TR—I—0069

Tools for Fundamental Frequency Modelling

William J. Poser

March 3, 1989

Abstract

Numerical modelling of fundamental frequency (F0) is hampered by the lack of suitable tools. This memorandum describes two tools intended to facilitate the exploration of numerical models of F0 and other phrase-level phenomena. One is a modelling program with some unusual characteristics, including input in the form of a description of the prosodic tree, run-time control over parameter settings and rule application, and provision of analysis-by-synthesis facilities that allow analysis-by-synthesis to be carried out using arbitrary mixtures of any utterance types for which the model is defined. The second is an interactive interface to the model that allows it to be manipulated using a programmable time-series editor. This allows the user to see immediately the result of changing parameters and enabling and disabling rules, and also allows easy visual comparison of computed values with data.

CONTENTS

1. Introduction

Numerical modelling of fundamental frequency (F0) is becoming increasingly important not only for work in areas like speech synthesis, where numerical models are essential, but also for more theoretically oriented work in linguistics. However, this work is hampered by the lack of suitable tools, without which the study of a wide range of F0 models, especially models that depend crucially on the structure of the utterance, is time-consuming and tedious.

This memorandum describes two tools intended to facilitate the exploration of numerical models of F0 and other phrase-level phenomena. One is a modelling program with some unusual characteristics, including input in the form of a description of the prosodic tree, run-time control over parameter settings and rule application, and provision of analysis-by-synthesis facilities that allow analysis-by-synthesis to be carried out using arbitrary mixtures of any utterance types for which the model is defined. The second is an interactive interface to the model that allows it to be manipulated using a programmable time-series editor. This allows the user to see immediately the result of changing parameters and enabling and disabling rules, and also allows easy visual comparison of computed values with data.

2. A Flexible F0 Modelling Program

Computing F0 contours for utterances of fixed structure, using a fixed F0 model, is easy. Even if the F0 model is fairly complex, with many parameters and complex equations, implementing it is a matter of simple numerical programming. However, such simple models are difficult to use for studying F0. A program that can handle only a fixed utterance type is of little use if intonation depends on the structure of the utterance. A program containing a model that is completely fixed must be edited in order to try out different models. In order to make it easier to study a variety of different F0 models, we have implemented an F0 modelling program that provides an unusual degree of flexibility.

One special property is that it accepts as input a description of the utterance in an extremely general format, so that, for a very wide range of possible models, no change either in the input module or the input files is necessary when the model itself is changed — it is only necessary to change the model subroutine and how it uses the information about the structure of the utterance.

Provision is also made for a degree of run-time configuration of the model. Moreover, the program is written in such a way that the actual model code is very much isolated, so that the user can change the F0 model by replacing a single source file, without knowing anything about the bulk of the program.

Finally, the program provides for the model to be fit to data using arbitrary mixtures of any type of data for which the model is defined.

2.1. Tree Descriptions

Much research on intonation in a variety of languages shows us that fundamental frequency is determined not only by lexical information and segmental perturbations but by the prosodic structure of the utterance, where the prosodic structure is a hierarchical phrasal structure similar to, but different from, syntactic structure (Selkirk 1984).

Consequently, in order to represent this information as directly as possible, the program takes as input a file containing a description of a tree structure. The file describes the type (level in the prosodic hierarchy) of each node and names its daughters. It also allows attribute-value pairs to be associated with each node. For example, the following file describes an utterance containing two minor phrases, each three moras long and unaccented, contained within a single major phrase.[1]

```
S utterance (M1)
M1 "major phrase" (m1,m2)
m1 "minor phrase" (NIL)
m2 "minor phrase" (NIL)
m1 morae 3
m2 morae 3
m1 accent 0
m2 accent 0
```

The first line says that the node named *S* is of type *utterance* and has one daughter, *M1*. The next line says that the node named *M1* is a *major phrase* and has two daughters, *m1* and *m2*. The next two lines indicate that each of the nodes *m1* and *m2* is a *minor phrase* and has no daughters. The last four lines associate attribute-value pairs with the two minor phrase nodes. In both cases the number of morae is set to three and the accent position is set to zero (i.e. there is no accent).

The program reads the tree description from the file and constructs the tree in memory, associating with each node its own attribute list. The subroutine that actually computes the F0 contour is passed a pointer to the tree structure, so that it has available all of the information associated with the tree.

This input format allows us to interpret theoretical notions about prosodic structure directly. It also provides an input format that is independent of the details of the F0 model, and whose similarity to syntactic structure facilitates studies of the relationship between syntactic structure and prosodic structure.

2.2. Model Descriptions

Although some aspects of the model are fixed by the underlying model program, the model may be configured to some extent at run time. The

[1] In this file, as in all of the other files read by the model, blank lines and anything on a line following a cross-hatch character(#) are ignored. The treatment of a cross-hatch character as a comment indicator may be disabled by preceding it with a backslash (/).

model configuration is read from the model definition file, whose name is specified on the command line. Model parameters may be set by including lines whose first field is the keyword *set* followed by the name of the parameter to be set and its value, e.g.:

```
set DsRatio 0.9
```

which sets the value of the parameter *DsRatio* to 0.9. If the model is used for analysis-by-synthesis and the named parameter is estimated, the value specified in the model definition file is used as the initial value.

If the model contains rules whose application is conditional, by default these rules are disabled. To cause a rule to be applied it is necessary to include in the model definition file a line beginning with the keyword *apply* followed by the name of the rule to apply, e.g.:

```
apply Downstep
```

which causes the *Downstep* rule to be applied when the model is run.

2.3. Analysis-by-Synthesis

Any or all of the model parameters may be estimated by analysis-by-synthesis, using data from utterances of any type for which the model is defined. The analysis-by-synthesis algorithm used is essentially the same as that of Miyatake & Sagisaka (1988). At each iteration, it loops over the parameters to be estimated and attempts to optimize each parameter in turn, where optimization means minimization of the total distance between the predictions of the model and the data. Optimization is performed by following the gradient at a quantization given by the current step size for the given parameter. When the current parameter value is no worse than those one step to either side, the algorithm halves the step size for that parameter and moves on to the next parameter on the list. When it has optimized all of the parameters on the list, it checks to see whether it should proceed to the next iteration.

Information about how to carry out the analysis-by-synthesis process and what data to use is provided in the *ABS control file*, whose name is specified on the command line. Lines may begin with one of the keywords: *estimate*, *iterations*, *norm*, *step* or *threshold*, in which case they are interpreted as follows:

estimate ParameterName

> Estimate the value of the parameter ParameterName during the ABS run. If a parameter is not specified as subject to estimation it is fixed.

iterations IterationCount

> Iterate for a minimum of IterationCount iterations. The default is 25.

norm NormName

> Use the named norm as the distance function. The legal values are H1_norm and L2_norm. The H1_norm the sum of the difference of the two values and

the difference of their first derivatives, i.e., if r and s are vectors of length N, their root-mean-square H1 distance is: $d(r,s) = [(\sum_{i=0}^{i=N}[r(i) - s(i)]^2 + [dr(i)/dt - ds(i)/dt]^2)/N]^{1/2}$. The first derivative is approximated by computing the first left difference (i.e. $df/dt|_i \simeq f(i) - f(i-1)$). The L2_norm is the usual Euclidean distance. The default is the L2_norm.

step ParameterName StepSize

Set the initial search step for the parameter ParameterName to StepSize. The defaults are dependent on the model.

threshold Threshold

If the ABS process is converging sufficiently quickly, only the minimum number of iterations are carried out. If the difference in the error between two iterations is larger than Threshold convergence is taken to be going slowly and the number of iterations to be carried out is increased. The default value is 0.0.

All other lines are taken to be data specifications and consist of the name of the data file, the number of tokens in the data file, and the name of the corresponding tree description file.

There are no constraints on the order in which the elements of an ABS control file are given. Keywords may be given in any order and may be freely intermixed with data file specifications. However, in order to allow the user to control the order in which parameters are estimated, which can make a difference to the ABS process, parameters are estimated in the order in which they are named in lines with the keyword *estimate*.

Here is a typical ABS control file:

```
# Parameters to estimate
estimate DsRatio
estimate DecSlope
estimate Range
estimate Base
estimate Beta

# Step sizes
step Range 5.0
step Base 5.0
step DsRatio 0.1
step DecSlope 1.0
step Beta 0.1

# ABS control parameters
iterations 25
threshold 0.5
norm H1_norm

# Data files
corfiles/takeda2_001.cor 4 treefiles/takeda2_001.tree
corfiles/takeda2_002.cor 7 treefiles/takeda2_002.tree
corfiles/takeda2_003.cor 6 treefiles/takeda2_003.tree
corfiles/takeda2_004.cor 8 treefiles/takeda2_004.tree
```

The ABS data files are text files containing one line for each token. Each line consists of data values separate by whitespace (spaces and/or tabs). A well-formed data file therefore looks like this:

```
117   153   127   93   85
123   139   107   83   80
123   142   115   93   86
96    141   126   96   82
```

The tree description files for ABS data have exactly the same format as the tree description files used to drive the model when used to generate synthetic F0 contours.

During an ABS run the model reports on its progress by writing information on the standard error output. It makes a report at the beginning of each iteration, reporting the time, the iteration number, the current value of the distance, and the change in the distance since the previous iteration.

More detailed information is written in a log file, whose name is the name by which the program is called followed by the suffix *.abslog*, e.g. *model.abslog*. An entry is made in the log file at the beginning of each iteration. Each entry contains the same information as is written on the standard error output (the time, the iteration count, the distance, and the change in the distance), together with the values of the parameters being estimated and the step size for each parameter. At the end of the ABS run summary information is generated, including the time at which the ABS run terminated and the iteration count at termination, the number of data points used, and the final error. The L2-norm error is always given, as well as the H1-norm if that was used as the distance for the ABS process. This information is followed by the final values of all model parameters together with an indication of whether the parameter value was obtained by ABS or was fixed.

An extract from a typical ABS log file follows. This log is from a run of a simple pause duration model with two parameters, *MajorPause* (the length of a pause after a major phrase) and *MinorPause* (the length of a pause after a minor phrase), in which the value of MajorPause was fixed and the value of MinorPause was estimated. Notice that the first value of *delta* is negative since prior to the first iteration the model assumes a distance of zero.

```
Mar 3 13:30:18 iteration 0 delta = -162738.6197 distance = 162738.6197
        MinorPause = 12.000 step = 5.000

Mar 3 13:30:19 iteration 1 delta = 162721.2463 distance = 17.3733
        MinorPause = 2.0000 step = 2.5000

Mar 3 13:30:19 iteration 2 delta = 0.0000 distance = 17.3733
        MinorPause = 2.0000 step = 1.2500

...

Mar 3 13:30:37 iteration 25 delta = 0.0000 distance = 14.9395
        MinorPause = 2.7885 step = 0.0000

ABS terminated Mar 3 13:30:38 at iteration 26
Total data points = 17640
RMS distance using H1_norm = 14.9395
```

RMS distance using L2_norm = 13.1945

Parameter Estimates:

MajorPause 40.0000 (fixed)
MinorPause 2.78853 (estimated)

2.4. Command Line Syntax

Command line options are used both to provide the names of the various control files and to specify other options. The command line options are:

–a[bs control file specification] ABS_control_file
: Read ABS control information from the file ABS_control_file.

–d[raw tree]
: Write on standard output a description of the tree read from the utterance description file in the form of a LISP S-Expression. This can be used as input to other tree drawing programs.

–h[elp]
: Print a summary of command line options.

–m[model definition] model_definition_file
: Read the model description information from the file model_definition_file.

–o[utput file] output_file
: Write the results of the synthesis in the file output_file.

–p[arameters]
: Print a list of the available model parameters and their documentation.

–r[ules]
: Print a list of the available rules and their documentation.

–u[tterance description] utterance_description_file
: Read the tree description information from the file utterance_description_file.

–v[erbose]
: Print information about what is happening. This is mainly useful during ABS runs.

If an ABS control file is specified on the command line, analysis-by-synthesis is attempted. Otherwise it is assumed that the user wishes to generate a synthetic F0 contour. Thus, for an ABS run a typical command line would be:

```
model -a absctrl -m model.def -v
```

while for a synthesis run a typical command line would be:

```
model -m model.def -u nibun.tree
```

2.5. Program Internals

Although the model can be configured to some extent at run time, major modifications of the model require changes in the underlying C code. This section provides the information necessary to make such changes in the model.

The actual model code is contained in a single file named *model.c* in the source directory. This file is not compiled separately but is automatically included in the file *modelframe.c* at compile time. To change the model one creates a new *model.c* file, moves it to the source directory, and types *make*. Assuming that there are no errors in *model.c*, a new model program will be generated automatically.

The file *model.c* must define two data structures and one subroutine. The two data structures are the parameter list and the rule list. The parameter list looks like this:

```
static struct pardesc pars[]={
"DecSlope",10.0,1.0,FALSE,"Slope of declination line",
"Beta",1.0,1.0,FALSE,"Time constant of accent unit smoothing function"
};
```

The parameter list must be declared to be an array of *pardesc* structures named *pars*. A *pardesc* structure contains the name of the parameter, its value, the ABS step size, a flag indicating whether or not the parameter is to be estimated, and a documentation string:

```
struct pardesc {
        char *name;
        double val;
        double step
        int estimate;
        char *doc;
};
```

The estimation flag should be initialized to *FALSE*.

The rule list looks like this:

```
struct ruledesc rules[]= {
"Declination",FALSE,"Linear declination rule",
"Downstep",FALSE,"Downstep of step function amplitude"
};
```

The rule list must be declared to be an array of *ruledesc* structures named *rules*. A *ruledesc* structure contains the name of the rule, a flag indicating whether or not the is to be applied, and a documentation string:

```
struct ruledesc {
        char *name;
        int apply;
        char *doc;
};
```

The rule application flag should be initialized to *FALSE*.

The file *model.c* must also define the subroutine *model*, which performs the actual calculations. *model* is called with two formal parameters, a pointer to the root of an utterance description tree, and the name of an array in which to write the results, e.g.:

```
model(tree,result)
NODE_PTR tree;
double result[];
{
                    Body of Subroutine
}
```

model should return the number of points for which values were generated.

To refer to the value of a parameter, simply index into the *pars* array and use the *val* member of the structure. In the above example, the value of the parameter *DecSlope* is *pars[0].val* and the value of *Beta* is *pars[1].val*.[2]

To decide whether to apply a rule, simply index into the *rules* array and test the value of the *apply* member. For example, to test whether the *Declination* rule is to apply, write:

```
if(rules[0].apply) {
                    Code for Declination Rule
}
```

The root of the utterance description tree is a pointer to a node structure, defined in the file *nodes.h*. This structure contains the name of the node, its type, how many daughters it has, a list of daughters, a pointer to its parent node, and pointer to an attribute list. The parent node and the daughter nodes are themselves pointers to node structures.

An attribute has two values, one a string, the other a double precision floating point number. The model program automatically converts the string value to a double, if that is possible, so the *model* subroutine may refer to whichever value is appropriate.

It is possible for the user to manipulate a node's attribute list directly, but this should never be necessary. Instead, access is provided by the subroutines *get_attribute* and *get_pnval*.

get_attribute takes as arguments a pointer to a node and the name of an attribute and returns the string that is the value of the given attribute of the specified node, or a null pointer if no such attribute exists.

get_pnval takes as arguments a pointer to a node and the name of an attribute and returns a pointer to the numerical value of the given attribute of the specified node, or a null pointer if no such attribute exists.

[2] It is convenient to define symbolic names for the array entries so that it is not necessary to refer directly to the indices into the parameter list.

3. Interactive F0 Modelling Using a Programmable Time Series Editor

When studying numerical models of fundamental frequency, it is necessary either to visualize the effect of changing parameters by carefully studying the equations, or to execute a numerical model and plot the results. This quickly becomes difficult and tedious, especially if we change not only the parameters of the model but the model itself. In order to allow easy interactive study of a variety of F0 models we have created an interface between the F0 modelling program and the **L3** programmable time series editor.

The use of this interface, which provides an instant display of any particular model configuration and allows easy visual comparison of model configurations and of synthetic F0 contours with data, greatly facilitates the study of a variety of F0 models.

3.1. The L3 Programmable Time Series Editor

L3 is an editor that provides for display, measurement, printing, and modification of any number of time series. Unlike other tools of this type, L3 is completely programmable. In fact, it is best thought of as an interpreter for a Turing-complete recursive programming language with unusual semantics, much as the programmable text-editor EMACS (Stallman 1984) may be viewed as an interpreter for a programming language that contains primitives for editing text. L3 can read commands from files, can write text to files, and can execute other programs as child processes, so it has available the necessary means of communication with the F0 model. L3 is described in detail in (Poser 1987) and (Poser 1988).

3.2. The Interface to the F0 Model

The interface to the F0 model is created by executing the initialization file *runmodel.init*. This in turn executes commands that create the necessary windows and data buffers, define the procedures that form the interface, and create the necessary L3-internal variables. *runmodel.init* loads the standard L3 library files *std_aliases.l3*, *std_procs.l3*, and *mouse_modes.l3*, the files *synf01.l3* and *synf02.l3*, which create and configure the necessary windows and data buffers, and the file *runf0.l3*, which contains the heart of the interface to the F0 model. All of these files are to be found in the L3 library directory, which is normally /usr/local/lib/L3.

L3 variables are used to represent the parameters of the model and to control the application of rules. When the L3 program that implements the interface is loaded, it queries the model program for the currently defined rules and parameters, so changes in the model do not require changes in the interface.[3] The user may ask L3 to show the current values of the parameters

[3] L3 queries the model by executing the model with the *-p* and *-r* command line options and then executing an AWK program to convert the parameter and rule lists provided by the model into L3 commands, which are then read. Consequently, the AWK language must be available (as it is on virtually all UNIX systems).

and may set parameters and enable or disable rules. Execution of the L3 procedure **run** causes L3 to pass the current values of the parameters to the model, to execute the model, and to read in and display the resulting synthetic F0 contour.

The user may arrange for the model to be re-executed and the new synthetic F0 contour displayed automatically whenever a parameter is changed or a rule enabled or disabled. L3 variables have, in addition to a name and a value, a **hook**, that is to say, a pointer to an L3 procedure which is automatically executed whenever the variable is assigned to. Binding the procedure **run** to a variable's hook, will cause the procedure **run** to be executed automatically whenever that variable is changed.

By using two windows instead of one it is possible to compare two different versions of the model or to compare a synthetic F0 contour with a real one. The procedures *L2_norm* and *H1_norm* may be executed in order to compute the distance between the time series in the two windows.

The procedures defined by the interface library are:

bind_all_hooks

Binds the procedure run to the hooks of all of the currently defined parameters, rule application flags, and the tree definition file.

disable rule

Disable application of the named rule.

enable rule

Enable application of the named rule.

H1_norm

Print the H1_norm distance between the time series in the two data buffers, leaving the value in the variable *rval*.

L2_norm

Print the Euclidean distance between the time series in the two data buffers, leaving the value in rval.

run

Execute the model with the current parameters, load the selected data buffer with the result, and display it.

show_parameters

Print the current model parameters in the Messages window.

toggle_windows

Switches back and forth between two different windows for displaying the synthetic F0 contours, if two windows have been set up.

unbind_all_hooks

Unbinds the procedure hooks of all of the currently defined parameters, rule application flags, and the tree definition file.

Since L3 provides a true programming language, the user may impose constraints on the values of model parameters by writing a procedure that computes the value of one parameter from that of another and binding this procedure to the hook of the independent variable. Thereafter, whenever the independent variable is assigned a new value, the procedure will automatically be executed and will compute a new value for the dependent parameter.

As a simple example, suppose that the model contains parameters named A and B and that we wish to constrain B to be 2.0 times A. First, we write a procedure that computes B given A. Then we bind this procedure to A's hook, so that whenever we change the value of A the procedure will be executed and change the value of B accordingly.

```
procedure SetB
      * $A 2.0
      set B $rval
end_procedure

bind_procedure_to_hook A SetB
```

Acknowledgments

I wish to thank Dr. Akira Kurematsu and Dr. Kiyohiro Shikano for their support of this research, Mr. Masanori Miyatake for his assistance with the analysis-by-synthesis algorithm, and Mr. Takaharu Tanaka for his assistance in porting L3 to the Microvax.

References

Miyatake, Masanori & Yoshinori Sagisaka (1988) "Prosodic Characteristics and Their Control in Japanese Speech with Various Speaking Styles," ATR Interpreting Telephony Research Laboratories Technical Report I-0025.

Poser, William J. (1987) *L3 Reference Manual.* Stanford, California: Center for the Study of Language and Information, Stanford University. Technical Report 87-94.

Poser, William J. (1988) "L3 — A Programmable Multiple Time Series Editor for Speech Research," ms. Stanford University, Stanford, California, USA.

Selkirk, Elizabeth (1984) *Phonology and Syntax: The Relation Between Sound and Meaning.* Cambridge, Massachusetts: The MIT Press.

Stallman, R. M. (1984). "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," *Interactive Programming Environments.* edited by D. R. Barstow, H. E. Shrobe, & E. Sandwell, (New York: McGraw-Hill), pp.300-325.