evolOT
Software for simulating language evolution using Stochastic Optimality Theory

User's manual

Gerhard Jäger
http://www.ling.uni-potsdam.de/~jaeger/evolOT

November 23, 2002

Contents

1 Introduction **1**

2 The algorithms **2**

3 Operating *evolOT* **6**
3.1 The OT system and the initial frequencies 6
3.1.1 The genfile 7
3.1.2 The scriptfile 9
3.1.3 Starting the program 10
3.2 Doing an experiment 12

1 Introduction

The *evolOT* program is an implementation of the iterated "Bidirectional Gradual Learning Algorithm" (BiGLA) for Stochastic Optimality Theory (see [3]), a variant of Paul Boersma's Gradual Learning Algorithm (GLA, [1]). This manual describes how *evolOT* is operated to conduct experiments with GLA or BiGLA. The software can be freely downloaded from the URL given above. There you will also find installation instructions and further useful information.

2 The algorithms

I will only give a sketchy account of the underlying algorithms; the interested reader is referred to the cited literature and the references given there.

Paul Boersma's GLA is an algorithm for learning a Stochastic OT grammar. It maps a set of utterance tokens—a training corpus—to a grammar that describes the language from which this corpus is drawn. As a stochastic grammar, the acquired grammar makes not just predictions about grammaticality and ungrammaticality, but it assign probability distributions over each non-empty set of potential utterances. If learning is successful, these probabilities converge towards the relative frequencies of utterance types in the training corpus.

GLA operates on a predefined generator relation GEN that determines what qualifies as possible inputs and outputs, and which input-output pairs are admitted by the grammatical architecture. Furthermore it is assumed that a set CON of constraints is given, i.e. a set of functions which each assign a natural number (the "number of violations") to each element of GEN.

GLA maps these components alongside with the training corpus to a ranking of CON on a continuous scale, i.e. it assigns each constraint a real number, its *rank*. (For the interpretation of continuously ranked constraints as stochastic grammars, see the standard literature on Stochastic OT, like [2]).

At each stage of the learning process, GLA assumes a certain constraint ranking (= a probability distribution). As an elementary learning step, GLA is confronted with an element of the training corpus, i.e. an input-output pair. The current grammar of the algorithm defines a probability distribution over possible outputs for the observed input, and the algorithm draws its own output for this input at random according to this distribution. If the result of this sampling does not coincide with the observation, the current grammar of the algorithm is slightly modified such that the observation becomes more likely and the hypothesis of the algorithm becomes less likely. This procedure is repeated for each item from the training corpus.[1]

This is the pseudo-code for GLA:

Initial state All constraint values are set to the *initial value*.

for $(i := 0; i < \mathit{NumberOfObservations}; i := i + 1)$ {

> **Observation** A training datum is drawn at random from the training corpus, i.e. a fully specified input-output pair $\langle i, o \rangle$.
>
> **Generation**
>
> - For each constraint, a noise value is drawn from a normal distribution N and added to its current ranking. This yields the *selection point*.

[1] Instead of a finite training corpus, Boersma assumes a probability distribution over possible utterance types as input for the algorithm, and GLA gradually converges toward the acquired grammar. In *evolOT* the size of the training corpus is fixed by the user in advance.

- Constraints are ranked by descending order of the selection points. This yields a linear order of the constraints.
- Based on this constraint ranking, the grammar generates an output o' for the input i in standard OT fashion.

Comparison If $o = o'$, nothing happens. Otherwise, the algorithm compares the constraint violations of the learning datum $\langle i, o\rangle$ with the self-generated pair $\langle i, o'\rangle$.

Adjustment

- All constraints that favor $\langle i, o\rangle$ over $\langle i, o'\rangle$ are *promoted* by some small predefined numerical amount (“plasticity”).
- All constraints that favor $\langle i, o'\rangle$ over $\langle i, o\rangle$ are *demoted* by the plasticity value.

}

The algorithm contains several parameters that have to be set by the user in *evolOT*:

- the number of observations
- the plasticity value
- the initial ranking of the constraints
- the “noise”, i.e. the standard deviation of the normal distribution N (its mean is assumed to be 0)

In *evolOT*, the training corpus is not directly supplied by the user. Instead, the user defines a frequency distribution over GEN, and the actual training corpus is generated by a random generator interpreting the relative frequencies as probabilities.

BiGLA, the bidirectional version of GLA, differs from that in two respects. First, during the generation step the algorithm generates an optimal output for the observed input on the basis of a certain constraint ranking. It is tacitly assumed that “optimal” here means “incurring the least severe pattern of constraint violations” in standard OT fashion. In BiGLA it is instead assumed that the optimal output is selected from the set of outputs from which the input is *recoverable*. The input is recoverable from the output if among all inputs that lead to this output, the input in question incurs the least severe constraint violation profile (i.e. we apply interpretive optimization). If there are several outputs from which the input is recoverable, the optimal one (in the standard sense) is selected. If recoverability is impossible, the unidirectionally optimal output is selected.

This modification can be called “bidirectional evaluation”. Besides BiGLA involves *bidirectional learning*. This means that BiGLA both generates the optimal output for the observed input, and the optimal input for the observed output. “Comparison” and “adjustment” apply both to inputs and outputs as well. Thus the pseudo-code for BiGLA is given like this:

Initial state All constraint values are set to the *initial value*.

for $(i := 0; i < \mathit{NumberOfObservations}; i := i + 1)$ {

Observation A training datum is drawn at random from the training corpus, i.e. a fully specified input-output pair $\langle i, o\rangle$.

Generation

- For each constraint, a noise value is drawn from a normal distribution N and added to its current ranking. This yields the *selection point*.
- Constraints are ranked by descending order of the selection points. This yields a linear order of the constraints.
- Based on this constraint ranking, the grammar generates an optimal output o' for the input i and an optimal input i' for the output o using bidirectional evaluation.

Comparison If $i = i'$ and $o = o'$, nothing happens. Otherwise, the algorithm compares the constraint violations of the learning datum $\langle i, o\rangle$ with the self-generated pairs $\langle i, o'\rangle$ and $\langle i', o\rangle$.

Adjustment

- All constraints that favor $\langle i, o\rangle$ over $\langle i, o'\rangle$ are *promoted* by some small predefined numerical amount (“plasticity”).
- All constraints that favor $\langle i, o\rangle$ over $\langle i', o\rangle$ are *promoted* by some small predefined numerical amount (“plasticity”).
- All constraints that favor $\langle i, o'\rangle$ over $\langle i, o\rangle$ are *demoted* by the plasticity value.
- All constraints that favor $\langle i', o\rangle$ over $\langle i, o\rangle$ are *demoted* by the plasticity value.

}

evolOT allows to choose between uni- and bidirectional evaluation, and uni- vs. bidirectional learning independently. So it actually implements four different learning algorithms, GLA, BiGLA, and two mixed versions.

Depending on the OT system that is used, the training corpus and the chosen parameters, the stochastic language that is defined by the acquired grammar may deviate to a greater or lesser degree from the training language. Especially for BiGLA this deviation can be considerable. (It is perhaps misplaced to call BiGLA a “learning” algorithm; it rather describes a certain adaptation mechanism.) If a sample corpus is drawn from this language and used for another run of GLA/BiGLA, the grammar that is acquired this time may differ from the previously learned language as well.

Such a repeated cycle of grammar acquisition and language production has been dubbed the *Iterated Learning Model* of language evolution by Kirby and Hurford [4]. It is schematically depicted in figure 1.

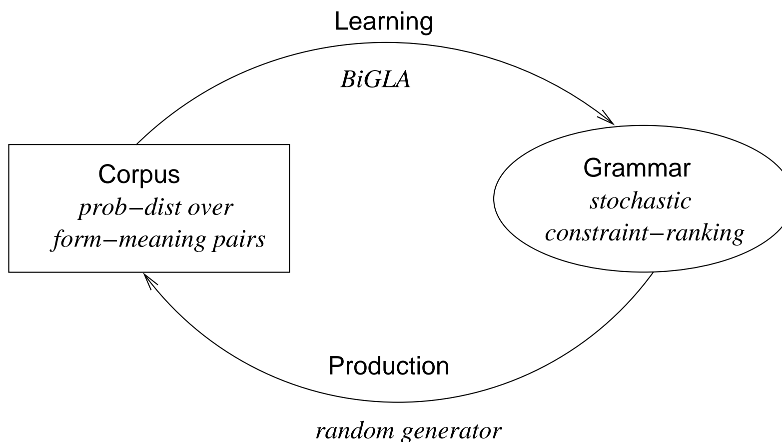

Figure 1: The Iterated Learning Model

The production half-cycle involves the usage of a random generator to produce a sample corpus from a stochastic grammar. In the *evolOT* implementation, we assume that this sample corpus has the same absolute size than the initial corpus. Furthermore we assume that the absolute frequencies of the different *inputs* are kept constant in each cycle. What may change from cycle ("generation") to cycle are the relative frequencies of the different outputs for each input. (I assume that the relative input frequencies are determined by extra-grammatical factors, and it is one of the main objectives of *evolOT* to model the interdependence between these factors and grammar.)

Formally put, the initial training corpus defines a frequency $\#(i)$ for each possible input i by

$$\#(i) \doteq \sum_{o} \#(\langle i, o \rangle)$$

where $\#(\langle i, o \rangle)$ is the number of occurrences of the utterance type $\langle i, o \rangle$ in the initial corpus. Furthermore, a given stochastic grammar G defines a probability distribution $p_G(\cdot|i)$ over the possible outputs o for each input i. Using this notation, the pseudo-code of the algorithm simulating the production step of the Iterated Learning Model can be formulated as in figure 2. I assume that there are finitely many possible inputs and outputs, which can be enumerated by as i_n, o_m etc. "*NewCorpus*" is a two-dimensional array representing the frequency distribution of the generated corpus. This means that $NewCorpus[k][l]$ is an integer representing the frequency of the pair $\langle i_k, o_l \rangle$ in the generated corpus.

One cycle of learning and production represents one generation in the evolutionary process that is simulated by *evolOT*. This cycle my be repeated arbitrarily many times, i.e. over an arbitrary number of generations (which is to be fixed by the user).

$\forall k, l$: *NewCorpus*$[k][l] := 0$
for $(k := 0; k <$ *NumberOfInputs*$; k := k + 1)$ {

  for $(l := 0; l < \#(i_k); l := l + 1)$ {

  - Draw an output o_n at random from the probability distribution $p_G(\cdot|i_k)$;
  - *NewCorpus*$[k][n] :=$ *NewCorpus*$[k][n] + 1$;

  }

}

Figure 2: Language production algorithm

3 Operating *evolOT*

To conduct an experiment, the user has to fix two components:

1. the OT system, consisting of
   - a GEN relation
   - a system of constraints
2. an initial training corpus
3. the parameters of the experiment:
   - the plasticity of the learning algorithm,
   - the initial value of the constraints,
   - the noise value (the standard deviation of the normal distribution from which the noise variable is drawn),
   - the mode of evaluation (bidirectional or unidirectional)
   - the mode of learning (bidirectional or unidirectional).

Furthermore *evolOT* offers a choice between simulating a single learning process or a sequence of generations (an evolution). In the former case, snapshots of the provisional grammar of the learner are taken, and the *number of observations between two snapshots* has to be fixed by the user. This parameter is called *StepSize*. If the experiment simulates evolution, only the results of an entire learning process are recorded, and the user has to fix the *number of generations* in advance.

3.1 The OT system and the initial frequencies

The OT system and the initial corpus are stored in to ASCII files that the user has to edit in some text editor by hand.

3.1.1 The genfile

The first file (called the “genfile” henceforth) determines both the GEN relation of the OT system and the initial corpus. Suppose that GEN admits *NInputs* many inputs and *NOutputs* many outputs. The genfile consists of a 2×2 chart with *NInputs* many rows and *NOutputs* many columns. It is assumed that the inputs and outputs are numbered consecutively. Thus each row represents some input, each column an output, and hence each cell a candidate.

If GEN disallows combination of input i with output o, the cell $\langle i, o \rangle$ has the entry -1. Otherwise the cells are filled with non-negative integers. They represent the frequency of the respective candidate in the initial corpus. (Note the difference between '0' and '-1': both means that the corresponding candidate does not occur in the initial corpus, but in the former case this is just by chance, while the candidate does not exist in the grammatical system in the latter case.)

Each row of this chart is written as a line in genfile (ending with a line break). The cells are separated by tab stops. Empty lines are ignored. Likewise, percent signs '%' and everything following on the same line are ignored. This can be used to add comments to the genfile.

To take an example, suppose our language comprises two meanings (=inputs) in total, $m1$ and $m2$, and three forms, $f1$, $f2$, and $f3$. $f1$ is unambiguously interpreted as $m1$, $f2$ as $m2$, and $f3$ is ambiguous. Suppose furthermore that $m1$ has been expressed 100 times and $m2$ 200 times in in the training corpus, and that all form alternatives for a given meaning occur equally often. The corresponding genfile might look as follows:

```
%f1     f2      f3
%---------------------------
50      -1      50      % m1
-1      100     100     % m2
```

It is possible that a GEN relation consists of several sub-relations that are not connected with each other. The following genfile would be an example:

```
1000    1000    -1      -1
-1      -1      340     12
2000    400     -1      -1
-1      -1      1000    1200
```

In such a case it is possible to write the genfile in a more compressed form. In the default case, every input (= row) competes with every other input (when doing interpretive optimization). In the example above though, the first row only really competes with the third row, and the second with the fourth. In this case it is possible to omit all the holes in GEN (the -1s), and to specify that two rows only compete if the difference between their numbers is a multiple of 2. This parameter is called *NumberOfEquivalenceClasses*. If it set to 2, the above genfile can be simplified to

```
1000    1000
340     12
```

```
2000     400
1000     1200
```

Seen from the perspective of the program, the very same genfile means different things depending on the value of *NumberOfEquivalenceClasses*. Consider the following genfile:

```
1       2
3       4
5       6
7       8
9       10
11      12
13      14
14      16
17      18
19      20
21      22
23      24
```

If *NumberOfEquivalenceClasses* = 2, this is equivalent to the following genfile (with *NumberOfEquivalenceClasses* = 2):

```
1       2       -1      -1
-1      -1      3       4
5       6       -1      -1
-1      -1      7       8
9       10      -1      -1
-1      -1      11      12
13      14      -1      -1
-1      -1      14      16
17      18      -1      -1
-1      -1      19      20
21      22      -1      -1
-1      -1      23      24
```

If *NumberOfEquivalenceClasses* = 3, it is read as

```
1       2       -1      -1      -1      -1
-1      -1      3       4       -1      -1
-1      -1      -1      -1       5      6
7       8       -1      -1      -1      -1
-1      -1      9       10      -1      -1
-1      -1      -1      -1      11      12
13      14      -1      -1      -1      -1
-1      -1      14      16      -1      -1
-1      -1      -1      -1      17      18
```

```
19      20      -1      -1      -1      -1
-1      -1      21      22      -1      -1
-1      -1      -1      -1      23      24
```

3.1.2 The scriptfile

The generator defines a set of candidates (which can be identified by the index of the input and the index of the output). Each constraint is a function that assigns a non-negative integer to each candidate, i.e. to each cell in a *NInputs*×*NOutputs* matrix. The *scriptfile* defines these functions.

The scriptfile is an ASCII file as well that the user has to edit by hand in some ASCII editor. Each line of the file contains one *command*. (As in the genfile, empty lines and comments starting with '%' are ignored.) A command consists of three components:

1. the constraint name
2. the candidate description
3. the number of violations

The *constraint name* is any arbitrary ASCII string not containing white spaces (space, newline, tab stop, EOF), the percent sign '%', the colon ':', and the semicolon ';'. The *candidate description* is a boolean formula describing a set of candidates. Its syntax is described below. The *number of violations* is some positive integer. It is optional; if it is omitted, 1 is assumed as the default value. Except in the middle of the constraint name, any number of spaces and tab stops may be used.

The constraint name is separated from the candidate description by a colon, and the the candidate description ends in a semicolon. Schematically:

<*constraint name*> : <*candidate description*>; (<*number of violations*>)

A command is to be read procedurally. Per default, the system assumes that each constraint violates each candidate 0 time. A command updates CON: the number of violations incurred by the constraint with the name *constraint name* on each candidate meeting the description *description* is increased by *number of violations*.

A *constraint description* is a boolean combination of atomic formulas. An *atomic formula* consists of a *left hand side*, a *comparison operator*, and a *right hand side*. The left hand side is either the letter `i` (for "input") or `o` (output). Comparison operators are `<, <=, >, >=, ==, !=`. They are interpreted in the obvious way. The right hand side is non-negative integer.

Atomic formulas are interpreted in the obvious way. For instance, `i <= 3` is true off the first two rows, `o != 2` is true of all but the second column etc.

Atomic formulas can be combined to larger formulas by means of the boolean operators `!` (negation), `&` (conjunction), `|` (disjunction), and `->` (implication). Following the syntactic conventions in propositional logic, `!` has the highest precedence, followed by `&`, `|`, and `->` (in that order). Parentheses can be used where necessary, and redundant parentheses are admitted as well.

Commands thus look like these examples:

```
*STRUC: o < 7; 3
*STRUC: o != 3 & o != 6 & o != 9;

FAITH: i < 5 & o > 3 & o < 7; 2
FAITH: i < 5 & (o == 1 | o == 4 | o == 7); 5
FAITH: i > 4 & o < 4;
FAITH: i > 4 & (o == 2 | o == 5 | o == 8);
```

Note that it is licit to have several commands starting with the same constraint name. (It isn't even necessary to write them into one block.) They are applied consecutively to update the same constraint.

To return to the example started above, suppose we have a constraint saying "Express $m1$ as $f1$!", and another constraint that requires "Express $m1$ as $f3$!". Likewise, there are constraints "Express $m2$ as $f2$!", and another constraint that requires "Express $m2$ as $f3$!". Let us finally assume that $f1$ is more complex than $f2$, which is in turn more complex than $f3$. This is implemented by a constraint "Avoid complexity" that is violated once by each candidate having $f2$ as output and twice by each candidate having $f3$ as output. A scriptfile describing this constraint systems might look as follows:

```
1=>1: !(i == 1 -> o == 1);
1=>3: !(i == 1 -> o == 3);
2=>2: !(i == 2 -> o == 2);
2=>3: !(i == 2 -> o == 3);
*C:   o > 1;
*C:   o > 2;
```

3.1.3 Starting the program

Once the genfile and the scriptfile are in place, it is time to start *evolOT*. Change to the directory `evolot/bin` and type `evolot&` at the command prompt. The window shown in figure 3 will appear.

Here the user can specify the numerical parameters of the OT system to be used—the number of inputs and outputs, the number of constraints, and the number of equivalence classes. (If these parameters do not fit together with the genfile or the scriptfile, you may get an error message, or the outcome of the experiment is just nonsense.)

After setting these parameters, activate the tab "Files". The mask shown in figure 4 appears.

Input the names of the genfile and the scriptfile; either by typing in their filenames in the first two lines of the mask, or by clicking at the corresponding buttons and selecting the files with the mouse.

In the next step, the commands from the scriptfile are executed and the results are stored in another file called *starfile*. It holds the numbers of constraints violations of each candidate for each constraint (i.e. the number of stars for each candidate/constraint in a traditional OT tableau). Choose a name for this file, type it into the third line of the mask, and push the button "Compile". The scriptfile is compiled into the starfile. This

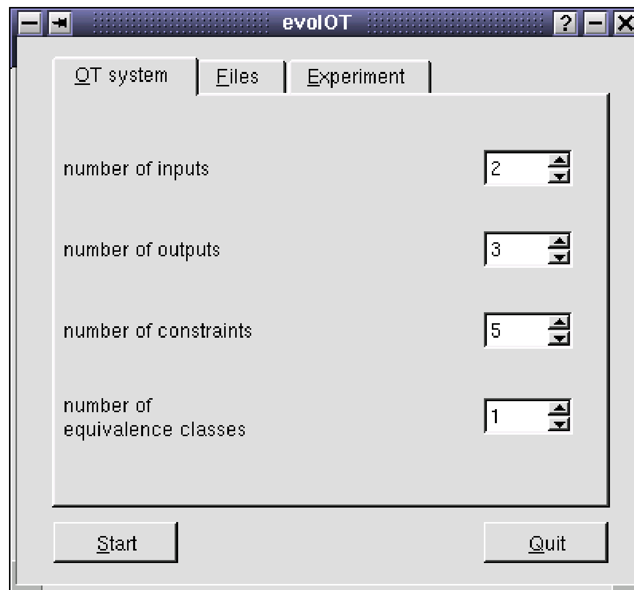

Figure 3: *evolOT*: OT system

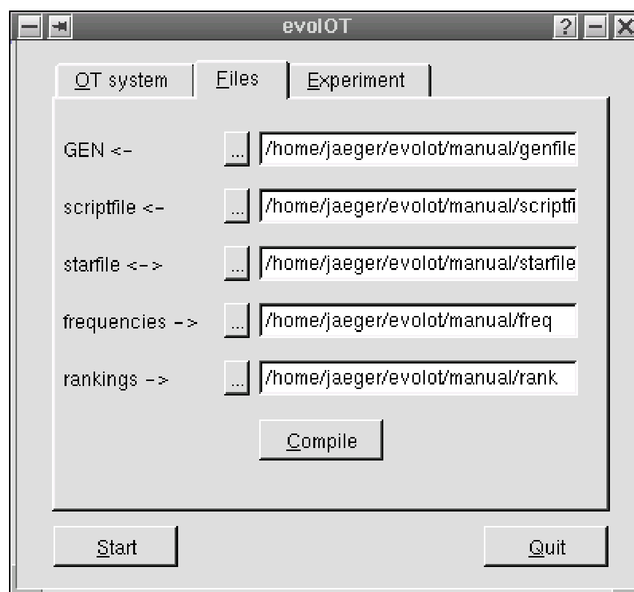

Figure 4: *evolOT*: Files

is again an ASCII file, consisting of a sequence of *NInputs*×*NOutputs*-matrixes, one per constraint. Each cell contains a natural number, namely the number of violations this constraint incurs on this candidate according to the specifications in the scriptfile. Each of these matrixes is preceded by a line starting with '%', a space, and the name of the constraint. For the sample scriptfile given above, the corresponding starfile would be

```
% 1=>1
0        1        1
0        0        0


% 1=>3
1        1        0
0        0        0


% 2=>2
0        0        0
1        0        1


% 2=>3
0        0        0
1        1        0


% *C
0        1        2
0        1        2
```

You may edit the starfile by hand. It is also possible to do without a scriptfile altogether and start a starfile from scratch. Be warned though: errors are much easier to spot and to fix in scriptfiles than in starfiles!

3.2 Doing an experiment

evolOT starts with an initial training corpus and produces a sequence of generated corpuses and constraint rankings. These sequences are stored the *frequencyfile* and the *rankingsfile*. Fix the names of these files in the fourth and fifth line of the mask.

A *frequencyfile* consists of a sequence of tables in the format of a genfile, each preceded by a line indicating the number of the generation. For our running example, the first few lines of the frequencyfile might look like

```
: 1
62       -1       38
```

```
-1      100     100

: 2
65      -1      35
-1      115     85

: 3
75      -1      25
-1      112     88

: 4
73      -1      27
-1      103     97

: 5
81      -1      19
-1      104     96
```

Each line of the rankingsfile contains the ranking of the constraints after completing the learning algorithm in each generation, if an evolution experiment is done. In a learning experiment, it contains the intermediate state of the learner after a number of observations that is a multiple of the parameter *StepSize*.

The first few lines of the rankingsfile corresponding to the frequencyfile given above (which was taken from an evolution experiment) are

```
-6.6    19.82   -7.36   -5.86   -13.96
-3.97   13.02   -4.97   -4.08   -8.94
-1.68   7.66    -3.34   -2.64   -5.02
-0.57   4.91    -2.72   -1.62   -3.29
-1.27   7.53    -3.68   -2.58   -4.95
```

If all slots at the “Files”-mask are filled, change to “Experiment”. You will see the following form:

In the first line you can decide between the types of experiment, *Evolution* or *Learning*. If you choose *Learning*, you can fix a *StepSize*. If you choose *Evolution*, you can instead fix a the number of generations. Furthermore you can determine the number of observations per generation, the initial ranking of the constraints at the beginning of a learning cycle, the noise, and the plasticity. (All these parameters have sensible default values.) Finally, there are two checkboxes to decide between bidirectional and unidirectional evaluation, and between bidirectional and unidirectional learning.

Start the experiment by pushing the “Start” button. A new window containing (as in figure 3.2) a table will be opened, that displays the frequency distributions of the input-output combinations generation after generation.

Additionally a progress bar (as in figure 3.2) is opened. By clicking at the “Cancel” button you can stop an experiment any time.

During an experiment the main window will not react to mouse or keyboard input.

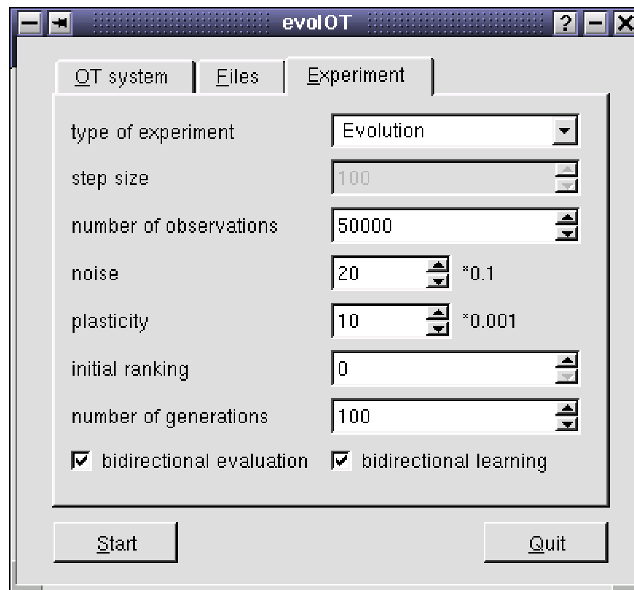

Figure 5: *evolOT*: parameters of an experiment

evolot

	1	2	3
1	99	-1	1
2	-1	108	92

Figure 6: Display of intermediate results: frequencies

Figure 7: Progress bar

To visualize the content of the rankings file, type in `otmovie` at the command prompt, followed by the name of the starfile and the name of the rankingsfile:

`otmovie` <starfile> <rankingsfile>

If *gnuplot* is installed on your system (and the executable is in your search path), another window will open that displays a Cartesian diagram. (See figure 8)

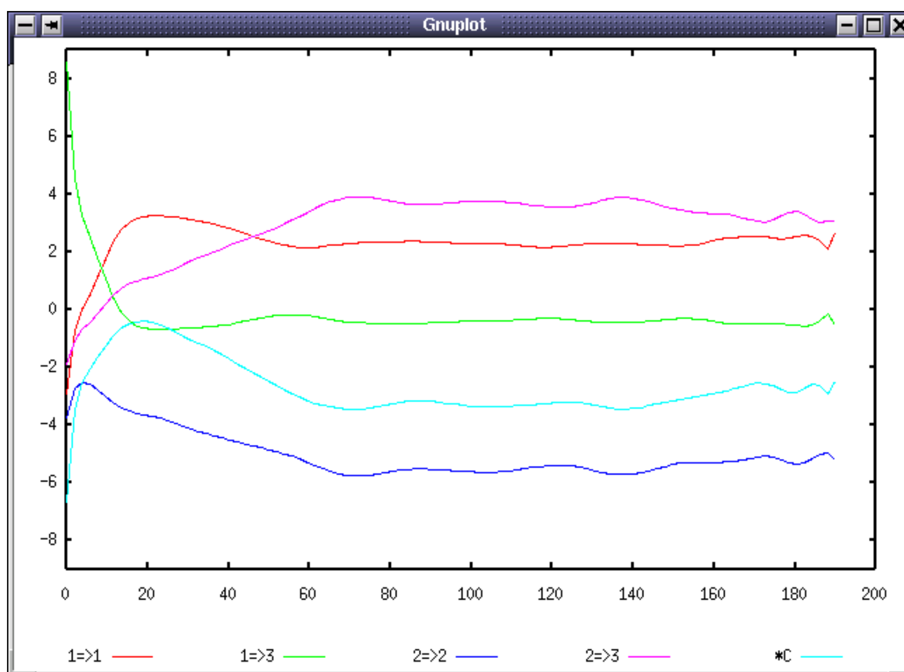

Figure 8: Display of intermediate results: rankings

The horizontal axis contains the generations/observations, and the vertical axis the rankings of the constraints. The development of the constraint rankings are plotted as function curves, with a different color for each constraint. Your desktop will look like figure 9.

The graphic is updated every two seconds.[2] To stop this display, activate the terminal window and type `Ctrl-C`.

If you want to visualize the development of the constraint rankings after completion of the experiment, you can produce a static display with

`r2x11` <starfile> <rankingsfile>

Both the dynamic and the static display restricts the range of the vertical axis from -9 to 9. This is sometimes insufficient. If you need a larger range, type

`r2x11_autoscale` <starfile> <rankingsfile>

[2]Therefore this window will alway be active, and if you close or iconify it, it will reappear after at most two seconds.

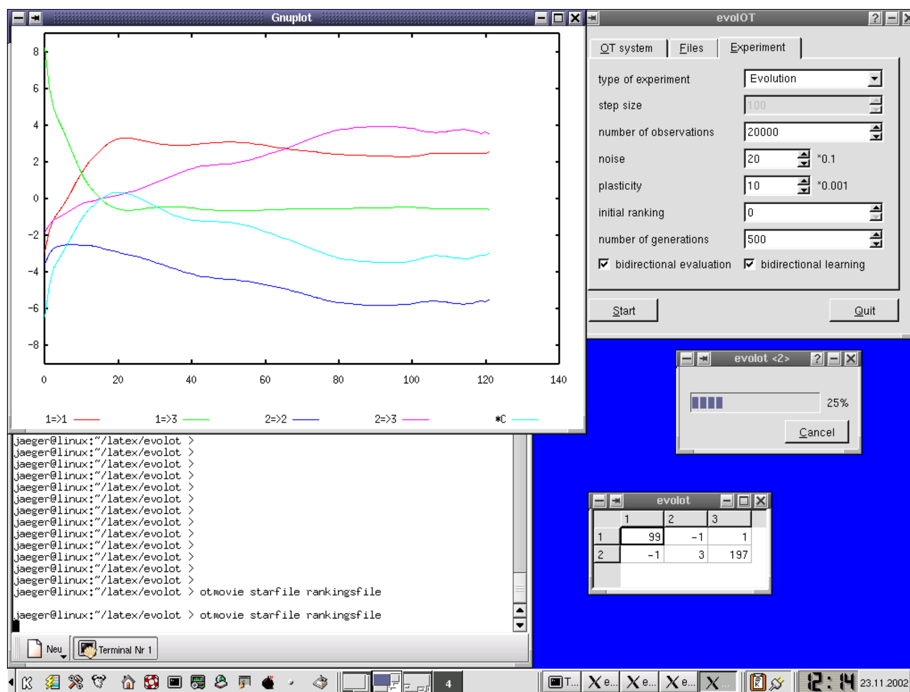

Figure 9: During an experiment

To safe the graphics in an .eps file, type

r2eps <starfile> <rankingsfile> <epsfile>

This will produce a black-and-white .eps file, where line colors are replaced by different styles of dotted lines. To get a colored .eps file, use the command.

cl_r2eps <starfile> <rankingsfile> <epsfile>

Likewise, you can safe the graphics in a .fig file (the graphics format of the Unix program *xfig*) by using

r2fig <starfile> <rankingsfile> <figfile>

for black-and-white, and

cl_r2fig <starfile> <rankingsfile> <figfile>

for colored output. Of course you can also use the rankingsfile with any spreadsheet program to visualize the results.

References

[1] Paul Boersma. *Functional Phonology*. PhD thesis, University of Amsterdam, 1998.

[2] Paul Boersma and Bruce Hayes. Empirical tests of the gradual learning algorithm. *Linguistic Inquiry*, 32(1):45–86, 2001.

[3] Gerhard Jäger. Learning constraint sub-hierarchies. The Bidirectional Gradual Learning Algorithm. manuscript, University of Potsdam, 2002.

[4] Simon Kirby and James R. Hurford. The emergence of linguistic structure: An overview of the Iterated Learning Model. manuscript, University of Edinburgh, 2001.