

OPTIMALITY-THEORETIC LEARNING IN THE PRAAT PROGRAM*

Paul Boersma

Abstract

This tutorial yields a step-by-step introduction to stochastic OT grammars and about how you can use the Gradual Learning Algorithm available in the Praat program to help you rank Optimality-Theoretic constraints in ordinal and stochastic grammars.

This tutorial describes how you can draw Optimality-Theoretic tableaux and simulate Optimality-Theoretic learning with the Praat program (Boersma & Weenink 1992-2000).

1. Kinds of OT grammars

According to Prince & Smolensky (1993), an *Optimality-Theoretic* (OT) grammar consists of a number of ranked *constraints*. For every possible input (underlying form), GEN generates a (possibly very large) number of *output candidates*, and the ranking order of the constraints determines the winning candidate, which becomes the single optimal output.

In OT, ranking is *strict*, i.e., if a constraint *A* is ranked higher than the constraints *B*, *C*, and *D*, a candidate that violates only constraint *A* will always be beaten by any candidate that respects *A* (and any higher constraints), even if it violates *B*, *C*, and *D*.

— **Ordinal OT grammars.** Because only the ranking order of the constraints plays a role in evaluating the output candidates, the grammar was taken to contain no absolute ranking values, i.e., there was only an ordinal relation between the constraint rankings. For such a grammar, Tesar & Smolensky (1998) devised a learning algorithm (Error-Driven Constraint Demotion, EDCD) that changes the complete ranking order with every learning step, i.e. whenever the form produced by the learner is different from the adult form.

— **Stochastic OT grammars.** The EDCD algorithm is extremely sensitive to errors in the learning data, it cannot deal with language variation, and it does not show realistic gradual learning curves. For these reasons, Boersma (to appear; 1997; 1998: chs.14–15) proposed stochastic constraint grammars in which every constraint has a *ranking value* along a continuous ranking scale, and a small amount of *noise* is added to this ranking value at evaluation time. The associated error-driven learning algorithm (Gradual Learning Algorithm, GLA) effects small changes in the ranking values of the constraints with every learning step.

* The text of this paper is virtually identical to the OT learning tutorial and the OTGrammar manual page as available from the Help menus in the Praat program (version December 1998).

Ordinal OT grammars form a special case of the more general stochastic OT grammars: they have integer ranking values (*strata*) and zero evaluation noise. In the Praat program, therefore, every constraint is taken to have a ranking value, so that you can do stochastic as well as ordinal OT.

An OT grammar is implemented as an **OTGrammar** or **OTAnyGrammar** object. In an OTGrammar object, you specify all the constraints, all the possible inputs and all their possible outputs. This makes the OTGrammar object the simplest of the two for most problems. In this tutorial, we will only discuss OTGrammar objects.

2. The grammar

We can ask the grammar to produce an output form for any input form that is in its list of tableaux.

2.1. Viewing a grammar

Consider a language where the underlying form /pat/ leads to the surface [pa], presumably because the structural constraint NOCODA outranks the faithfulness constraint PARSE.

To create such a grammar in Praat, choose **Create NoCoda grammar** from the Optimality Theory submenu of the New menu. An **OTGrammar** object will then appear in the list of objects. If you click Edit, an **OTGrammarEditor** will show up, containing:

1. The constraint list, sorted by *disharmony* (= ranking value + noise):

	<i>ranking value</i>	<i>disharmony</i>
NOCODA	100.000	100.000
PARSE	90.000	90.000

2. The tableaux for the two possible inputs /pat/ and /pa/:

/pat/	NOCODA	PARSE
 pa		*
pat	*!	

/pa/	NOCODA	PARSE
 pa		*

From the first tableau, we see that the underlying form /pat/ will surface as [pa], because the alternative [pat] violates a constraint (namely, NOCODA) with a higher disharmony than does [pa], which only violates PARSE, which has a lower disharmony.

Note the standard OT tableau layout: asterisks (*) showing violations, exclamation marks (!) showing crucial violations, greying of cells that do not contribute to determining the winning candidate, and a finger () pointing to the winner (this may look like a plus sign (+) if you don't have the Zapf Dingbats font installed on your computer or printer).

The second tableau shows that /pa/ always surfaces as [pa], which is no wonder since this is the only candidate. All cells are grey because none of them contributes to the determination of the winner.

2.2. Inside the grammar

You can write an OTGrammar grammar into a text file by choosing **Write to text file...** from the Write menu of the Objects window. For the NOCODA example, the contents of the file will look like:

```
File type = "ooTextFile"
Object class = "OTGrammar"

2 constraints
constraint [1]: "N\s{O}C\s{ODA}" 100 100 ! NOCODA
constraint [2]: "P\s{ARSE}" 90 90 ! PARSE

0 fixed rankings

2 tableaus
input [1]: "pat" 2
  candidate [1]: "pa" 0 1
  candidate [2]: "pat" 1 0
input [2]: "pa" 1
  candidate [1]: "pa" 0 0
```

To understand more about this data structure, consult the **OTGrammar** class description (Appendix A) or click **Inspect** after selecting the OTGrammar object. The "`\s{...}`" braces ensure that the constraint names show up with their traditional small capitals (see Praat's manual page on **Text styles**).

You can read this text file into Praat again with **Read from file...** from the Read menu in the Objects window.

2.3. Defining your own grammar

By editing a text file created from an example in the New menu, you can define your own OT grammars.

As explained in Praat's manual page **Write to text file...**, Praat is quite resilient about its text file formats. As long as the strings and numbers appear in the correct order, you can redistribute the data across the lines, add all kinds of comments, or leave the comments out. For the NOCODA example, the text file could also have looked like:

```
"ooTextFile"
"OTGrammar"
2
"N\s{O}C\s{ODA}" 100 100
"P\s{ARSE}" 90 90
0 ! number of fixed rankings
2 ! number of accepted inputs
"pat" 2 ! input form with number of output candidates
  "pa" 0 1 ! first candidate with violations
  "pat" 1 0 ! second candidate with violations
"pa" 1 ! input form with number of candidates
  "pa" 0 0
```

To define your own grammar, you just provide a number of constraints and their rankings, and all the possible input forms with all their output candidates, and all the constraint violations for each candidate. The order in which you specify the constraints is free (you don't have to specify the highest-ranked first), as long as the violations are in

the same order; you could also have reversed the order of the two input forms, as long as the corresponding candidates follow them; and, you could also have reversed the order of the candidates within the /pat/ tableau, as long as the violations follow the output forms. Thus, you could just as well have written:

```

"ooTextFile"
"OTGrammar"
2
"P\s{ARSE}"          90  90
"N\s{O}C\s{ODA}"    100 100
0
2
"pa" 1
  "pa" 0 0
"pat" 2
  "pat" 0 1
  "pa" 1 0

```

2.4. Evaluation

In an Optimality-Theoretic model of grammar, *evaluation* refers to the determination of the winning candidate on the basis of the constraint ranking.

In an ordinal OT model of grammar, repeated evaluations will yield the same winner again and again. We can simulate this behaviour with our NOCODA example. In the editor, you can choose **Evaluate (zero noise)** or use its keyboard shortcut Command-0 (= Command-zero). Repeated evaluations (keep Command-0 pressed) will always yield the following grammar:

	<i>ranking value</i>	<i>disharmony</i>
NOCODA	100.000	100.000
PARSE	90.000	90.000

In a stochastic OT model of grammar, repeated evaluations will yield different disharmonies each time. To see this, choose **Evaluate (noise 2.0)** or use its keyboard shortcut Command-2. Repeated evaluations will yield grammars like the following:

	<i>ranking value</i>	<i>disharmony</i>
NOCODA	100.000	100.427
PARSE	90.000	87.502

and

	<i>ranking value</i>	<i>disharmony</i>
NOCODA	100.000	101.041
PARSE	90.000	90.930

and

	<i>ranking value</i>	<i>disharmony</i>
NOCODA	100.000	96.398
PARSE	90.000	89.482

The disharmonies vary around the ranking values, according to a Gaussian distribution with a standard deviation of 2.0. The winner will still be [pa] in almost all cases, because the probability of bridging the gap between the two ranking values is very low, namely 0.02 per cent (Boersma 1998: 332).

With a noise much higher than 2.0, the chances of PARSE outranking NOCODA will rise. To see this, choose **Evaluate...** and supply 5.0 for the noise. Typical outcomes are:

	<i>ranking value</i>	<i>disharmony</i>
NOCODA	100.000	92.634
PARSE	90.000	86.931

and

	<i>ranking value</i>	<i>disharmony</i>
NOCODA	100.000	101.162
PARSE	90.000	85.311

and

	<i>ranking value</i>	<i>disharmony</i>
PARSE	90.000	99.778
NOCODA	100.000	98.711

In the last case, the order of the constraints has been reversed. You will see that [pat] has become the winning candidate:

/pat/	PARSE	NOCODA
pa	*!	
 pat		*

However, in the remaining part of this tutorial, we will stick with a noise with a standard deviation of 2.0. This specific number ensures that we can model fairly rigid rankings by giving the constraints a ranking difference of 10. Also, the learning algorithm will separate many constraints in such a way that the differences between their ranking values are in the vicinity of 10.

2.5. Editing a grammar

In the NOCODA example, the winning candidate for the input /pat/ was always [pa].

To make [pat] the winner instead, NOCODA should be ranked lower than PARSE. To achieve this even with zero noise, go to the editor and select the NOCODA constraint by clicking on it (a spade symbol ♠ will mark the selected constraint), and choose **Edit ranking...** from the Edit menu, or use the keyboard shortcut Command-E.

In the resulting dialog, we lower the ranking of the constraint from 100 to 80, and click OK. This is what you will see in the editor:

	<i>ranking value</i>	<i>disharmony</i>
♠ NOCODA	80.000	103.429
PARSE	90.000	88.083

/pat/	NOCODA	PARSE
 pa		*
pat	*!	

Nothing has happened to the tableau, because the disharmonies still have their old values. So choose **Evaluate (noise 2.0)** (Command-2) or **Evaluate (zero noise)** (Command-0). The new disharmonies will centre around the new ranking values, and we see that [pat] becomes the new winner:

	<i>ranking value</i>	<i>disharmony</i>
PARSE	90.000	90.743
NOCODA	80.000	81.581

/pat/	PARSE	NOCODA
pa	*!	
 pat		*

2.6. Variable output

Each time you press Command-2, which invokes the command **Evaluate (noise 2.0)** from the Edit menu, you will see the disharmonies changing. If the distance between the constraint rankings is 10, however, the winning candidates will very probably stay the same.

So starting from the NOCODA example, we edit the rankings of the constraints again, setting the ranking value of PARSE to 88 and that of NOCODA to 85. If we now press Command-2 repeatedly, we will get [pat] in most of the cases, but we will see the finger pointing at [pa] in 14 percent of the cases:

	<i>ranking value</i>	<i>disharmony</i>
PARSE	88.000	87.421
NOCODA	85.000	85.585

/pat/	PARSE	NOCODA
pa	*!	
 pat		*

but

	<i>ranking value</i>	<i>disharmony</i>
NOCODA	85.000	87.128
PARSE	88.000	85.076

/pat/	NOCODA	PARSE
 pa		*
pat	*!	

As a more functionally oriented example, we consider nasal place assimilation. Suppose that the underlying sequence /an+pa/ surfaces as the assimilated [ampa] in 80 percent

of the cases, and as the faithful [anpa] in the remaining 20 percent, while the non-nasal stop /t/ never assimilates. This can be achieved by having the articulatory constraint *GESTURE ranked at a short distance above *REPLACE (n, m):

```
"ooTextFile"
"OTGrammar"
3 constraints
"*G\s{ESTURE}"           102.7 0
"*R\s{EPLACE} (n, m)"    100.0 0
"*R\s{EPLACE} (t, p)"    112.0 0
0 fixed rankings
2 tableaux
"an+pa" 2
  "anpa" 1 0 0
  "ampa" 0 1 0
"at+ma" 2
  "atma" 1 0 0
  "apma" 0 0 1
```

You can create this grammar with **Create place assimilation grammar** from the New menu. In the editor, it will often look like follows:

	<i>ranking value</i>	<i>disharmony</i>
*REPLACE (t, p)	112.000	109.806
*GESTURE	102.700	102.742
*REPLACE (n, m)	100.000	101.044

/an+pa/	*REPLACE (t, p)	*GESTURE	*REPLACE (n, m)
anpa		*!	
 ampa			*

/at+ma/	*REPLACE (t, p)	*GESTURE	*REPLACE (n, m)
 atma		*	
apma	*!		

If you keep the Command-2 keys pressed, however, you will see that the tableaux change into something like the following in approximately 20 percent of the cases:

	<i>ranking value</i>	<i>disharmony</i>
*REPLACE (t, p)	112.000	113.395
*REPLACE (n, m)	100.000	103.324
*GESTURE	102.700	101.722

/an+pa/	*REPLACE (t, p)	*REPLACE (n, m)	*GESTURE
 anpa			*
ampa		*!	

/at+ma/	*REPLACE (t, p)	*REPLACE (n, m)	*GESTURE
 atma			*
apma	*!		

We see that /at+ma/ always surfaces at [atma], because *REPLACE (t, p) is ranked much higher than the other two, and that the output of /an+pa/ is variable because of the close rankings of *GESTURE and *REPLACE (n, m).

2.7. Tableau pictures

To show a tableau in Praat's **Picture window** instead of in the editor, you select an **OTGrammar** object and click **Draw tableau...** After you specify the input form, a tableau is drawn with the current font and size at the location of the current selection (*viewport*) in the Picture window. The top left corner of the tableau is aligned with the top left corner of the selection. You can draw more than one object into the Picture window, whose menus also allow you to add a lot of graphics of your own design.

Besides printing the picture to a PostScript printer (with **Print picture to PostScript printer...**), you can save a part of it to an EPS file for inclusion into your favourite word processor (with **Write picture to EPS file...**). For this to succeed, make sure that the selection includes at least your tableau; otherwise, some part of your tableau may end up truncated.

2.8. Asking for one output

To ask the grammar to produce a single output for a specified input form, you can choose **OTGrammar: Input to output...** The dialog will ask you to provide an input form and the strength of the noise (the default is 2.0 again). This will perform an evaluation and write the result into Praat's Info window.

If you are viewing the grammar in the **OTGrammarEditor**, you will see the disharmonies change, and if the grammar allows variation, you will see that the winner in the tableau in the editor varies with the winner shown in the Info window.

Since the editor shows more information than the Info window, this command is not very useful except for purposes of scripting. See the following page for some related but more useful commands.

2.9. Output distributions

To ask the grammar to produce *many* outputs for a specified input form, and collect them in a **Strings** object, you select an **OTGrammar** and choose **Input to outputs...**

For example, select the object "OTGrammar assimilation" from our place assimilation example (§2.6), and click **Input to outputs...** In the resulting dialog, you specify 1000 trials, a noise strength of 2.0, and "an+pa" for the input form.

After you click OK, a **Strings** object will appear in the list. If you click Info, you will see that it contains 1000 strings. If you click Inspect, you will see that most of the strings are "ampa", but some of them are "anpa". These are the output forms computed from 1000 evaluations for the input /an+pa/.

To count how many instances of [ampa] and [anpa] were generated, you select the **Strings** object and click **To Distributions**. You will see a new **Distributions** object appear in the list. If you draw this to the Picture window (with **Draw as numbers...**), you will see something like:

ampa	815
anpa	185

which means that our grammar, when fed with 1000 /an+pa/ inputs, produced [ampa] 815 times, and [anpa] 185 times, which is consistent with our initial guess that a ranking difference of 2.7 would cause approximately an 80% – 20% distribution of [ampa] and [anpa].

— **Checking the distribution hypothesis.** To see whether the guess of a 2.7 ranking difference is correct, we perform 1,000,000 trials instead of 1000. The output distribution (if you have enough memory in your computer) becomes something like (set the *Precision* to 7 in the drawing dialog):

ampa	830080
anpa	169920

The expected values under the 80% – 20% distribution hypothesis are:

ampa	800000
anpa	200000

We compute (e.g. with **Calculator...**) a χ^2 of $30080^2/800000 + 30080^2/200000 = 5655.04$, which, of course, is much too high for a distribution with a single degree of freedom. So the ranking difference must be smaller. If it is 2.4 (change the ranking of *GESTURE to 102.4), the numbers become something like

ampa	801974
anpa	198026

which gives a χ^2 of 24.35. By using the Calculator with the formula `chiSquareQ(24.35, 1)`, we find that values larger than this have a probability of $8 \cdot 10^{-7}$ under the 80% – 20% distribution hypothesis, which must therefore be rejected again.

Rather than continuing this iterative procedure to find the correct ranking values for an 80% – 20% grammar, we will use the Gradual Learning Algorithm (§5) to determine the rankings automatically, without any memory of past events other than the memory associated with maintaining the ranking values.

— **Measuring all inputs.** To measure the outcomes of all the possible inputs at once, you select an **OTGrammar** and choose **To output Distributions...** As an example, try this on our place assimilation grammar. You can supply 1000000 for the number of trials, and the usual 2.0 for the standard deviation of the noise. After you click OK, a **Distributions** object will appear in the list. If you draw this to the Picture window, the result will look like:

/an+pa/ → anpa	169855
/an+pa/ → ampa	830145
/at+ma/ → atma	999492
/at+ma/ → apma	508

We see that the number of [apma] outputs is not zero. This is due to the difference of 9.3 between the rankings of *REPLACE (t, p) and *GESTURE. If you rank *REPLACE (t, p) at 116.0, the number of produced [apma] reduces to about one in a million, as you can easily check with some patience.

3. Generating language data

A learner needs two things: a grammar that she can adjust (§2), and language data.

3.1. Data from a pair distribution

If the grammar contains faithfulness constraints, the learner needs pairs of underlying and adult surface forms. For our place assimilation example, she needs a lot of /at+ma/ - [atma] pairs, and four times as many /an+pa/ - [ampa] pairs as /an+pa/ - [anpa] pairs. We can specify this language-data distribution in a **PairDistribution** object, which we could simply write into a text file:

```
"ooTextFile"  
"PairDistribution"  
4 pairs  
"at+ma"  "atma"  100  
"at+ma"  "apma"   0  
"an+pa"  "anpa"   20  
"an+pa"  "ampa"   80
```

The values seem to represent percentages, but they could also have been 1.0, 0.0, 0.2, and 0.8, or any other values with the same proportions. We could also have left out the second pair and specified “3” instead of “4” in the third line.

We can create this pair distribution with **Create place assimilation distribution** from the Optimality Theory submenu of the New menu in the Objects window. To see that it really contains the above data, you can draw it to the Picture window. To change the values, use Inspect (in which case you should remember to click Change after any change).

To generate input-output pairs from the above distribution, select the **PairDistribution** and click **To Strings...** If you then just click OK, there will appear two **Strings** objects in the list, called “input” (underlying forms) and “output” (surface forms). Both contain 1000 strings. If you Inspect them both, you can see that e.g. the 377th string in “input” corresponds to the 377th string in “output”, i.e., the two series of strings are aligned. See also the example at the manual page **PairDistribution: To Strings...**

These two Strings objects are sufficient to help an **OTGrammar** grammar to change its constraint rankings in such a way that the output distributions generated by the grammar match the output distributions in the language data. See §5.

3.2. Data from another grammar

Instead of generating input-output pairs directly from a **PairDistribution** object, you can also generate input forms and their winning outputs from an **OTGrammar** grammar. Of course, that's what the language data presented to real children comes from. Our example will be a tongue-root harmony grammar.

Choose **Create tongue-root grammar...** from the Optimality Theory submenu of the New menu. Set *Constraint set* to “Five”, and *Ranking* to “Wolof”. Click OK. An object called “OTGrammar Wolof” will appear in the list. Click **Edit**. You will see the following grammar appear in the **OTGrammarEditor**:

	<i>ranking value</i>	<i>disharmony</i>
*[rtr / hi]	100.000	100.000
PARSE (rtr)	50.000	50.000
*GESTURE (contour)	30.000	30.000
PARSE (atr)	20.000	20.000
*[atr / lo]	10.000	10.000

This simplified Wolof grammar, with five constraints with clearly different rankings, is equivalent to the traditional OT ranking

*[rtr / hi] >> PARSE (rtr) >> *GESTURE (contour) >> PARSE (atr) >> *[atr / lo]

These constraints are based on a description of Wolof by Archangeli & Pulleyblank (1994: 225–239). For the meaning of these constraints, see Boersma (1998: 295), or the **Create tongue-root grammar...** manual page, which is included in this paper as Appendix B.

For each input, there are four output candidates: the vowel heights will be the same as those in the input, but the tongue-root values of V₁ and V₂ are varied. For example, for the input [ita] we will have the four candidates [ita], [itə], [ɪta], and [ɪtə].

With this way of generating candidates, we see that the five constraints are completely ranked. First, the absolute prohibition on surface [ɪ] shows that *[rtr / hi] outranks RTR faithfulness (otherwise, [ɪti] would have been the winner):

/iti/	*[rtr / hi]	PARSE (rtr)	*GESTURE (contour)	PARSE (atr)	*[atr / lo]
ɪti	*!*				
iti	*!	*	*		
iti	*!	*	*		
 iti		**			

Second, the faithful surfacing of the disharmonic /ite/ shows that RTR faithfulness must outrank the harmony (anti-contour) constraint (otherwise, [ite] would have been the winner):

/ite/	*[rtr / hi]	PARSE (rtr)	*GESTURE (contour)	PARSE (atr)	*[atr / lo]
 ite			*		
ite	*!			*	
ite		*!			
ite	*!	*	*	*	

Third, the RTR-dominant harmonization of underlying disharmonic /etɛ/ shows that harmony must outrank ATR faithfulness (otherwise, [etɛ] would have won):

/eɛ/	*[rtr / hi]	PARSE (rtr)	*GESTURE (contour)	PARSE (atr)	*[atr / lo]
ete			*!		
 eɛe				*	
ete		*!			
eɛe		*!	*	*	

Finally, the faithful surfacing of the low ATR vowel /ə/ even if not forced by harmony, shows that ATR faithfulness outranks *[atr / lo] (otherwise, [ata] would have been the winning candidate):

/ətə/	*[rtr / hi]	PARSE (rtr)	*GESTURE (contour)	PARSE (atr)	*[atr / lo]
 ətə					**
atə			*!	*	*
ətə			*!	*	*
ata				*!*	

These four ranking arguments clearly establish the crucial rankings of all five constraints.

— **Generating inputs from the grammar.** According to Prince & Smolensky (1993), the input to an OT grammar can be *anything*. This is the idea of *richness of the base*. When doing a practical investigation, however, we are only interested in the inputs that will illustrate the properties of our partial grammars. In the case of simplified Wolof, this means the 36 possible V_1tV_2 sequences where V_1 and V_2 are any of the six front vowels i, ɪ, e, ɛ, ə, and a (see **Create tongue-root grammar...**).

A set of inputs can be generated from an **OTGrammar** object by inspecting the list of tableaux. So select the Wolof tongue-root grammar and choose **Generate inputs...** Set *Number of trials* to 100, and click OK. A **Strings** object named “Wolof_in” will appear in the list. Click **Inspect** and examine the 100 input strings. You will see that they have been randomly chosen from the 36 possible V_1tV_2 sequences as described at **Create tongue-root grammar...**:

eta, ete, eti, ite, ete, iti, eti, iti, iti, ete, ...

Thus, when asked to generate a random input, these grammars produce any of the 36 possible V_1tV_2 sequences, all with equal probability.

— **Generating outputs from the grammar.** To compute the outputs for the above set of input forms, select *both* the **OTGrammar** object *and* the input **Strings** object, and choose **Inputs to outputs...**, perhaps specifying zero evaluation noise. A new **Strings** object called “Wolof_out” will appear in the list. If you **Inspect** it, you will see that it contains a string sequence aligned with the original input strings:

eta, ete, eti, ite, ete, iti, eti, iti, iti, ete, ...

In this way, we have created two Strings objects, which together form a series of input-output pairs needed for learning a grammar that contains faithfulness constraints.

4. Learning an ordinal grammar

With the data from a tongue-root-harmony language with five completely ranked constraints, we can have a throw at learning this language, starting with a grammar in which all the constraints are ranked at the same height, or randomly ranked, or with articulatory constraints outranking faithfulness constraints.

Let's try the third of these. Create an infant tongue-root grammar by choosing **Create tongue-root grammar...** and specifying "Five" for the constraint set and "Infant" for the ranking. The result after a single evaluation will be like:

	<i>ranking value</i>	<i>disharmony</i>
*GESTURE (contour)	100.000	100.631
*[atr / lo]	100.000	100.244
*[rtr / hi]	100.000	97.086
PARSE (rtr)	50.000	51.736
PARSE (atr)	50.000	46.959

Such a grammar produces all kinds of non-adult results. For instance, the input /əti/ will surface as [ati]:

/əti/	*GESTURE (contour)	*[atr / lo]	*[rtr / hi]	PARSE (rtr)	PARSE (atr)
əti	*!	*	*		
 ati			*		*
əti		*!		*	
ati	*!			*	*

The adult form is very different: [əti]. The cause of the discrepancy is in the order of the constraints *[atr / lo] and *[rtr / hi], which militate against [ə] and [ɪ], respectively. Simply reversing the rankings of these two constraints would solve the problem in this case. More generally, Tesar & Smolensky (1998) prove that demoting all the constraints that cause the adult form to lose into the stratum just below the highest-ranked constraint violated in the learner's form (here, moving *[atr / lo] just below *[rtr / hi] into the same stratum as PARSE (rtr)), will guarantee convergence to the target grammar, *if there is no variation in the data*.

But Tesar & Smolensky's algorithm cannot be used for variable data, since all constraints would be tumbling down, exchanging places and producing wildly different grammars at each learning step. Since language data do tend to be variable, we need a gradual and balanced learning algorithm, and the following algorithm is guaranteed to converge to the target language, if that language can be described by a stochastic OT grammar.

The reaction of the learner to hearing the mismatch between the adult [əti] and her own [ati] is simply:

1. to move the constraints violated in her own form, i.e. *[rtr / hi] and PARSE (atr), up by a small step along the ranking scale, thus decreasing the probability that her form will be the winner at the next evaluation of the same input;
2. and to move the constraints violated in the adult form, namely *[atr / lo] and PARSE (rtr), down along the ranking scale, thus increasing the probability that the adult form will be the learner's winner the next time.

If the small reranking step (the *plasticity*) is 0.1, the grammar will become:

	<i>ranking value</i>	<i>disharmony</i>
*GESTURE (contour)	100.000	100.631
*[atr / lo]	99.900	100.244
*[rtr / hi]	100.000	97.086
PARSE (rtr)	49.900	51.736
PARSE (atr)	50.000	46.959

The disharmonies, of course, will be different at the next evaluation, with a probability slightly higher than 50% that *[rtr / hi] will outrank *[atr / lo]. Thus the relative rankings of these two grounding constraints have moved into the direction of the adult grammar, in which they are ranked at opposite ends of the grammar.

Note that the relative rankings of PARSE (atr) and PARSE (rtr) are now moving in a direction opposite to where they will have to end up in this RTR-dominant language. This does not matter: the procedure will converge nevertheless.

We are now going to simulate the infant who learns simplified Wolof. Take an adult Wolof grammar and generate 1000 input strings and the corresponding 1000 output strings following the procedure described in §3.2. Now select the infant **OTGrammar** and both **Strings** objects, and choose **Learn (GLA)...** After you click OK, the learner processes each of the 1000 input-output pairs in succession, gradually changing the constraint ranking in case of a mismatch. The resulting grammar may look like:

	<i>ranking value</i>	<i>disharmony</i>
*[rtr / hi]	100.800	98.644
*GESTURE (contour)	89.728	94.774
*[atr / lo]	89.544	86.442
PARSE (rtr)	66.123	65.010
PARSE (atr)	63.553	64.622

We already see some features of the target grammar, namely the top ranking of *[rtr / hi] and RTR dominance (the mutual ranking of the PARSE constraints). The steps have not been exactly 0.1, because we also specified a relative plasticity spreading of 0.1, thus giving steps typically in the range of 0.7 to 1.3.

After learning once more with the same data, the result is:

	<i>ranking value</i>	<i>disharmony</i>
*[rtr / hi]	100.800	104.320
PARSE (rtr)	81.429	82.684
*[atr / lo]	79.966	78.764
*GESTURE (contour)	81.316	78.166
PARSE (atr)	77.991	77.875

This grammar now sometimes produces faithful disharmonic utterances, because the PARSE now often outrank the gestural constraints at evaluation time. But there is still a lot of variation produced. Learning once more with the same data gives:

	<i>ranking value</i>	<i>disharmony</i>
*[rtr / hi]	100.800	100.835
PARSE (rtr)	86.392	82.937
*GESTURE (contour)	81.855	81.018
*[atr / lo]	78.447	78.457
PARSE (atr)	79.409	76.853

By inspecting the first column, you can see that the ranking values are already in the same order as in the target grammar, so that the learner will produce 100 percent correct adult utterances if her evaluation noise is zero. However, with a noise of 2.0, there will still be variation. For instance, the disharmonies above will produce [ata] instead of [ətə] for underlying /ətə/. Learning seven times more with the same data gives a reasonable proficiency:

	<i>ranking value</i>	<i>disharmony</i>
*[rtr / hi]	100.800	99.167
PARSE (rtr)	91.580	93.388
*GESTURE (contour)	85.487	86.925
PARSE (atr)	80.369	78.290
*[atr / lo]	75.407	74.594

No input forms have error rates above 4 percent now, so the child has learned a lot with only 10,000 data, which may be on the order of the number of input data she receives every day.

We could have sped up the learning process appreciably by using a plasticity of 1.0 instead of 0.1. This would have given a comparable grammar after only 1000 data. After 10,000 data, we would have

	<i>ranking value</i>	<i>disharmony</i>
*[rtr / hi]	107.013	104.362
PARSE (rtr)	97.924	99.984
*GESTURE (contour)	89.679	89.473
PARSE (atr)	81.479	83.510
*[atr / lo]	73.067	72.633

With this grammar, all the error rates are below 0.2 percent. We see that crucially ranked constraints will become separated after a while by a gap of about 10 along the ranking scale.

If we have three constraints obligatorily ranked as A >> B >> C in the adult grammar, with ranking differences of 8 between A and B and between B and C in the learner's grammar (giving an error rate of 0.2%), the ranking A >> C has a chance of less than 1 in 100 million to be reversed at evaluation time. This relativity of error rates is an empirical prediction of our stochastic grammar model.

5. Learning a stochastic grammar

Having shown that the algorithm can learn deep obligatory rankings, we will now see that it also performs well in replicating the variation in the language environment.

Create a place assimilation grammar as described in §2.6, and set all its rankings to 100.000:

	<i>ranking value</i>	<i>disharmony</i>
*GESTURE	100.000	100.000
*REPLACE (t, p)	100.000	100.000
*REPLACE (n, m)	100.000	100.000

Create a place assimilation distribution and generate 1000 string pairs (§3.1). Select the grammar and the two **Strings** objects, and learn (GLA) with a plasticity of 0.1:

	<i>ranking value</i>	<i>disharmony</i>
*REPLACE (t, p)	104.540	103.140
*REPLACE (n, m)	96.214	99.321
*GESTURE	99.246	97.861

The output distributions are now (using **OTGrammar: To output Distributions...**, see §2.9):

/an+pa/ → anpa	14.3%
/an+pa/ → ampa	85.7%
/at+ma/ → atma	96.9%
/at+ma/ → apma	3.1%

After another 10,000 new string pairs, we have:

	<i>ranking value</i>	<i>disharmony</i>
*REPLACE (t, p)	106.764	107.154
*GESTURE	97.899	97.161
*REPLACE (n, m)	95.337	96.848

With the following output distributions (measured with a million draws):

/an+pa/ → anpa	18.31%
/an+pa/ → ampa	81.69%
/at+ma/ → atma	99.91%
/at+ma/ → apma	0.09%

The error rate is acceptably low, but the accuracy in reproducing the 80% – 20% distribution could be better. This is because the relatively high plasticity of 0.1 can only give a coarse approximation. So we lower the plasticity to 0.001, and supply 100,000 new data:

	<i>ranking value</i>	<i>disharmony</i>
*REPLACE (t, p)	106.810	107.184
*GESTURE	97.782	99.682
*REPLACE (n, m)	95.407	98.760

With the following output distributions:

/an+pa/ → anpa	20.08%
/an+pa/ → ampa	79.92%
/at+ma/ → atma	99.94%
/at+ma/ → apma	0.06%

So besides learning obligatory rankings like a child does, the algorithm can also replicate very well the probabilities of the environment. This means that a GLA learner can learn stochastic grammars.

Appendix A. The OTGrammar class description

OTGrammar is a class of objects in the Praat program. An OTGrammar object contains the following attributes:

struct-list *constraints*

a list of constraints. Each constraint contains the following attributes:

string *name*

the fixed name of the constraint, for instance “PARSE”.

real *ranking*

the continuous ranking value; will change during learning.

real *disharmony*

the effective ranking value during stochastic evaluation; with a non-zero evaluation noise, this will be different from *ranking*.

struct-list *fixedRankings*

an often empty list of locally ranked pairs of constraints. Each local-ranking pair contains the following attributes:

natural *higher*

the index of the universally higher-ranked of the two constraints, a number between 1 and the number of constraints.

natural *lower*

the index of the universally lower-ranked of the two constraints.

struct-list *tableaus*

a list of tableaux. Each tableau contains the following attributes:

string *input*

the input string of the tableau. In generative phonology: the underlying form of the utterance, for example /an+pa/ or /b.ɪŋ + PAST/. In functional phonology: the perceptual specification of the utterance, for example |an+pa| or the morphological specification, for example |b.ɪŋ + PAST|.

struct-list *candidates*

a list of output candidates. Each output candidate consists of:

string *output*

the output string of the tableau. In generative phonology: the surface form of the utterance, for example [anpa] or [ampa] or [b.ɪɔ:t] or [b.ɪæŋ]. In functional phonology: the combination of the articulatory and the perceptual results, for example [anpa]-/anpa/ or [ampa]-/ampa/ or [b.ɪɔ:t]-/b.ɪɔ:t/ or [b.ɪæŋ]-/b.ɪæŋ/.

natural-list *marks*

a list of the number of violations of each constraint for this output form. If there are 13 constraints, this list will contain 13 integer numbers for each candidate.

Appendix B. The “Create tongue-root grammar...” manual page

“Create tongue-root grammar...” is a command in the New menu for creating an **OTGrammar** object with a tongue-root-harmony grammar.

These OTGrammar grammars only accept inputs of the form V_1tV_2 , where V_1 and V_2 are chosen from the six front vowels *i*, *ɪ*, *e*, *ɛ*, *ə*, and *a*. In a text field, these vowels should be typed as **i**, **\ic**, **e**, **\ep**, **\sw**, and **a**, respectively (see Praat’s **Special symbols** manual page).

The following phonological features are relevant:

	ATR	RTR
high	i	ɪ
mid	e	ɛ
low	ə	a

The resulting OTGrammar object will usually contain at least the following five constraints:

*[rtr / hi]

“do not implement [retracted tongue root] if the vowel is high.”

*[atr / lo]

“do not implement [advanced tongue root] if the vowel is low.”

PARSE (rtr)

“make an underlying [retracted tongue root] specification surface.”

PARSE (atr)

“make an underlying [advanced tongue root] specification surface.”

*GESTURE (contour)

“do not go from advanced to retracted tongue root, nor the other way around, within a word.”

This set of constraints thus comprises:

- two **grounding conditions** (Archangeli & Pulleyblank 1994), which we can see as gestural constraints;
- two **faithfulness constraints**, which favour the similarity between input and output, and can be seen as implementing the principle of maximization of perceptual contrast;
- a **harmony constraint**, which, if crucially ranked higher than at least one faithfulness constraint, forces **tongue-root harmony**.

In addition, there may be the following four constraints:

*[rtr / mid]

“do not implement [retracted tongue root] if the vowel is mid”;
universally ranked lower than *[rtr / hi].

*[rtr / lo]

“do not implement [retracted tongue root] if the vowel is low”;
universally ranked lower than *[rtr / mid].

*[atr / mid]

“do not implement [advanced tongue root] if the vowel is mid”;
universally ranked lower than *[atr / lo].

*[atr / hi]

“do not implement [advanced tongue root] if the vowel is high”;
universally ranked lower than *[atr / mid].

The universal rankings referred to are due to the *local-ranking principle* (Boersma 1998). A learning algorithm may enforce this principle, e.g., if *[rtr / hi] falls down the ranking scale, *[rtr / mid] may be pushed along.

Appendix C. Shortcuts

Besides the quite explicit learning process presented in this tutorial, in which you have to create Strings objects for underlying and surface forms, it is also possible (in Praat versions from August 1999 on) to learn directly from an OTGrammar and a PairDistribution: just select an OTGrammar object together with a PairDistribution object and click **Learn (GLA)**....

Appendix D. Further reading and more examples

Three examples have been worked out in detail in Boersma & Hayes (1999). The OTGrammar and PairDistribution files for these examples can be downloaded from <http://www.fon.hum.uva.nl/paul/gla/>. The learning regimes that we used can easily be replicated with the procedure mentioned above in Appendix B.

References

- Archangeli, Diana & Douglas Pulleyblank (1994): *Grounded phonology*. Cambridge: MIT Press.
- Boersma, Paul (1997): “How we learn variation, optionality, and probability.” *IFA Proceedings* **21**: 43–58. Downloadable from www.fon.hum.uva.nl/paul/. [equals ch. 15 of Boersma 1998]
- Boersma, Paul (1998). *Functional phonology: Formalizing the interactions between articulatory and perceptual drives*. PhD dissertation, University of Amsterdam. LOT International Series **11**. The Hague: Holland Academic Graphics. [www.fon.hum.uva.nl/paul/diss/]
- Boersma, Paul (to appear). “Learning a grammar in Functional Phonology.” In Joost Dekkers, Frank van der Leeuw & Jeroen van de Weijer (eds.) *Optimality Theory: Phonology, syntax, and acquisition*. Oxford University Press. [equals most of ch. 14 of Boersma 1998]
- Boersma, Paul & Bruce Hayes (1999): “Empirical tests of the Gradual Learning Algorithm.” Ms. Univ. of Amsterdam and UCLA. [*Rutgers Optimality Archive* **349**, ruccs.rutgers.edu/roa.html]
- Boersma, Paul & David Weenink (1992–2000): *Praat, a system for doing phonetics by computer*. Web site: www.praat.org.
- Tesar, Bruce & Paul Smolensky (1998): “Learnability in Optimality Theory.” *Linguistic Inquiry* **29**: 229–268.