding, it will be helpful to review the structure of the model. TISK has four key sets of units that take activations and have dynamics over time (via, e.g., decay of activation), as is schematized in Fig. 2. (1) Phoneme inputs are time-specific, with a copy of every phoneme at every time step. The phoneme inputs map to appropriate (2) diphones and (3) single-phoneme nodes within the *n*-phone layer. (4) The *n*-phone layer units map to appropriate words. There is lateral inhibition among the word units. In the original Hannagan et al. (2013) model, which we call "TISK

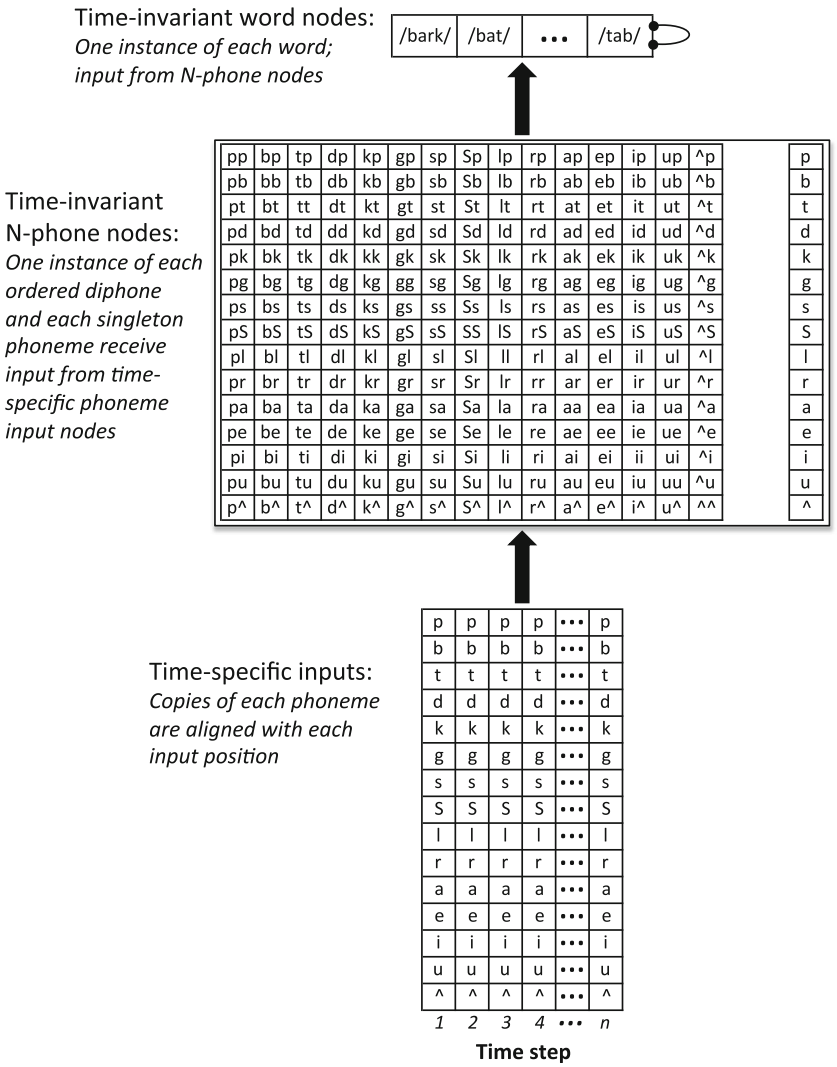

Fig. 2 The structure of TISK (from Figs. 3 and 4 of Hannagan et al., 2013).

1.0," word-to-*n*-phone feedback was not implemented. (You & Magnuson, 2018, report that the model is stable with such feedback added, given minor changes to other parameters. Feedback will allow TISK to account for top-down [lexical-on-phoneme] influences in spoken word recognition, although the need for feedback to account for such effects is controversial [see Norris, Cutler, & McQueen, 2000, for arguments and demonstrations of how models without feedback can do so]. However, as Magnuson, Mirman, Luthra, Strauss, & Harris, 2018, have demonstrated with the TRACE model, there are additional advantages of feedback in interactive-activation models: Feedback preserves accuracy and processing speed as noise is added to the inputs. You & Magnuson report similar benefits in TISK.)

There is a complication to our statement that "phoneme inputs map to appropriate . . . diphones": TISK does not actually use a simple open diphone coding like the one we described above. Instead, it is approximated by a *symmetry network* (schematized in Fig. 2). The symmetry network provides graded activation of diphones, such that the strength of activation decreases as the gap between phonemes increases. The symmetry network, with its relation to string kernel approaches, was inspired by the analysis of a learning network for visual word recognition, which approximated string kernels through such a symmetry structure (Hannagan, Dandurand, & Grainger, 2011; Hannagan & Grainger, 2012). Hannagan and Grainger discussed at length the plausibility of string kernels as neural coding strategies.

Relative advantages of TISK and TRACE

Hannagan et al. (2013) demonstrated that the TISK model behaves remarkably like TRACE, despite the large differences in their architectures. Hanngan et al. (2013) documented very similar behavior in the two models over time as a function of phonological similarity (e.g., classic differences between cohort and rhyme competition, as in Allopenna, Magnuson, & Tanenhaus, 1998). They also compared the models at an item grain of analysis by calculating recognition times (RTs) for every word in the original 211-word TRACE lexicon for both models. The correlations of the item-specific RTs were high for the two models (as high as .88 for RTs based on a time-based decision rule [target wins if it is the most highly activated for at least ten cycles], but still high for a rule based on either a relative threshold [target is more active than any other word by at least .05, $R = .83$] or an absolute threshold [target and no other word exceeds .75, $R = .68$]). They further compared models in terms of the influences of different "lexical dimensions" on RT, such as the number of cohorts (onset competitors), number of rhymes, and number of one-phoneme neighbors (differing from the target by no more than one phoneme deletion, addition, or substitution). The direction of the influence of each dimension was the same for both models. Indeed, lines of best fit (or RTs for every word in the lexicon plotted against each lexical dimension) were nearly identical. Thus, TISK performs remarkably similarly to TRACE in every comparison that has been conducted.

What advantages might TISK have compared to TRACE? First, consider the practical implications of the replacement of most time-specific units (TRACE) with time-invariant units (TISK). Due to its use of time-invariant nodes, TISK scales much more gracefully than TRACE. Hannagan et al. calculated that to accommodate a realistic inventory of 40 phonemes and 20,000 words, TISK would require approximately 30,000 nodes and 349 million connections, whereas TRACE would require 1.3 million nodes and 40 *billion* connections. This provides important practical advantages for TISK, as well as the potential for exploring pressing theoretical issues. Although TRACE has not been expanded beyond a lexicon approaching 1,000 words (Frauenfelder & Peeters, 1998), because it becomes impossible to construct many more English (or English-like) words using only 14 phonemes, expanding models of SWR to realistic lexicon sizes is a crucial concern, because the dynamics of lexical activation and competition may change in interesting ways as words gain more close and distant neighbors (i.e., words varying in degrees of similarity).[1]

Expanding the TRACE phoneme inventory is possible, but is a complex task that has not yet been tackled. However, we can get rough estimates of how quickly the model could process large lexicons by creating lexicons without worrying about connections to English words. That is, we can generate three- and four-phoneme words as random sequences of TRACE phonemes, and then run simulations to assess how long it takes TRACE to simulate a single word as lexicon size increases. We did this by creating such a random-sequence lexicon of 20,000 words that were three or four phonemes long, and then subset lexicons of 15,000, 10,000, 5,000, 1,000, and 200 words. We would expect processing time per word to increase with lexicon size because of the increase in nodes and connections required as lexicon size increases.

With TISK, we can easily expand the phoneme inventory, and so can use lexicons of real English words. We conducted exploratory simulations using 43 phonemes and lexicons with a range of sizes as our TRACE pseudo-lexicons, from 200 words to 20,000 words.[2] We might expect less of a processing time cost in TISK as the lexical and phoneme inventories are

[1] On the other hand, a larger lexicon can make it harder to measure the "pure" effects of specific influences because of the myriad interactions that will underlie a word's activation in the context of a large lexicon. It can be useful to reduce the lexicon size in order to better understand such "pure" effects. Magnuson (2015), for example, reduced the TRACE lexicon to a single word in order to isolate the effects of feedback.

[2] In these simulations, we did not try to modify the parameters to promote accuracy. We simply ran the simulations with default parameters to test how TISK would scale as lexicon size increased (changes in parameters would have negligible impact on timing) on a high-end Linux workstation. In future work we will strive to find parameters for large lexicons that will allow a high level of accuracy. For now, these explorations demonstrate the feasibility of scaling TISK to much larger lexicons than are typical for connectionist models of SWR.

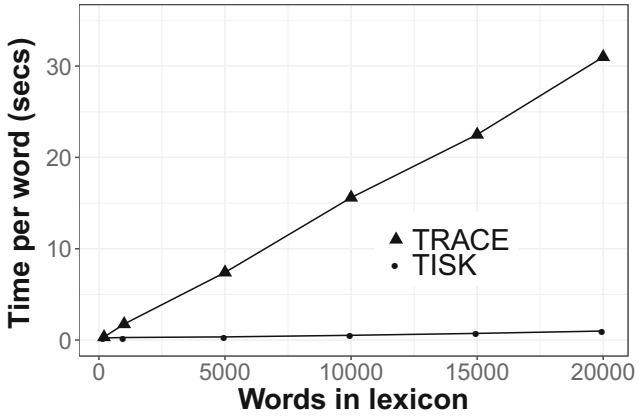

Fig. 3 Processing times per word as lexicon size is increased for TISK and TRACE.

increased, because of its reliance on time-invariant rather than time-specific units (as in TRACE). An advantage in our TISK implementation is the possibility of generating the results of simulating each word in the lexicon in parallel, using matrix operations.[3] Figure 3 compares the times per word required in both models as lexicon size increases. Both model implementations show fairly linear increases in processing time, but the slope for TISK is much shallower: Processing time for TISK increases from 226 ms per word with a lexicon of 200 words to 987 ms per word with a lexicon of 20,000 words (thus, it would take 5.5 h to simulate every word in the 20,000-word lexicon), whereas the increase is from 319 ms for a 200-word lexicon in TRACE to 31 s per word for a lexicon of 20,000 words (requiring 172 h [a bit more than 1 week] to simulate the entire lexicon[4]). Keeping in mind that TRACE would be exponentially slowed by increasing the phoneme inventory to a realistic level, this demonstrates a practical advantage of TISK over TRACE: It can be scaled more easily to large lexicons.

There are other practical trade-offs between the two models. The jTRACE implementation (Strauss et al., 2007) has a graphical user interface and graphing capabilities that promote accessibility without programming. TRACE also has a finer-grained (but still abstract) input representation than TISK (acoustic–phonetic features rather than phonemes). However, as we will demonstrate in this article, TISK is very flexible since it is implemented as a Python class; with a modicum of programming skill, a user can create arbitrarily complex simulation scripts, and can easily generate graphs as well. The original C code for TRACE can be similarly scripted, but requires a higher level of programming proficiency. jTRACE also includes scripting facilities, but these are unfortunately cumbersome.

Beyond practical considerations, there are other reasons a researcher might prefer TISK over TRACE, or wish to compare the models. As we discussed above, the symmetry network in TISK that provides a form of graded open-diphone coding is interesting not simply because it allows us to create a model that behaves very similarly to TRACE with dramatically reduced complexity, but also because of computational (theoretical) arguments for the utility of such encoding in biological systems as well as behavioral and brain imaging results consistent with open bigram coding for visual word recognition (Hannagan & Grainger, 2012), and evidence that similar coding may emerge in trained connectionist models (Dandurand et al. 2013). While arguments and evidence based on visual word recognition may not transfer to SWR, they certainly suggest the potential for alternative coding schemes for SWR. In addition, TISK and TRACE represent quite different theories at Marr's (1982) algorithmic level. Although the models have thus far behaved quite similarly, there may be domains where the two algorithms may in fact make distinct predictions. By making TISK available, we enable other researchers to explore potential similarities and differences between the algorithmic theories the models implement.

The need for easy-to-use implementations of simulating models

Magnuson et al. (2012) and Strauss et al. (2007) have discussed the importance of testing seemingly logical predictions about models with actual simulations. Magnuson et al. (2012) reviewed cases in which plausible predictions about TRACE turned out to be incorrect when simulations were actually conducted. It is thus crucial that developers of computational models share usable, understandable code or user-friendly applications (e.g., jTRACE; Strauss et al., 2007) so that others can replicate key simulations or devise their own tests of the models.

To that end, we have completely reimplemented the original TISK model with clear, commented Python code. We have added convenient features for generating graphs online during a Python session, saving graphs to a standard (PNG) format, extracting data to a standard Python scientific format (numpy; Oliphant, 2007), and saving data in simple text formats. In the rest of this article, we describe fundamental aspects of installing and using this new distribution of TISK.

Installing TISK

TISK 1.0 can be downloaded from its github repository: https://github.com/maglab-uconn/TISK1.0. Those familiar with git can use git methods to clone the repository (and possibly contribute modifications or extensions). Novices can just use the "clone or download" option to download the code to a local computer. TISK is implemented as a

[3] Although it might be possible to implement a similar batch function for TRACE, the memory demands would be exponentially greater, so the function might not be feasible for large lexicons on modern computers.

[4] Of course, the TRACE code could likely be optimized, and these speeds could be improved. However, these results represent the standard C version of TRACE.

single Python 3 script. Python 3 must be installed before TISK. TISK imports some standard libraries (`time, os`) but also requires two additional libraries: `numpy` and `matplotlib.pyplot` Some Python environments (e.g., Anaconda with the Spyder Python environment) come preinstalled with these libraries. We have confirmed that a Python novice (with some familiarity with integrated development environments [IDEs] such as RStudio) was able to get TISK up and running under Spyder using only the instructions we present here.

TISK 1.0 requires three files:

1. **`Basic_TISK_Class.py`** This is the core Python code for TISK.
2. **`Pronunciation_Data.txt`** This file corresponds to the original 211-word TRACE lexicon (McClelland & Elman, 1986) that was used in the original TISK article (Hannagan et al., 2013). We explain below how to specify a different lexicon file.
3. **`Phoneme_Data.txt`** This file is actually optional. If it is not specified, the phoneme list will become the set of unique phonemes in the pronunciation list. If a `Phoneme_Data.txt` file is specified, the program will first find all unique phonemes in the lexicon file and then add any additional phonemes found in `Phoneme_Data.txt`. This allows for cases in which one might wish for some reason to be able to present the model with pronunciations that use phonemes not used in any lexical item. Since the original version of TISK uses the TRACE lexicon, we originally limited TISK to the 14 phonemes implemented for TRACE:

   `/p/,/b/,/t/,/d/,/k/,/g/,/s/,/S/, /l/,/r/,/a/,/e/,/i/,/u/,/^/`. Note that phonemes have no internal structure but are simply localist, all-or-none representations; /p/ is no more similar to /b/ than it is to /u/. The phoneme set can be expanded, with phonemes being represented by any arbitrary single character. TISK is case sensitive. So, for example, to add `/æ/`, `/A/` would be a reasonable substitute.

Running TISK

The first step is setting up the Python environment. Perhaps the easiest option is to install Anaconda (Continuum Analytics; https://www.continuum.io/anaconda), a platform-independent environment. As of September 2017, the default Anaconda installation comes with Spyder, an IDE for Python that is already somewhat customized for scientific purposes. It comes with the `numpy` and `matplotlib.pyplot` libraries mentioned above. Once it is installed, launch Spyder. Use the menus or icons to open a file. Navigate to the directory with the TISK files. Set the working directory to that location (the working directory is shown near the top of the Spyder window; click the folder icon next to the working directory to change it).

Alternatively, use standard Python consoles under the Windows, Linux, or Macintosh operating systems. iPython is the recommended environment for working from the console.

Preliminary steps

The following commands prepare TISK for simulations.[5] Lines preceded by "#" are comments and can be skipped but can also be pasted into the Python interpreter, since they will be ignored:

```
# load the TISK functions

import Basic_TISK_Class as tisk

# load the phoneme and pronunciation [word] lists and
# prepare appropriate connections

phoneme_List, pronunciation_List = tisk.List_Generate()

# initialize the model with the the phoneme_List,
# pronunciation_List, number of time slots, and threshold

tisk_Model = tisk.TISK_Model(phoneme_List, pronunciation_List,
                             time_Slots = 10,
                             nPhone_Threshold = 0.91)
```

One can load an alternative lexicon file by specifying a filename in the generate command:

Ten time slots are enough for the words in the default "slex"

```
phoneme_List, pronunciation_List=
    tisk.List_Generate(
    pronunciation_File='other_lexicon.txt')
```

`pronunciation_List`. However, if the lexicon has longer words (longer than ten phonemes), the value of `time_Slots` can be increased accordingly (or just leave this parameter out: Just put a ")" after `pronunciation_List` and end the command there, and `time_Slots` will automatically be set to the length of the longest word in `pronunciation_List`). However, there are cases in which phoneme series longer than the longest word are required (e.g., to present long nonwords or to present series of words). In those cases, `time_Slots` must be adjusted appropriately.[6]

[5] Note that all of the code examples are collected in one Python file at the github repository ("Example_Code.py").

[6] A reviewer noted that a previous version of TISK allowed the user to set variables to unworkable combinations (e.g., setting `time_Slots` to a value smaller than the largest word in the lexicon creates problems). We have tried to anticipate unworkable parameter combinations and to include corresponding warnings to the user, but we probably cannot anticipate all possible problematic parameter combinations. However, if a parameter change alters the model's behavior in a negative way, the user should consider the possibility that the parameter set must be adjusted.

Initialize and/or modify parameters

Before running simulations, you must initialize the parameters. To use the default parameters of TISK 1.0 (Hannagan et al., 2013), just enter:

```
# initialize the model with default or current parameters
tisk_Model.Weight_Initialize()
```

The model does not automatically initialize, because this is the step at which all connections are made, and so forth, and initialization can take a long time for a large model with thousands of words. To control specific categories of parameters or specific parameters, use the following examples:

```
# change selected TISK parameters

tisk_Model.Decay_Parameter_Assign(
                    decay_Phoneme = 0.001,
                    decay_Diphone = 0.001,
                    decay_SPhone = 0.001,
                    decay_Word = 0.01)
tisk_Model.Weight_Parameter_Assign(
                    input_to_Phoneme_Weight = 1.0,
                    phoneme_to_Phone_Weight = 0.1,
                    diphone_to_Word_Weight = 0.05,
                    sPhone_to_Word_Weight = 0.01,
                    word_to_Word_Weight = -0.005)
tisk_Model.Feedback_Parameter_Assign(
                    word_to_Diphone_Activation = 0,
                    word_to_SPhone_Activation = 0,
                    word_to_Diphone_Inhibition = 0,
                    word_to_SPhone_Inhibition = 0)
tisk_Model.Weight_Initialize()
```

To modify a subset of parameters, just specify the subset, and the others will retain their current values. For example:

```
tisk_Model.Decay_Parameter_Assign(
                    decay_Phoneme = 0.002,
                    decay_Diphone = 0.0005)
```

Parameter details

To list the current parameters, enter the following command:

```
tisk_Model.Parameter_Display()
```

Note that if the parameters have not yet been initialized, this will result in an error message. When the parameters are initialized, this command will return a list like this:

```
nPhone_Threshold: 0.91
iStep: 10
time_Slots: 10
Phoneme_Decay: 0.001
Diphone_Decay: 0.001
SPhone_Decay: 0.001
Word_Decay: 0.01
Input_to_Phoneme_Weight: 1.0
Phoneme_to_Phone_Weight: 0.1
Diphone_to_Word_Weight: 0.05
SPhone_to_Word_Weight: 0.01
Word_to_Word_Weight: -0.005
Word_to_Diphone_Activation_Feedback: 0.0
Word_to_SPhone_Activation_Feedback: 0.0
Word_to_Diphone_Inhibition_Feedback: 0.0
Word_to_SPhone_Inhibition_Feedback: 0.0
```

Table 1 describes these parameters.

Changing parameters The example above (under "Initialize and/or modify parameters") shows how to change several of these variables. Others must be modified by changing the Basic_TISK_Class.py file directly.

Basic methods for TISK simulations, graphing, and data export

Simulate processing of a phoneme string and graph results for phonemes and words

Here is an example of a basic command that calls a simulation of the word "pat" (technically, it is more correct to say "a simulation of the pronunciation 'pat'," since the user can specify pronunciations that are not in the lexicon, i.e., the `pronunciation_List`):

```
# trigger a simulation without producing output;
# this prepares a model for inspection

tisk_Model.Display_Graph(pronunciation='pat')
```

Specifying a pronunciation with this command triggers a simulation with that phoneme sequence as the input. The user can specify any arbitrary sequence of phonemes (that is, the sequence does not have to be a word), as long as the phonemes are in the `phoneme_List`, and the sequence length does not exceed the maximum length.

On its own, this command doesn't do anything apparent to the user (though the simulation is in fact conducted). To create

Table 1 TISK parameters

Parameter	Default	Description
nPhone_Threshold	0.91	N-phone nodes only send activation to words when activation exceeds this threshold.[a]
iStep	10	Number of activation cycles between time slots (e.g., if iStep is 10, the first phoneme will be presented from cycles 0-9, the second from 10-19, etc.).[b]
time_Slots	10	Specify the maximum length of words. Time slots will be spaced iStep cycles apart.[7]
Phoneme_Decay	0.001	Governs how much activation from previous time step is retained for phoneme input nodes.
Diphone_Decay	0.001	Decay for diphone nodes.
SPhone_Decay	0.001	Decay for single phone nodes in the diphone layer.
Word_Decay	0.01	Decay for word nodes.
Input_to_Phoneme_Weight	1	Gain from input pattern to phoneme input nodes.
Phoneme_to_Phone_Weight	0.1	Gain: phoneme inputs to appropriate single-phonemes.
Diphone_to_Word_Weight	0.05	Gain from diphones to words containing them.
SPhone_to_Word_Weight	0.01	Gain from single-phonemes to words containing them.
Word_to_Word_Weight	-0.005	Lateral inhibition.
Word_to_Diphone_Activation_Feedback	0	Positive weights from words to constituent diphones.[c]
Word_to_SPhone_Activation_Feedback	0	Positive weights from words to constituent phonemes.[a]
Word_to_Diphone_Inhibition_Feedback	0	Weights from words to *non*-constituent[d] diphones.
Word_to_SPhone_Inhibition_Feedback	0	Weights from words to *non*-constituent single phonemes.[b]

Parameters were established through trial and error with the goal of obtaining robust, TRACE-like performance. The default parameters were easily discovered, and limited exploration of the parameter space indicated that a wide range of values would be possible for all parameters, although many parameters interact, so that changing one may require changes in one or several others in order to maintain performance.

[a] This threshold relates importantly to time_Slots, and changing time_Slots will lead to a warning and recommendation to adjust the parameters to conform to the following formula: [iStep x (time_Slots - 1) + 1] / [iStep x time_Slots], when Phoneme_to_Phone_Weight x time_Slots) ≤ nPhone_Threshold.

[b] Thus, a simulation will have iStep x time_Slots cycles. So if iStep is 10 and time_Slots is 8, Phone 1 would be sustained over Cycles 0–9, Phone 2 over Cycles 10–19, and Phone 8 over Cycles 70–79.

[c] Note that feedback was not used in Hannagan et al. (2013). You & Magnuson, (2018) report that the model is stable with feedback, with minor changes to other parameters.

[d] The "inhibition feedback" nodes are meant for top-down inhibition (consistent, e.g., with the Cohort model). These are connections to diphones or phones *not* contained in a word. If this parameter is set to a positive value, words will activate all the sublexical units they do not contain. These parameters have not been tested, and users are warned that they may radically alter the model's behavior.

a graph that is displayed within an IDE, add arguments to display specific phonemes:

```
# trigger a simulation and create a phoneme input graph

tisk_Model.Display_Graph(
      pronunciation='pat',
      display_Phoneme_List = [('p', 0), ('a', 1), ('t', 2)])
```

This code means "input the pronunciation /pat/, and export a graph with phoneme activations for /p/, /a/, and /t/ in the first, second, and third positions, respectively." "Phoneme" is the label that specifies the phonemic *input* units. Since the phoneme input nodes are duplicated at each time slot, time slots must be specified to select a phoneme input node. The command above creates a graph like the one shown in Fig. 4 (displayed in-line in some IDEs, including Spyder).

We can extend this procedure to create activation graphs for diphones, single phones, and words. The following example creates one of each:

```
# trigger a simulation and make 3 graphs

tisk_Model.Display_Graph(pronunciation='pat',
                  display_Diphone_List = ['pa', 'pt', 'ap'],
                  display_Single_Phone_List = ['p', 'a', 't'],
                  display_Word_List = ['pat', 'tap'])
```

This command results in the three graphs shown in Fig. 5. The diphone and single-phone plots correspond to the units at the *n*-phone level, depicted in Fig. 2, whereas the word plot corresponds to the word level. Arbitrarily many items may be specified in such commands. To export the graphs in a standard graphics format (PNG), simply add one more argument to the command:

```
# trigger a simulation, make 3 graphs, save them as PNG files

tisk_Model.Display_Graph(pronunciation='pat',
                  display_Diphone_List = ['pa', 'pt', 'ap'],
                  display_Single_Phone_List = ['p', 'a', 't'],
                  display_Word_List = ['pat', 'tap'],
                  file_Save = True)
```

This command will save the three graphs as p_a_t.Diphone.png, p_a_t.Single_Phone.png, and p_a_t.Word.png.

The graphs that are produced are fairly basic and may not suffice for publication. A user could tweak the code to adjust the graphs, but a more typical procedure would be to export the underlying data and create publication-ready graphs using other software, such as R. Let's next look at the methods for extracting data.

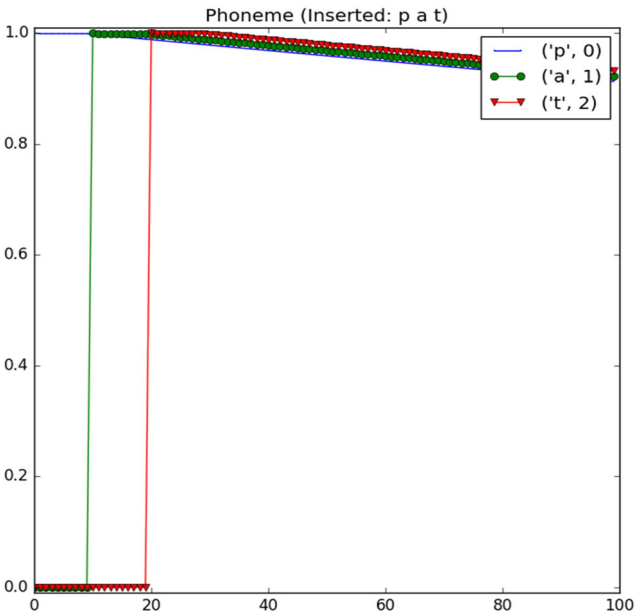

Fig. 4 Example phoneme input plot.

Extract simulation data to a `numpy` matrix

The basic method for extracting data as a `numpy` matrix is as follows:

```
# trigger a simulation and ready data structures
# without creating any output

tisk_Model.Extract_Data(pronunciation='pat')
```

The method is similar to the graphing functions. The code above is the core command that readies appropriate structures for extraction, but it does not by itself generate any result for the user. To extract data, arguments specifying the details desired are required. For example, to get data corresponding to the word plot above (showing the activations of /pat/ and /tap/ given the input /pat/), the following command would put the data in a `numpy` matrix called `result`:

```
# trigger a simulation and assign data structures to 'result'

result = tisk_Model.Extract_Data(pronunciation='pat',
                           extract_Word_List = ['pat', 'tap'])
```

When this command is executed, the `result` variable becomes a list with length 1, consisting of a single `numpy` matrix with shape (2, 100). The first and second rows are the activation patterns of the word units for /pat/ and /tap/, respectively.

Let's try a slightly more complex example.

```
# trigger a simulation and assign data structures to 'result'

result = tisk_Model.Extract_Data(pronunciation='pat',
       extract_Phoneme_List = [('p', 0), ('a', 1), ('t', 2)],
       extract_Single_Phone_List = ['p', 'a', 't'])
```

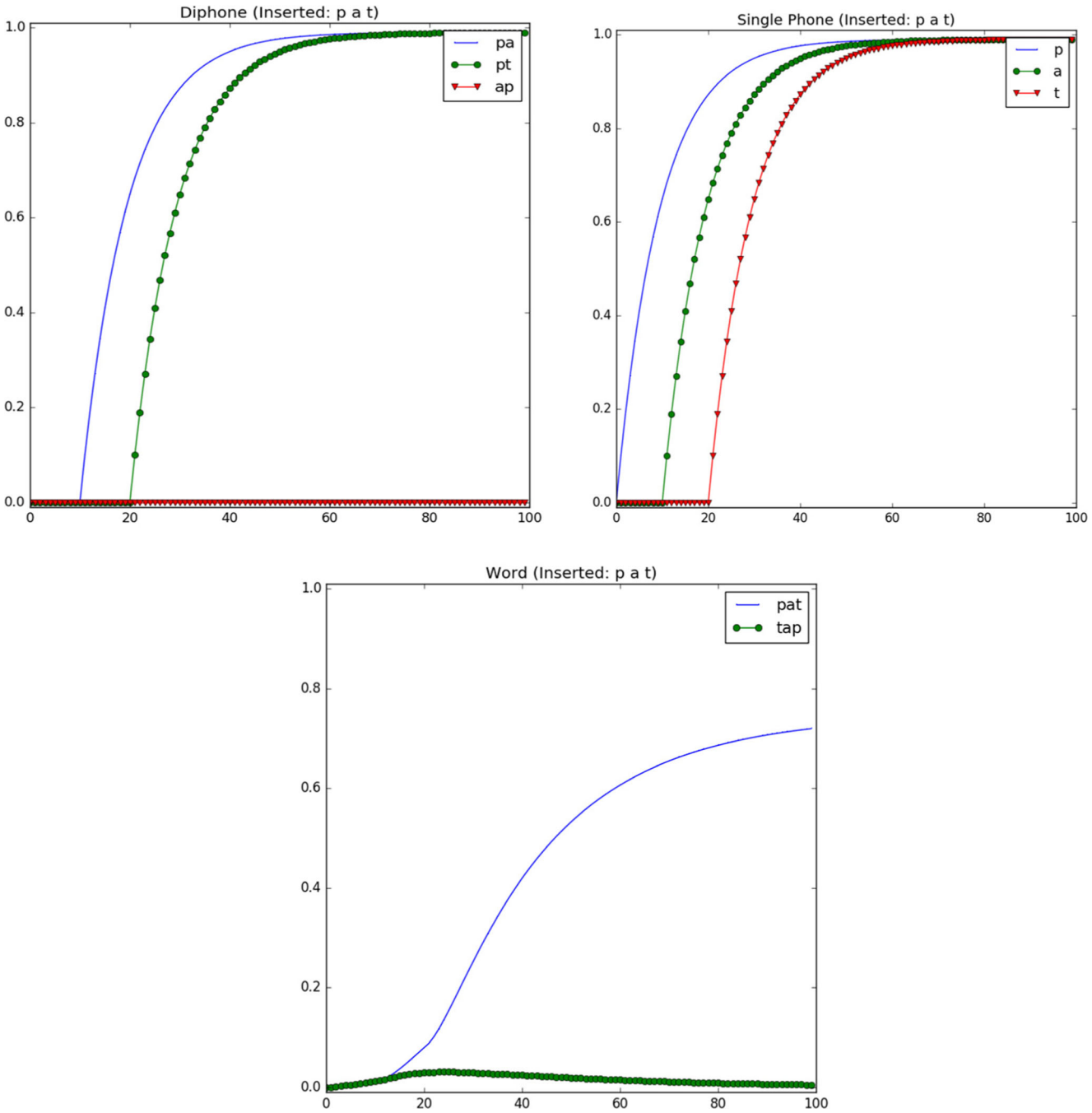

Fig. 5 Examples of diphone, single-phone, and word graphs that can be generated in-line in an IDE.

Here, `result` becomes a list with length 2. The first item is a `numpy` matrix with the input unit activations for the three specified phonemes across the 100 steps of the simulation. The second is a `numpy` matrix with the activations of the specified single phonemes in the *n*-phone layer over the 100 steps of the simulation.

Export simulation data to text files

To export the results to text files, we add a parameter:

```
# trigger a simulation and assign data structures to 'result'

result = tisk_Model.Extract_Data(pronunciation='pat',
                                 extract_Word_List = ['pat', 'tap'])
```

This creates a text file called "`p_a_t.Word.txt`". The file has 102 columns and three lines. The first line is a header, labeling the columns; the subsequent lines contain the data. The first column is the input string ("p a t"), and the second is the specified word to track (line 2 is "pat," and line 3 is "tap").

Columns 3–102 are activations for the corresponding word in Cycles 0–99.

Batch simulation of multiple words

Of course, we often want to conduct simulations of many words. First, here's an easy way to assess average performance on a specified list of items:

```
# get mean RT and accuracy for the specified set of words

rt_and_ACC = tisk_Model.Run_List(pronunciation_List =
['baks','bar','bark','bat^l','bi'])
```

Given this command, the model will simulate the five words and check the RT and accuracy for each. The variable `'acc_and_RT'` will be a list of six items, with the mean RT and accuracy for the specified words computed using three different methods (`abs` = based on an absolute threshold [target must exceed threshold], `rel` = relative threshold [target must exceed next most active item by threshold], `tim` = time-based threshold [target must exceed next most active item by threshold for at least a specified number of cycles]):

`rt_and_ACC[0]`: Mean of RTabs
`rt_and_ACC[1]`: Mean of ACCabs
`rt_and_ACC[2]`: Mean of RTrel
`rt_and_ ACC[3]`: Mean of ACCrel
`rt_and_ACC[4]`: Mean of RTtim
`rt_and_ACC[5]`: Mean of ACCtim

More commonly, one might want to evaluate the mean accuracy and RT for every word in the current lexicon with the current parameter settings. The following command would do this, where we specify the `pronunciation_List` to be the full `pronunciation_List`::

```
# get mean RT and accuracy for all words in pronunciation_List
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = pronunciation_List)
```

The parameters used for the different accuracy methods can also be modified. The default criteria are: `abs` = 0.75, `rel` = 0.05, `tim` = 10 (time steps). These criteria refer to absolute activation values (to win, a target's activation must exceed .75), relative activation values (to win, the target's activation must exceed all other words' activations by at least .05), and time steps (to win, the target must have the highest activation, and its activation must exceed that of all other words' activations for at least ten time steps). Here is an example in which the criteria for each accuracy method are specified:

```
# get mean RT and accuracy for specified word list with
# specified accuracy criteria for abs, rel, and tim, respectively

rt_and_ACC = tisk_Model.Run_List(
   pronunciation_List = ['baks','bar','bark','bat^l','bi'],
   absolute_Acc_Criteria=0.6,
   relative_Acc_Criteria=0.01,
   time_Acc_Criteria=5)
```

Often we may want to obtain the RT values for each word in a list, rather than the mean values. We can do this using the `reaction_Time` flag with the `Run_List` procedure. Currently, this requires you to specify a file to write the data to (which could be read back in using standard Python techniques):

```
tisk_Model.Run_List(pronunciation_List =
   ['baks','bar','bark','bat^l','bi'],
   output_File_Name = "Test",
   reaction_Time=True)
```

This will create an output file named Test_Reaction_Time.txt. Its contents would be:

Target	Absolute	Relative	Time_Dependent
baks	58	40	46
bar	84	28	33
bark	74	52	56
bat^l	60	39	46
bi	nan	23	13

Accuracy is indicated by the value for each word for each accuracy criterion. Items that were correctly recognized according to the criterion will have integer values (cycle at which the criterion was met). Items that were not will have values of `"nan"` (not a number, a standard designation for a missing value). In the present example, we can see that /bi/ ("bee") did not meet the absolute criterion.

If you wanted to obtain the RTs for every word in your lexicon, you would replace the word list with `pronunciation_List`:

```
tisk_Model.Run_List(pronunciation_List = pronunciation_List,
   output_File_Name = "all",
   reaction_Time=True)
```

Extract data for multiple words in text files

To export the results for multiple words, we can use the `'Run_List'` function again, as follows:

```
# get mean RT and accuracy for specified word list
# with specified accuracy criteria but ALSO
# save activation histories in 'raw' and 'category' formats

rt_and_ACC = tisk_Model.Run_List(
   pronunciation_List = ['baks','bar','bark','bat^l','bi'],
   output_File_Name = 'Result',
   raw_Data = True,
   categorize=True)
```

When we run this code, we get text files with what we call "raw" and "category" outputs. *Raw files* (e.g., for this example, Result_Word_Activation_Data.txt) contain the activations for every word in the lexicon at every time step for each target specified. The file format is very simple. There is a one-line header with column labels. The first column is "target," the second is "word," and the following columns are cycles 0–*C*, where *C* is the final cycle (which will have the value [(`time_Slots x IStep_Length`) - 1]. So, in a row that begins

```
baks ad   0.0  0.0  ...
```

the target is /baks/ ("box"), and this row contains the activation of /ad/ when /baks/ was the target over all time steps. To find the actual target activations, find the row that has /baks/ in the first two columns:

```
baks baks 0.0 0.0 0.001 0.002557431 0.0044149027022 ...
```

The phoneme file (e.g., Result_Phoneme_Activation.txt) has a similar structure, but it adds the needed phoneme position column. Here are some examples for /b/ at Positions 0–3 when the input is /baks/. This shows how the phoneme activation ramps up slightly after insertion but then begins to decay:

Target	Phoneme	Position	0	1	2	3	4	5
baks	b	0	1	0.999	0.999001	0.999000999	0.999000999	0.999000999
baks	b	1	0	0	0	0	0	0
baks	b	2	0	0	0	0	0	0
baks	b	3	0	0	0	0	0	0

The diphone file (e.g., Result_Diphone_Activation.txt) simply contains in each row the target (word), a diphone, and then the activation of that diphone over time steps, given that target as input. The single phone file (e.g., Result_Single_Phone_Activation.txt) does the same thing for single phonemes from the *n*-phone layer.

Category files (e.g., for this example, Result_Category_Activation_Data.txt) contain several categories of activations for a given target word (e.g., for /baks/ ["box"]):

Target	Category	0	1	2	3	4	5	6
baks	Target	0	0	0.001	0.002557	0.004415	0.006421	0.008485
baks	Cohort	0	0	0.001	0.002527	0.004313	0.006201	0.008103
baks	Rhyme	0	0	0	0	0	0	0
baks	Embedding	0	0	0	0	0	0	0
baks	Other	0	0	0.000141	0.000352	0.000591	0.000837	0.00108

The Target rows contain the activations of that target word over time. In addition, for every word, this file includes the *mean* activations for different categories of items. These are:

Cohort: Words matching the target in the first two phonemes
Embedding: Words embedded in the target (e.g., AT is embedded in CAT)
Rhyme: Words mismatching the target only at first position[7]
Other: The mean of all other words (excluding the target, cohorts, embeddings, and rhymes)

The "Other" category provides a rough baseline for words unrelated to the target, even though it may still

[7] In most cases, these are actually rhymes (/kæt/, /bæt/, /sæt/). In cases in which a word begins with a vowel, they may not be true rhymes (e.g., AND /ænd/ vs. END /ɛnd/). Rather than coining a new term or using a long description ("words mismatching only in first position"), we use "rhyme" and ask users to be mindful of the exceptions.

contain many related words (e.g., neighbors not included in cohorts, embeddings, and rhymes). As long as the lexicon is large, this provides a good baseline that should hover near 0.

You can also make graphs that correspond to the category data. For example, let's plot the category data for /baks/ (the resulting graph is shown in Fig. 6).

```
# trigger a simulation and make a graph

tisk_Model.Average_Activation_by_Category_Graph(
        pronunciation_List=['baks'])
```

We can also get average data and average plots for a set of specified words. For example, suppose for some reason we were interested in the average category plot for the words /pat/, /tap/, and /art/ ("pot," "top," and "art"). We could call this command, with the result shown in Fig. 7:

```
# trigger a simulation and make a graph

tisk_Model.Average_Activation_by_Category_Graph(
        pronunciation_List=['pat', 'tap', 'art'])
```

To save this graph as a PNG file, add the `file_Save=True` argument:

```
# trigger a simulation, make a graph, save them as PNG files

tisk_Model.Average_Activation_by_Category_Graph(
        pronunciation_List=['pat','tap', 'art'],
        file_Save=True)
```

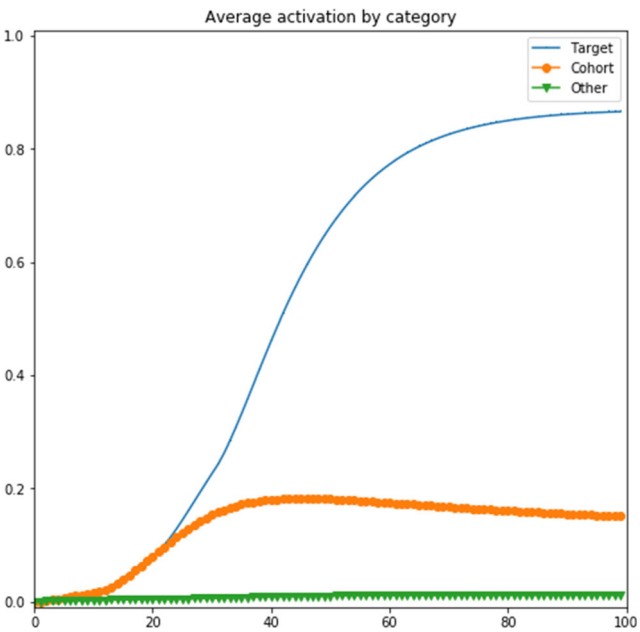

Fig. 6 Category plot for /baks/ ("box"). Because there are no rhymes (i.e., other words ending in /aks/) or embeddings (words completely embedded in /baks/) in the lexicon, those lines are not plotted.

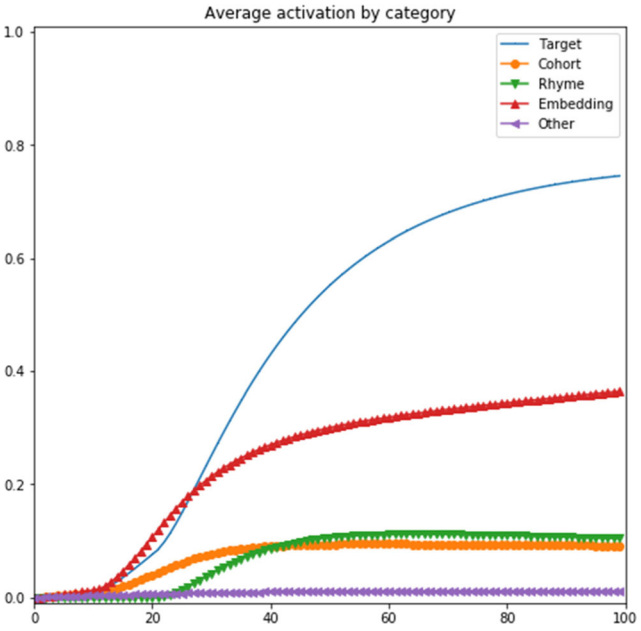

Fig. 7 Category graph averaging over the words /pat/, /tap/, and /art/ ("pot," "top," "art").

By default, the graph associated with this command will be saved as Average_Activation_by_Category_Graph.png. To specify a different filename (important if you wish to, e.g., loop through many example sets in a Python script), you can do so as follows:

```
# trigger a simulation, make a graph, save them as PNG files

tisk_Model.Average_Activation_by_Category_Graph(
        pronunciation_List=['pat','tap', 'art'],
        output_File_Name='Result',
        file_Save=True)
```

In this case, the exported graph file name will become Result_Average_Activation_by_Category_Graph.png. By setting the `output_File_Name` parameter, you can control the prefix of exported file name.

To save the corresponding data file in text format, use the `Run_List` function:

```
tisk_Model.Run_List(
          pronunciation_List=['pat','tap', 'art'],
          output_File_Name='Result',
          categorize=True)
```

Getting comprehensive data for every word in the lexicon

If we combine our last few examples, we can save activations for simulations of every word by replacing our `pronunciation_List` argument above with

`pronunciation_List` (i.e., all words included in `pronunciation_List`):

```
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = pronunciation_List,
     output_File_Name = 'all_words',
     raw_Data = True,
     categorize=True)
```

By extension, we can also generate a mean category plot over *every word in the lexicon* (with the result shown in Fig. 8):

```
# make a graph for all words in pronunciation_List

tisk_Model.Average_Activation_by_Category_Graph(
        pronunciation_List = pronunciation_List)
```

One detail is very important to be aware of for category plots: The category means are based only on cases for which the competitor type exists. For example, if a word has no rhyme competitors, the mean rhyme activation for that word would be 0 at every time step. Our goal is to characterize the time course of activation for competitor types when they exist. Thus, we exclude items for which the value would be zero at every time step because they do not have a certain competitor type. However, such an item would still contribute to the mean target time course and would be included for other categories where it did have relevant competitors.

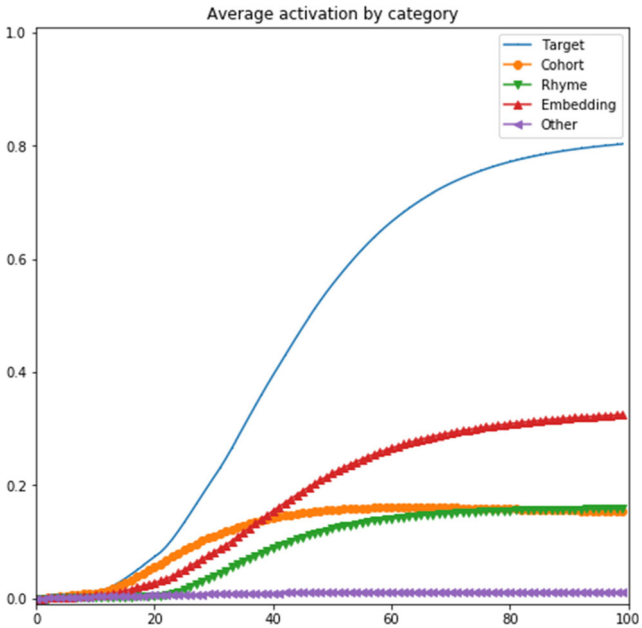

Fig. 8 Category plot averaging over all words in the original 211-word TRACE lexicon.

Competitor details

This points to the need, on occasion, to know more about the details of the competitors for a specific word. TISK includes special commands for getting this information.

```
# Getting the competitor information
competitor_List = tisk_Model.Category_List(
            pronunciation = 'b^s')
```

When you use this command, the model will return four lists. The lists contain the cohorts, rhyme, embedding, and other words, respectively.

`competitor_List[0]`: cohort list
`competitor_List[1]`: rhyme list
`competitor_List[2]`: embedding list
`competitor_List[3]`: other list

To see the contents of `competitor_List`, use the following sorts of command:

```
# Display cohort list
print(competitor_List[0])
```

To see the count of a competitor type, get the length of a specific list:

```
# Display rhyme count
print(len(competitor_List[1]))
```

You can also inspect the details of the competitors for a list of words, rather than a single word:

```
# Display the mean competitor count
tisk_Model.Display_Mean_Category_Count(
      pronunciation_List = pronunciation_List)
```

This command will display the mean number of each competitor type for the specified word list. The preceding command would display the results for the full default lexicon, as follows:

```
Mean cohort count: 4.33018867925
Mean rhyme count: 1.08490566038
Mean embedding count: 1.25943396226
Mean other count: 204.622641509
```

Batch size control

Depending on the size of your lexicon and the memory available on your computer, you may see the "Memory Error" message when you run batch mode. Batch-mode simulation

is not possible if the memory of the machine is too small to handle the size of the batch. To resolve this, you can use the `batch_Size` parameter to reduce the size of the batch. This parameter determines how many word simulations are conducted in parallel. It only controls the batch size, and does not affect any result. You will get the same result with any batch size your computer's memory can handle. The default value is 100. To see whether your computer memory can handle it, you can test larger values.

```
rt_and_ACC = tisk_Model.Run_List(
     pronunciation_List = pronunciation_List,
     batch_Size = 10)
```

Reaction time and accuracy for specific words

To check specific kinds of RTs for specific words, use commands like these:

```
result = tisk_Model.Run('pat')
abs_RT = tisk_Model.RT_Absolute_Threshold(
                  pronunciation = 'pat',
                  word_Activation_Array = result[3],
                  criterion = 0.75)
rel_RT = tisk_Model.RT_Relative_Threshold(
                  pronunciation = 'pat',
                  word_Activation_Array = result[3],
                  criterion = 0.05)
tim_RT = tisk_Model.RT_Time_Dependent(
                  pronunciation = 'pat',
                  word_Activation_Array = result[3],
                  criterion = 10)
```

If TISK has successfully recognized the inserted word, the RT will be returned. If the model has failed to recognize the word, the returned value is `'numpy.nan'`. One can change the criterion by modifying the parameter `'criterion'`.

Alternatively, we could get all accuracy and RT values for a specific word by using a command we introduced earlier:

```
rt_and_ACC = tisk_Model.Run_List(pronunciation_List = ['pat'])
```

More complex simulations

Since TISK is implemented as a Python class, the user can do arbitrarily complex simulations by writing Python scripts. Doing this may require the user to acquire expertise in Python that is beyond the scope of this short introductory guide. However, to illustrate how one might do this, we include one full, realistic example here. In this example, we will compare competitor effects as a function of word length, by comparing competitor effects for words that are three phonemes long versus words that are five phonemes long. All explanations are embedded as comments (preceded by "#") in the code below. Graphical results are in Fig. 9.

```
# first, select all words that have length 3 in the lexicon
length3_Pronunciation_List = [x for x in pronunciation_List if
len(x) == 3]

# now do the same for words with length 5
length5_Pronunciation_List = [x for x in pronunciation_List if
          len(x) == 5]

# make a graph of average competitor effects for 3-phoneme words
tisk_Model.Average_Activation_by_Category_Graph(
          pronunciation_List = length3_Pronunciation_List)

# make a graph and also save to a PNG file
tisk_Model.Average_Activation_by_Category_Graph(
          pronunciation_List= length3_Pronunciation_List,
          file_Save=True,
          output_File_Name='length_3_category_results.png')

# make a graph and also save to a PNG file
tisk_Model.Average_Activation_by_Category_Graph(
          pronunciation_List= length5_Pronunciation_List,
          file_Save=True,
          output_File_Name= 'length_5_category_results.png')

# save the length 3 data
tisk_Model.Run_List( pronunciation_List =
          length3_Pronunciation_List,
          output_File_Name='length3data',
          raw_Data = True, categorize = True)
```

Note that when you save data to text files, if you leave out the " `categorize = True` " argument, the file with word results will include the results over time for every target in the pronunciation list. The first column will list the target, the second will list the time step, and then there will be one column for every word in the lexicon (i.e., with the activation of that word given the current target at the specified time step).

We see several interesting differences in Fig. 9. First, three-phoneme targets activate faster (e.g., hitting a value of .4 about ten cycles sooner than five-phoneme words). On average, cohort effects appear to be weaker for longer words, and rhyme effects appear to be stronger. These results represent testable hypotheses about human SWR. They may also represent testable differences between models (e.g., if TRACE or some other model predicts little or no effect of word length on the magnitude of competitor effects). We might also consider competing explanations for the differences we see. It could be that cohort effects tend to be stronger for shorter words because short cohort pairs will tend to have a greater proportion of overlap than longer pairs (e.g., three-phoneme pairs will overlap minimally in two out of three phonemes, whereas five-phoneme pairs will overlap minimally in two out of five phonemes). Similarly, longer rhymes will have a greater proportion of overlap (four of five phonemes vs. two of three phonemes). Alternatively, it could be that shorter words simply tend to have *more* cohorts and *fewer* rhymes than longer words. We can check this using the display command introduced above; we leave it as an exercise for the reader to discover

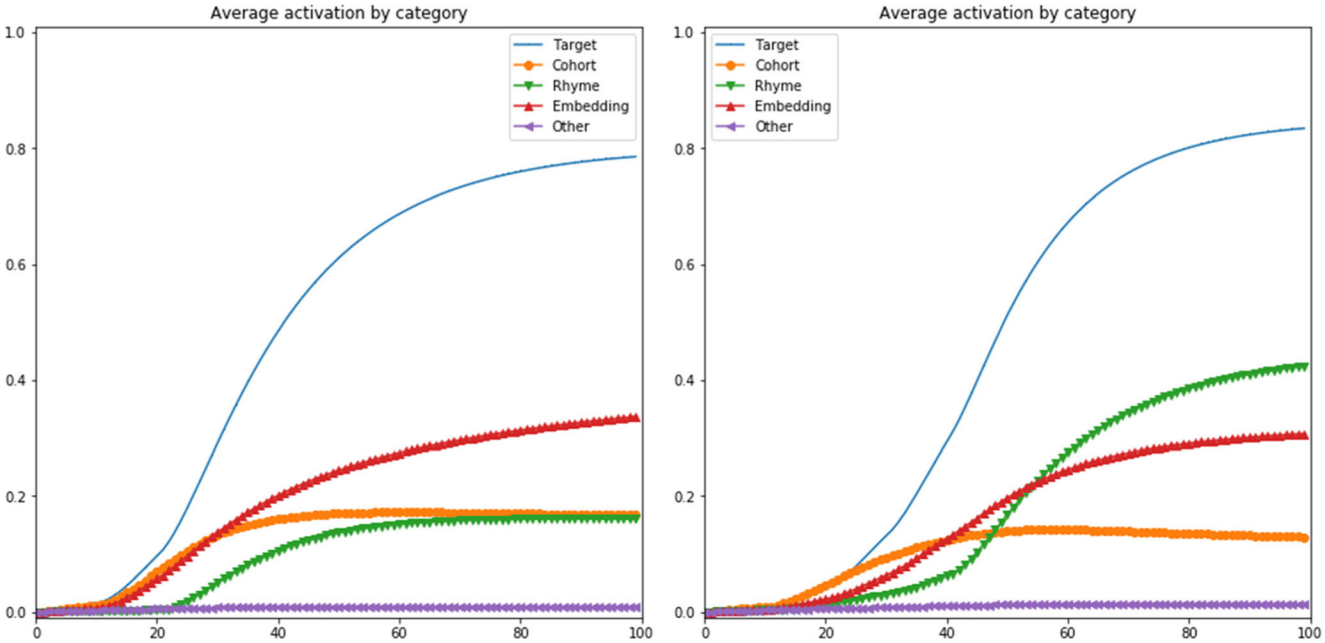

Fig. 9 Comparing the competitor time course effects for all three-phoneme words (left) versus all five-phoneme words.

whether there is a confound of word length with the competitor counts.

```
# check competitor counts of length 3 and length 5 lists
tisk_Model.Display_Mean_Category_Count(
          length3_Pronunciation_List)

tisk_Model.Display_Mean_Category_Count(
          length5_Pronunciation_List)
```

Of course, this example merely scratches the surface of what is possible, since TISK is embedded within a complete scripting language. Using standard Python syntax, we can easily filter words by specifying arbitrarily complex conditions. Here are some examples:

```
# Select words with length greater than or equal to 3 with
# phoneme /a/ in position 2
filtered_Pronunciation_List = [x for x in pronunciation_List if
         len(x) >= 3 and x[2] == 'a']

# make a graph
tisk_Model.Average_Activation_by_Category_Graph(
        pronunciation_List = filtered_Pronunciation_List)

# select words with length greater than or equal to 4 with
# phoneme /a/ in position 2 and phoneme /k/ in position 3
filtered_Pronunciation_List = [x for x in pronunciation_List if
         len(x) >= 4 and x[2] == 'a' and x[3]=='k']
```

Even more complex examples

These examples scratch the surface of what is possible with TISK. It is not possible here to include examples that encompass the full range of simulations that researchers may wish to conduct. We encourage users who are unsure of how to conduct a desired simulation to contact us for advice, which we will provide on an as-available basis.

Ease of use relative to other models

Note that because TISK is implemented as a Python class, users with a modicum of programming skill can conduct complex and comprehensive simulations with TISK much more easily than they could with other implemented models. Consider the example above of the average competition for cohorts, rhymes, and embeddings for three- and five-phone words. Doing this required six lines of code (six instructions, even though some are spread over multiple lines for ease of reading). To do equivalent simulations with TRACE or Shortlist B, the experimenter would first have to create scripts outside of the model to identify the desired word sets, then run the simulations, and then write complex analysis scripts. The analysis scripts would have to include algorithms for finding the cohorts, rhymes, and embeddings and averaging the activations of those competitor types over all words included in the simulations. Despite the greater ease of use afforded by using Python, which allows the user to flexibly interact with the TISK class, there is no apparent sacrifice in speed.

Future plans

Our immediate plans for extending TISK include testing the behavior of TISK with and without feedback, and whether TISK with feedback continues to operate

similarly to TRACE at the item level (You & Magnuson, 2018), as well as whether TISK exhibits benefits of feedback similar to those shown by TRACE (Magnuson et al., 2018). When this project is complete, we will update the code and documentation at the github site (https://github.com/maglab-uconn/TISK1.0) with alternative parameter settings that optimize TISK's behavior with feedback. A full discussion of the utility of a version control repository like github is beyond the scope of this article, but note that other researchers who wish to extend TISK may do so and contribute either to the primary development code of TISK or create their own "forks" on github for alternative versions.

Conclusions

TISK has advantages over similar implemented models, such as TRACE (McClelland & Elman, 1986) and jTRACE (Strauss et al., 2007), because it can be easily extended to realistic phoneme and word inventories (as we discussed above). The numbers of nodes and connections required for such expansion are exponentially less in TISK than in TRACE, thanks to its approximation of open-diphone coding, which allows most of the time-specific nodes in TRACE to be replaced with time-invariant nodes. We hope that by releasing TISK as open-source software, we will make using TISK possible for many researchers who might not otherwise attempt to use the model (or any model), and that this will promote comparisons of TISK and other extant or future models. We also hope that users who extend or improve the model will contribute to the github repository.

Author note This work was supported in part by National Institute of Child Health and Human Development P01-HD001994 (J. Rueckl, principal investigator), and National Science Foundation IGERT 114399 (J.S. Magnuson, principal investigator). We are indebted to Thomas Hannagan for expert advice and for providing the original TISK code to start this project. We thank Paul Allopenna for helpful comments on this document and the TISK source code.

References

Allopenna, P. D., Magnuson, J. S., & Tanenhaus, M. K. (1998). Tracking the time course of spoken word recognition using eye movements: Evidence for continuous mapping models. *Journal of Memory and Language*, *38*, 419–439. doi:https://doi.org/10.1006/jmla.1997.2558

Dandurand, F., Hannagan, T., and Grainger, J. (2013). Computational models of location-invariant orthographic processing. Connect. Sci. 25, 1–26. doi:https://doi.org/10.1080/09540091.2013.801934

Davis, M. H., Marslen-Wilson, W. D., & Gaskell, M. G. (2002). Leading up the lexical garden-path: Segmentation and ambiguity in spoken word recognition. *Journal of Experimental Psychology: Human Perception and Performance*, *28*, 218–244.

Farrell, S., & Lewandowsky, S. (2010). Computational models as aids to better reasoning in psychology. *Current Directions in Psychological Science*, *19*, 329–335.

Frauenfelder, U. H., & Peeters, G. (1998). Simulating the time course of spoken word recognition: An analysis of lexical competition in TRACE. In J. Grainger & A. M. Jacobs (Eds.), *Localist connectionist approaches to human cognition* (pp. 101–146). Mahwah, NJ: Erlbaum.

Grossberg, S., & Kazerounian, S. (2011). Laminar cortical dynamics of conscious speech perception: Neural model of phonemic restoration using subsequent context in noise. *Journal of the Acoustical Society of America*, *130*, 440–460.

Hannagan, T., Dandurand, F., & Grainger, J. (2011). Broken symmetries in a location invariant word recognition network. *Neural Computation*, *23*, 251–283.

Hannagan, T., & Grainger, J. (2012). Protein analysis meets visual word recognition: A case for string kernels in the brain. *Cognitive Science*, *36*, 575–606. doi:https://doi.org/10.1111/j.1551-6709.2012.01236.x

Hannagan, T., Magnuson, J. S., & Grainger, J. (2013). Spoken word recognition without a TRACE. *Frontiers in Psychology*, *4*, 563. doi:https://doi.org/10.3389/fpsyg.2013.00563

Lewandowsky, S. (1993). The rewards and hazards computer simulations. *Psychological Science*, *4*, 236–243. doi:https://doi.org/10.1111/j.1467-9280.1993.tb00267.x

Magnuson, J. S. (2008). Nondeterminism, pleiotropy, and single word reading: Theoretical and practical concerns. In E. Grigorenko & A. Naples (Eds.), *Single word reading* (pp. 377–404). Mahwah, NJ: Erlbaum.

Magnuson, J. S. (2015). Phoneme restoration and empirical coverage of interactive activation and adaptive resonance models of human speech processing. *Journal of the Acoustical Society of America*, *137*, 1481–1492. doi:10.1121/1.4904543

Magnuson, J. S., Mirman, D., & Harris, H. D. (2012). Computational models of spoken word recognition. In M. Spivey, K. McRae, & M. Joanisse (Eds.), *The Cambridge handbook of psycholinguistics* (pp. 76–103). Cambridge, UK: Cambridge University Press.

Magnuson, J. S., Mirman, D., Luthra, S., Strauss, T., & Harris, H. D. (2018). *Interaction in spoken word recognition models: Feedback helps*. Manuscript submitted for publication.

Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information*. San Francisco, CA: W. H. Freeman.

McClelland, J. L., & Elman, J. L. (1986). The TRACE model of speech perception. *Cognitive Psychology*, *18*, 1–86. doi:https://doi.org/10.1016/0010-0285(86)90015-0

McQueen, J. M., Cutler, A., Briscoe, T., & Norris, D. (1995). Models of continuous speech recognition and the contents of the vocabulary. *Language and Cognitive Processes*, *10*, 309–331.

Norris, D. (1994). Shortlist: A connectionist model of continuous speech recognition. *Cognition*, *52*, 189–234. doi:https://doi.org/10.1016/0010-0277(94)90043-4

Norris, D., & McQueen, J. M. (2008). Shortlist B: A Bayesian model of continuous speech recognition. *Psychological Review*, *115*, 357–395. doi:https://doi.org/10.1037/0033-295X.115.2.357

Norris, D., McQueen, J. M., & Cutler, A. (2000). Merging information in speech recognition: Feedback is never necessary. *Behavioral and*

Brain Sciences, *23*, 299–325, disc. 325–370. doi:https://doi.org/10.1017/S0140525X00003241

Oliphant, T. (2007). Python for scientific computing. *Computing in Science and Engineering*, *9*, 10–20.

Pitt, M. A., Kim, W., Navarro, D. J., & Myung, J.-I. (2006). Global model analysis by parameter space partitioning. *Psychological Review*, *113*, 57–83.

Salverda, A. P., Dahan, D., & McQueen, J. M. (2003). The role of prosodic boundaries in the resolution of lexical embedding in speech comprehension. *Cognition*, *90*, 51–89.

Strauss, T. J., Harris, H. D., & Magnuson, J. S. (2007). jTRACE: A reimplementation and extension of the TRACE model of speech perception and spoken word recognition. *Behavior Research Methods*, *39*, 19–30.

You, H., & Magnuson, J. S. (2018). *Lexical influences in spoken word recognition: Adding feedback to the Time-Invariant String Kernel model*. Manuscript in preparation.