TR-IT-0133

Signal processing for concatenative synthesis

Christian LELONG

1995.9

ABSTRACT
For efficient control of pitch and duration in a synthesis system employinbg waveform concatenation, direct modificatino of pitch-period-sized samples of the speech signal is performed. This report describes the six-months work spent under an INT internship developing and extending the PSOLA module of the ITL dept 2 ChATR synthesiser. A version of the report was presented as part of the graduation requirement for Stage Ingenieur.

ATR

STAGE INGENIEUR

OPTION TRAITEMENT & APPLICATIONS DE L'IMAGE

INSTITUT NATIONAL DES TELECOMMUNICATIONS

PROSODY MODIFICATION IN SPEECH SYNTHESIS

Interpreting Telecommunications
Research Laboratories, ATR, Japon

Fevrier - Juillet 1995

Christian LELONG

RAPPORT CONFIDENTIEL

DIRECTEUR DE STAGE : NICK CAMPBELL

CONSEILLER D'ETUDES : BERNARDETTE DORIZZI

RESUME

Le cadre général de ce travail est celui de la synthèse de la parole par ordinateur. Dans ce contexte, la prosodie occupe une place importante : un bon ajustement de la durée, du ton et de la puissance de la voix de synthèse contribueront en grande partie à rendre celle-ci plus "naturelle". La démarche habituelle est la suivante. Dans un premier temps, le texte d'entrée est analyse, afin d'obtenir une transcription phonétique. Une deuxième étape consiste à créer une structure rythmique en accord avec la syntaxe. Le dernier maillon génére le signal de synthèse à partir de toutes ces informations. Le but de ce projet se situe dans cette dernière étape et consiste, d'une part, à mettre en place puis évaluer divers algorithmes de modification des paramètres prosodiques, en particulier la méthode PSOLA (Pitch Synchronous Overlap and Add), d'autre part à détecter et pallier par des techniques du traitement du signal, dans la mesure du possible, aux éventuelles imperfections responsables du timbre artificiel de la voix de synthèse. Le tout devait ensuite être integré au sein du système de synthèse CHATR développé a ATR.

ABSTRACT

The general framework of this report is speech synthesis. Prosody plays an important role in this context : an efficient control of duration, pitch and power in the synthesized voice will contribute greatly to make it sound more "natural". The usual approach is as follows. In the first stage, input text is analysed in order to obtain a phonetic transcription. The second step consists of deducting a certain rythmic structure stemming from the syntax. In the final stage, we make use of all the previous information and create the synthesized signal. This project has to do with this last step, and is intended to implement and evaluate different algorithms for prosodic modifications, in particular the PSOLA method (Pitch Synchronous Overlap and Add), as well as to detect and palliate when possible, using signal processing techniques, all the eventual clicks responsible for the artificial sound in the synthesized voice. The whole module had then to be integrated into CHATR, a speech synthesis system developped at ATR.

Contents

General introduction

Machines seeming to understand and/or produce speech have always caused amazement. In Ancient Greece, smart acoustical devices, chanelling the voice of a hidden speaker, allowed priests to surprise the faithful, as they made oracles and statues address the crowd. In the XVIII century, the first attempts at artificially reproducing human speech consisted in small, mechanical machines employing pumps and valves, and capable of producing more or less natural sounding phonemes, and even streams of words [1]. However, it was not until the 1940s that speech processing has made significant progress and was viewed with interest both by industry and the scientific community.

This vast field, where many other disciplines converge (Linguistics, Phonetics, Signal Processing, Computer Science, etc), is divided into two main areas :

- speech recognition, consisting of transforming an acoustic signal into a list of graphemic symbols.

- speech synthesis, where speech waveforms result from a symbolic representation of information, and constitutes the framework of this specific project.

As in many other fields closely related to the human brain and body (vision, artificial intelligence...), speech is the output of a long, complex procedure, that can't be easily accessed for close study. Hidden Markov models, neural nets and sets of rules have not so far accounted for all the phenomena involved. Practical applications have been scarce, current know-how being sometimes unable to elaborate sufficiently reliable models. Potential applications, however, are numerous, and often seem to come out of a science fiction book : real-time translation between languages, better interfaces, aids for the handicapped and for children, voice recognition as a security measure, etc.

In speech synthesis, the prime consideration is the quality of the synthesized speech. A metallic-sounding voice is often difficult to understand, and always tiring after a few minutes. Also, unnatural intonation and duration patterns might modify the meaning,

result in a weird-sounding sentence, and puzzle the listener. Other criteria such as speed, hardware requirements (mainly in processing power and memory available) and type of speaker are also important. Needless to say, producing synthesized speech undistinguishable from real speech has turned out to be an elusive and difficult task, hence the different possible approaches. Part of the problem lies in the fact that prosody, ie power, duration and intonation, depend on both the speaker and the meaning of the spoken utterance, and its prediction is a non-trivial task. The vocal tract acts as a complex filter whose transfer function changes from speaker to speaker, and from phoneme to phoneme : efforts at modelling it accurately have so far given mixed results.

One possible approach in speech synthesis is concatenative synthesis. Here, the artificial speech is built by putting together tokens of real, recorded speech selected from a large database. These tokens (units), however, cannot be used as such. They must undergo a series of signal processing modifications, ensuring that their prosody matches the predicted value, and contributing as a whole to render the output voice as natural as possible.

Such is the main goal of this project : to implement and evaluate different methods of performing prosodic modifications, unit concatenation, and overall improvement of the synthesized speech quality.

Special attention will be paid to PSOLA (Pitch Synchronous Overlap and Add), a particularly promising method [3], but other techniques will be discussed as well.

A general overview of the project is presented in Chapter 1. A brief presentation of ATR, and the laboratory where this work took place is also included. Readers who are not familiar with this field will find in Chapter 2 an introduction to speech processing, with a particular emphasis on concatenative synthesis. The speech synthesis system constantly used, *CHATR*, is presented in Chapter 3. Being a large, complex system, we will focus on its global architecture, and then on the modules most relevant to our work. We end by presenting in detail the PSOLA method.

Readers already familiar with this context will start at Chapter 4, where the different implemented algorithms, regrouped in a unit labeled *XPSOLA*, are discussed in detail. Where needed, algorithms, graphs and plots will be reproduced, and a brief assesment will follow each individual method. Therefore, this chapter might be seen as a reference guide.

The following chapter is rather like a user's guide. It shows how the new unit was implemented in *CHATR*, how the different options can be used, and which tests were performed. And finally, Chapter 6 gives a global evaluation of the project, and points to possible directions for future work.

Chapter 1

Overview

1.1 ATR research laboratories

Advanced Telecommunications Research Institute International, which was established in 1986, in Osaka, with support from various sections of industry, academia and government, is intended to serve as a major center of telecommunications R&D. In 1989, it moved to its current location in Kansai Science City. Currently, around 300 researchers, 20% of them foreigners, work in ATR laboratories.

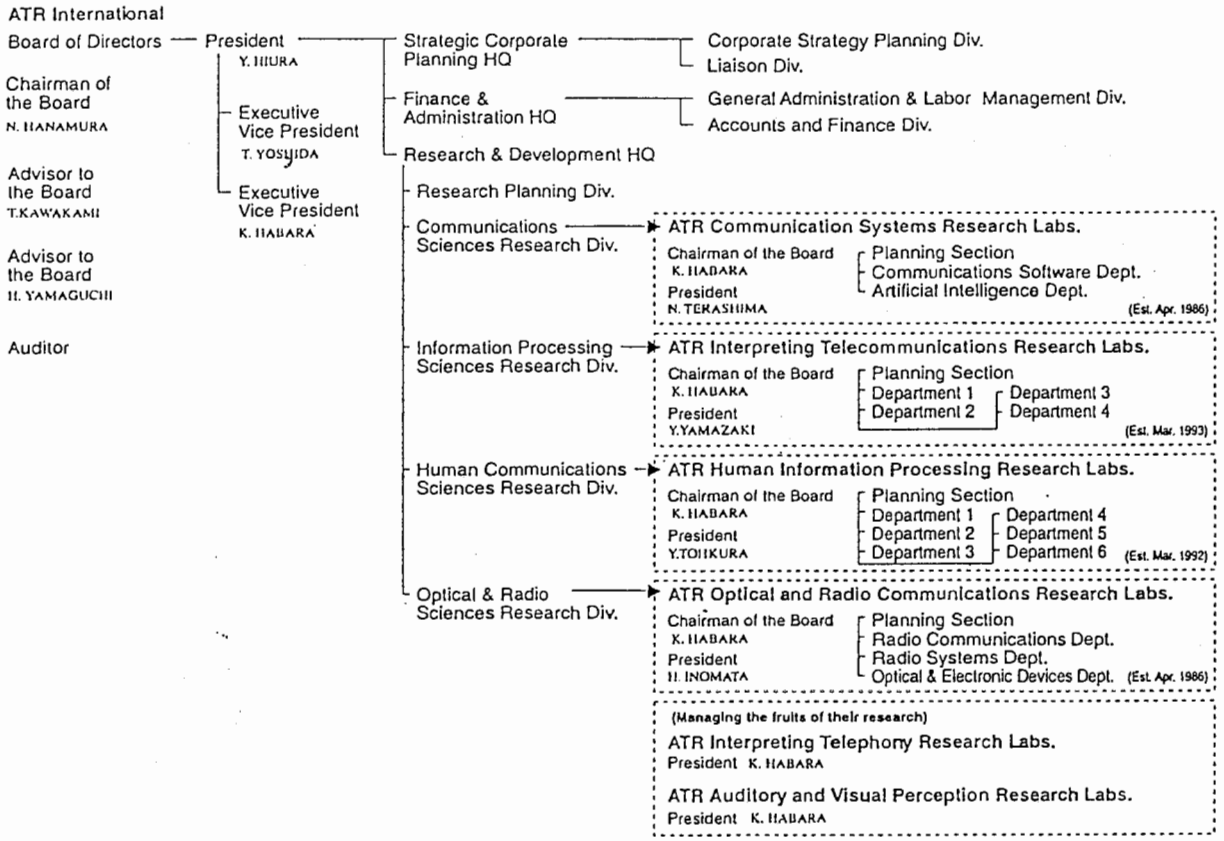
ATR is organized into five separate companies, each specializing in a particular field of telecommunications research. The five companies share the same buidings, but each one has its own computer network, laboratories and management. Given their different fields of research, cooperation between them is limited.

- **Communication Systems** research laboratories focus on human-oriented intelligent communication systems. The acquisition, recognition, comprehension and display of 3-D images in order to create a virtual environment are one of the main axis of research. Other areas include system security and automatic generation of communications software.

- **Human Information Processing** research laboratories are developing human/machine communication technologies. Stress is put on the way the human brain receives and processes information, particularly in the areas of speech recognition and visual information such as recognition of facial expressions, and motor signals. A wide range of disciplines is involved, from neural nets to psychology.

- **Interpreting Telecommunications** research laboratories aim at cross-language global communications. The three areas of research are speech

ORGANIZATION



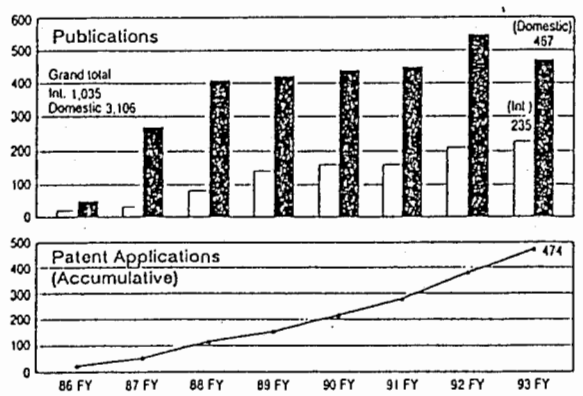
CAPITAL

ATR International : ¥22.03 billion
(Invested by 141 Companies)

RESEARCH BUDGET

Total of 4 Research Div. : About ¥9 billion /Year
(Including all expenses)

RESEARCH RESULTS



EMPLOYEES (Including Executives)

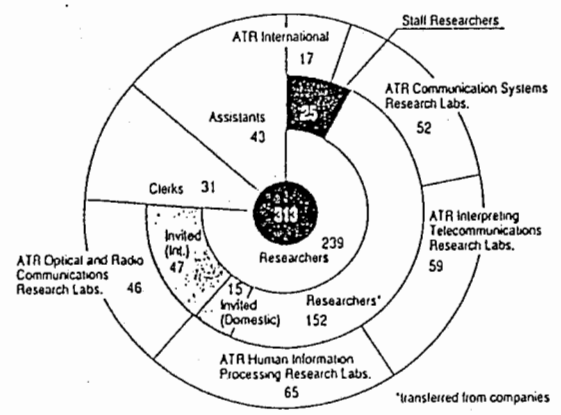


Fig 1. *outline of ATR*

recognition, language translation and speech synthesis (currently for Japanese, Korean, English), each discipline interacting with the other two. The ultimate goal is a system playing the role of a human translator : listening, translating, speaking.

• **Optical and Radio Communications** research laboratories are concerned with the new technologies involved in future telecommunications networks. The main disciplines involved are optics, electromagnetism and quantum physics. Research is particularly aimed at mobile and optical intersatellite communications, array antennas, and the optical and electronic devices involved.

• **Media Integration & Communications** research laboratories were created recently, and will eventually replace the CSR laboratory, their goals being somehow similar.

Strong support from the Japanese government and industry, plus an active program of exchange between universities and research institutions in Japan and abroad, have made of ATR an important player in the telecommunications research field. An example of this is the list of events that have been held over the past months :

- 1995 International Workshop on Computational Modeling of Prosody for Spontaneous Speech Processing, April 12-14 1995.

- the Science & Technology seminars, spreading over several weeks in April and May 1995, featuring invited lecturers from : University of Toronto, Universite de Geneve, Institut National Polytechnique de Grenoble, Tokyo University, MIT, Beckman Institute, etc.

1.2 The PEGASUS group

This work group was established in Department 2 of Interpreting Telecommunications laboratories, ITL. Amongst the 4 departments of ITL, it is mostly concerned with speech synthesis, and houses a team of 15 researchers approximately.

PEGASUS stands for **P**rosody **E**xtraction and **G**eneration **A**nd **S**ignal processing and **U**nit **S**election, representing the two overlapping research areas of Department 2 (*cf Figure 2*). Each sub-group holds informal meetings every two weeks, in which every member is expected to present the state of his or her personal research, and to make constructive comments on colleagues' work. I was assigned to the SUS sub-group. As most members were Japanese, the meetings were usually held in Japanese, with occasional shifts to English. The typical duration was one hour and a half. Each meeting was presided by Norio Higuchi, the Department Head.

The specific tasks currently under way are :

- *optimisation of unit selection from speech databases* (Alan Black & Nick Campbell) : a database with greater variability allows closer modelling of naturalness and differences in speaking styles.

- *sub-phonemic optimal path search* (Yoshiharu Itoh, Makoto Hashimoto & Norio Higuchi) : paying attention to the spectral distortion at concatenation points increases the quality of synthesized speech.

- *automatic detection of phrase boundaries* (Toshio Hirai, Norio Higuchi & Yoshinori Sagisaka) : using information concerning the fundamental frequency, the goal is to find the limits (i.e., subjet, verb, etc) in a spoken utterance.

- *spectral mapping for voice conversion* (Makoto Hashimoto & Norio Higuchi) : using speaker selection and vector field smoothing, the aim is to reduce the spectral distance between the transformed speech and the target speaker.

- *cepstral analysis / synthesis* (Lee Yang-Hee & Toshinori Satoh) : improving the algorithms involved in cepstrum estimation and manipulation.

- *evaluating objective cost function for unit selection* (Andrew Hunt) : using MEL-cepstrum, power information, etc, finding the best points for unit concatenation in terms of sound quality.

- *building a Korean database* (Lee Yang-Hee) : labelling of phonemic units, creating pitch mark sets, documenting durations, voicing, etc.

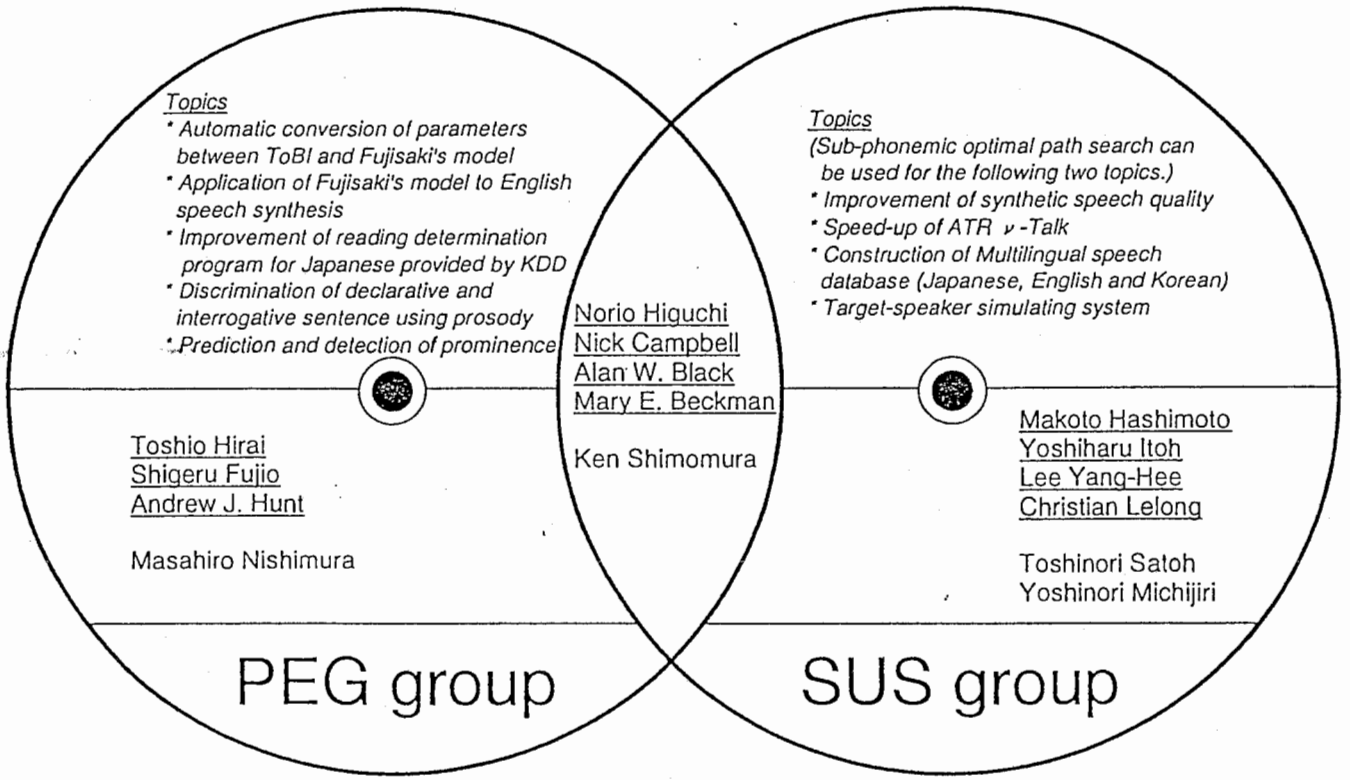


Figure 2 . the PEGASUS group

I found the working environment rather relaxed. Working hours are flexible : the office is open 24 hours a day, seven days a week, and everyone is free to choose his or her schedule. The only constraints are the 11 a.m. - 2 p.m. period, which must be spent at the

ofice (break for lunch included), and the total number of working hours per month (a minimum average of 7h40 per day is required). Also, researchers have some control over the course and direction of their work. Rather than imposing strict deadlines for completion of a given project, the supervisors are open to comments, and follow closely the evolution of each particular task. Finally, exchanges between researchers are encouraged, with good reason. All this, coupled with excellent facilities and a number of leisure activities organized by the laboratory, make for a pleasant environment.

1.3 CHATR, a speech synthesis system

ITL's Department 2 has developed a generic speech synthesis system, called *CHATR* [6]. Its main authors are Nick Campbell, Alan W. Black and Paul Taylor, but it has received contributions from many different people, adding and updating modules around the core system. Its purpose is to serve as a research tool in speech synthesis, and not as a product that might be commercialized in the future. Therefore, *CHATR* has been designed as a way of testing, improving or creating new algorithms and models. It is also part of the global project under way at ITL, the automatic translator.

The main characteristics of the system are :

- multilingual : So far, Japanese and English are supported, and in the coming months Korean will be added.

- multi-speaker : within a given language, it is possible to select amongst a set of different speakers, each one recorded into a database, the voice used for synthesis.

- the main parameters and algorithms can be set and selected at run-time.

- unlike most Text-To-Speech systems, it accepts different kinds of input, from plain text to other, more complex kinds of representation.

A more detailed description of *CHATR* can be found in chapter 3.

1.4 Project outline

My internship, part of an informal accord between ATR and the INT, took place from February 1 to July 31, 1995, under the supervision of Nick Campbell, Supervisor at Department 2, ITL.

A previous researcher had contributed a signal processing module based on the PSOLA algorithm. As its performance was not deemed satisfactory, I was requested to implement a new version of the same algorithm, using the previous module, called RUC,(*cf § 3.3*), as a source of inspiration, while avoiding any of its mistakes. Also, I was given total freedom concerning any personal, suplementary algorithms I might develop. As I would constantly be in contact with the persons concerned, namely Dr. Campbell, Dr. Black, and any other potential user, my task was expected to be quite interactive.

Thus the first stage of my internship consisted of getting familiar with the field, the environment, and the tools provided. I spent around four weeks reading books, papers and

thesis related to speech processing, learning the fundamentals of CHATR, and becoming familiar with the software I was going to use : **xwaves+** and **MATLAB**.

Soon after I began writing the new module, I decided to ignore the previous work and start from scratch. I found much to criticize on the programs I was handed, and the additional length of time involved in starting anew paid off in terms of performance, robustness and ease of use. This made me realize the importance of writing clear, well-commented code.

Being an outsider to the field of speech processing proved to be both a handicap and an advantage. On one hand I did not posess most of the basic knowledge needed for this kind of task, so I committed many mistakes, particularly in the first three months, that I could have avoided otherwise ; that resulted in a loss of time, but also in valuable experience acquired. But on the other hand, this helped me come up with new ideas, some of which were wrong and without solid theoric basis, but others proved to be workable and valid.

The second stage, implementing a working version of the RUC module took a full three months, from March to May. At this point, only the basic functions were operational. The month of June was spent incorporating the rest of the options, as well as new suggestions, and running tests on all the databases in order to proove the robustness of the module. Finally, during the last four weeks, major changes were made to the system in order to accommodate a new type of pitch marks, obtained by a more reliable method. This last stage somehow delayed the final tests and the definitive implementation into version 0.7 of *CHATR*. This last stage was conducted in collaboration with Alan Black.

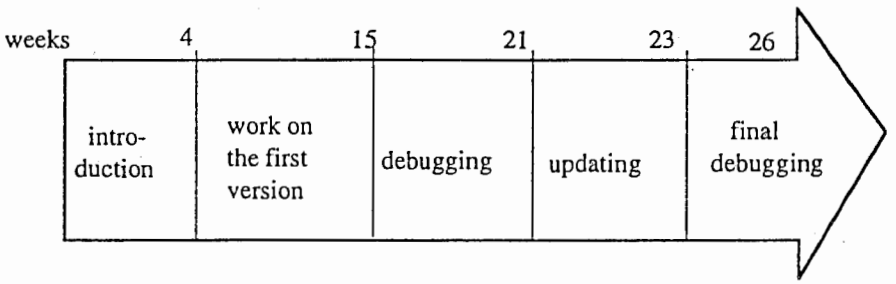


Fig 3. *project outline*

During my internship at ATR, I was also required to make two informal presentations, open to members of Departments 1 and 2 of ITL. The first one took place a few weeks before finishing the first running version. I presented the algorithms I was implementing, but without examples to support them. Most people present were members of PEGASUS, and people likely to use the *CHATR* system. The final talk, in mid-July, was an occasion for showing the definitive version, backed with tapes of synthesized speech thus produced. Each presentation lasted for an hour, followed by a brief period for feedback and discussion. Also, prior to leaving, I was requested to deliver a "Technical Report" concerning my work at ATR, addressed to future users and programmers of the system.

Chapter 2

Introduction to speech processing

2.1 Production

Sound production by the vocal tract is the result of a complex mechanism, not yet fully understood. It is possible, however, to slightly simplify the way it works. In doing so, we will introduce a number of terms that will be useful for the rest of this report.

Speech is the result of voluntary motions of the vocal apparatus. It is a behavior which must be learned, and which is constantly controlled by the acoustic feedback of the hearing mechanism and the kinesthetic feedback of the musculature involved. For example, partially or totally deaf persons experience difficulties in producing adequate speech ; the same happens when a patient has parts of his mouth desensitized by his dentist. The vocal apparatus involved in speech production is shown in Fig. 4. It is composed of three main parts, each playing a different role : the respiratory apparatus, the larynx and the vocal tract.

The respiratory apparatus produces the air flow needed for producing most sounds found in speech. Its behavior presents two stages : inhalation and expiration. Inhalation is accomplished by enlarging the chest cavity, making use of the thoracic and abdominal musculature. As a result, air pressure in the lungs goes down, creating an inflow of air, entering by the nose or mouth, down the pharynx and the trachea. On the other side, expiration consists of contracting the chest cavity and expelling the air from the lungs, by the mouth and/or the nose. This air flow can be used for phonation. Therefore, the respiratory apparatus acts as the energy source for most sounds. However, certain sounds can be produced without producing any airflow (clicks, etc).

The first perturbation of the air flow coming from the lungs takes place in the larynx, situated at the end of the trachea (*cf Fig. 4*). It features nine cartilages, but only

four of them play a role in speech production (*cf Fig.5*). The vocal cords are attached to the arytenoid cartilages, which control the configuration and tension. When the vocal cords vibrate, the sound produced is called *voiced*, and *unvoiced* otherwise. The triangular opening at the base of the vocal cords is called the glottis, and it regulates the air flow like a valve (*cf Fig 6*).

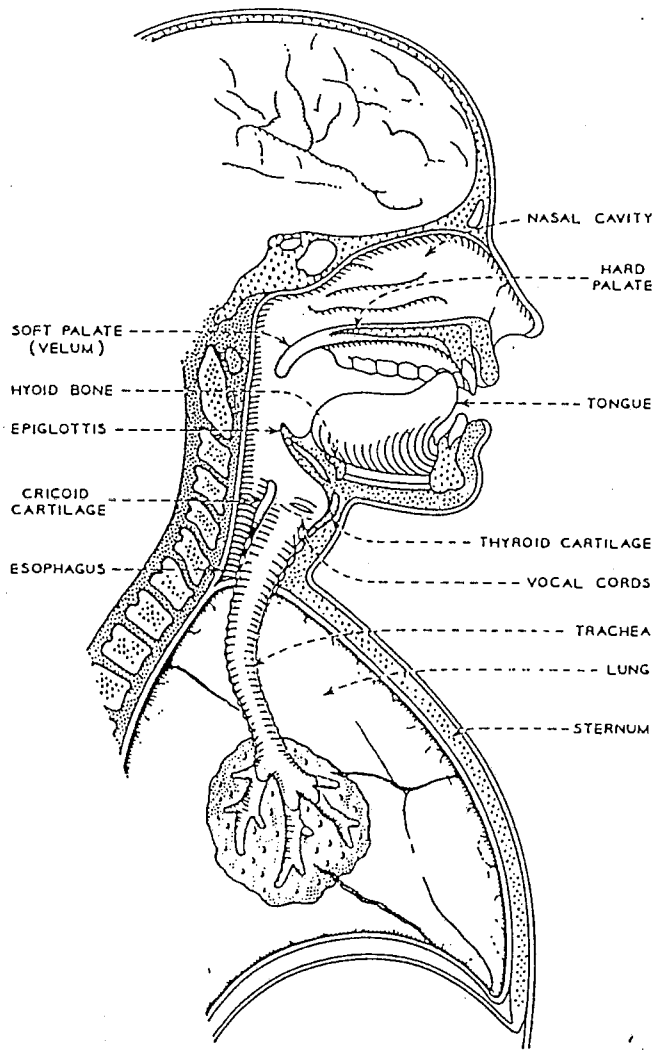


Fig 4. *diagram of the vocal apparatus*

Finally, the vocal tract consits of a set of articulators modifying the air flow : the velum, tongue, lips, pharynx and jaw, mainly. They can either partially or totally interrupt the air flow at a given point of the vocal tract, creating the kind of turbulence found in fricatives, for example. They can also act as a filter moduling the semi-periodic signal emitted by the larynx during voiced speech.

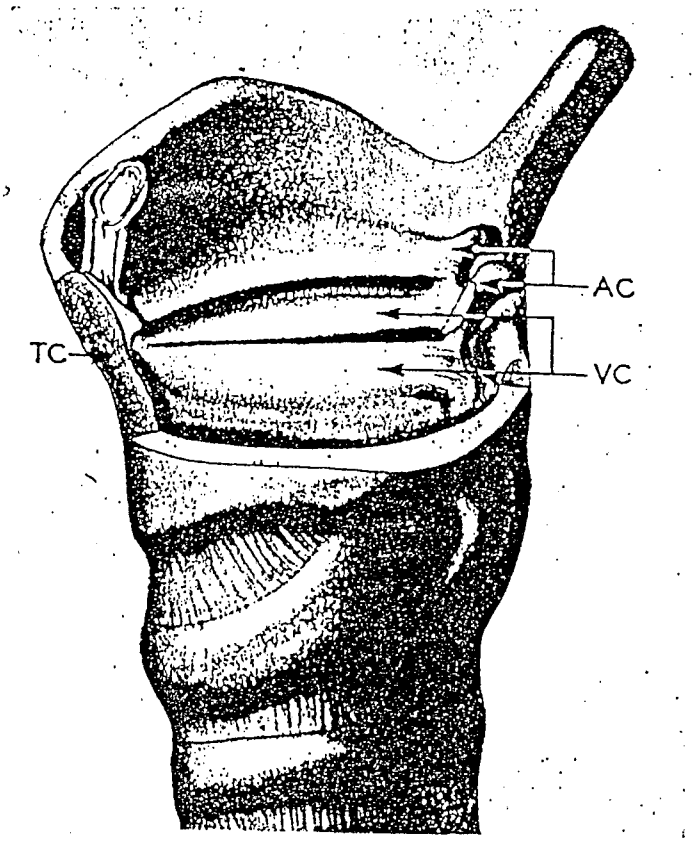


Fig 5. *cut-away view of the human larynx*
VC - vocal cords ; AC - arytenoid cartilages ;
TC - thyroid cartilages

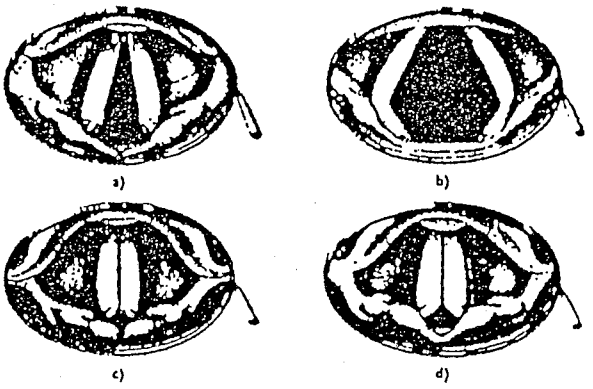


Fig 6. *the glottis in different positions of a) breathing;*
b) deep inhalation; c) phonation; d) whispered voice

2.2 The speakers

2.2.1 the kinds of speech

Speech varies between languages, dialects and individuals. Every "dialect", e.g.. American English, has its own set of phonemes (*cf Table 1*), and a certain rhythmical structure. Therefore, the study of prosody depends on the language studied. And once a language has been selected, each speaker has a unique voice and intonation pattern.

phoneme	example	phoneme	example	phoneme	example
i	b*ea*d	a^{w}	b*ou*t	t	*t*ea
ɪ	b*i*d	a^{j}	b*i*de	k	*k*ey
ɛ	b*e*d	ɔ^{j}	b*oy*d	v	*v*eal
æ	b*a*d	r	*r*ide	ð	*th*en
ɒ	b*o*dy	l	*l*ight	z	*z*eal
ɑ	f*a*ther	w	*w*ide	ʒ	gara*g*e
ʌ	b*u*d	j	*y*acht	f	*f*eel
ɔ	b*au*d	m	*m*ight	θ	*th*in
o	b*oa*t	n	*n*ight	s	*s*eal
ʊ	b*oo*k	ŋ	so*ng*	ʃ	*sh*ore
u	b*oo*t	b	*b*ite	h	*h*ead
ə	*a*bout	d	*d*og	ǰ	*j*eep
ɚ	b*ir*d	g	*g*et	č	*ch*ore
e^{j}	b*ai*t	p	*p*et		

Table 1. *American English phonemes*

However, every individual speaker can present considerable variability in the speech he produces, depending on a number of factors : whether he is angry, tired or excited, whether he is hesitating or not, etc. As far as this project is concerned, we shall consider only the two following categories of speech :

Read speech is produced by a speaker reading a given text. Unless the speaker has been specially trained as a story-teller, anchorman or actor, the prosody of read speech has little variability, and the intonation is rather monotonous. Databases of read speech can be obtained from sets of words (minimum intonation variability) or sentences (more variability).

Spontaneous speech is produced during normal conversations. As opposed to read speech, the prosody can vary greatly : hesitations, bursts of fast speech, various emotions contribute to a wide range of examples for every class of phoneme. Databases are generally obtained from interviews, where the subject is asked a set of questions (i.e. *"tell me about your stay in Japan"*).

2.2.2 the tools employed

Two software packages were principally used to develop the new signal processing module, *XPSOLA*. Before any serious programming could begin, a good command of these was needed. MATLAB was only used briefly, while xwaves+ was of great help at all times.

MATLAB is a programming environment consisting of a core unit for general, math-oriented use, and a number of modules, called "toolboxes". Each toolbox covers a specific field, such as neural nets, image processing, etc. The motivation behind this is twofold. On one side, the programmer has at his disposal a very large number of library functions. On the other, the whole system is designed to be very fast at run time. As a result, an algorithm that takes 15 seconds of CPU time when coded in C language might be executed in less than one second when MATLAB is used. The main data type is the matrix, and getting acquainted with the new syntax takes only a few hours. I used MATLAB with the *Signal Processing Toolbox* [*10*] in order to obtain a bank of lowpass filters (*cf. § 4.3.3*).

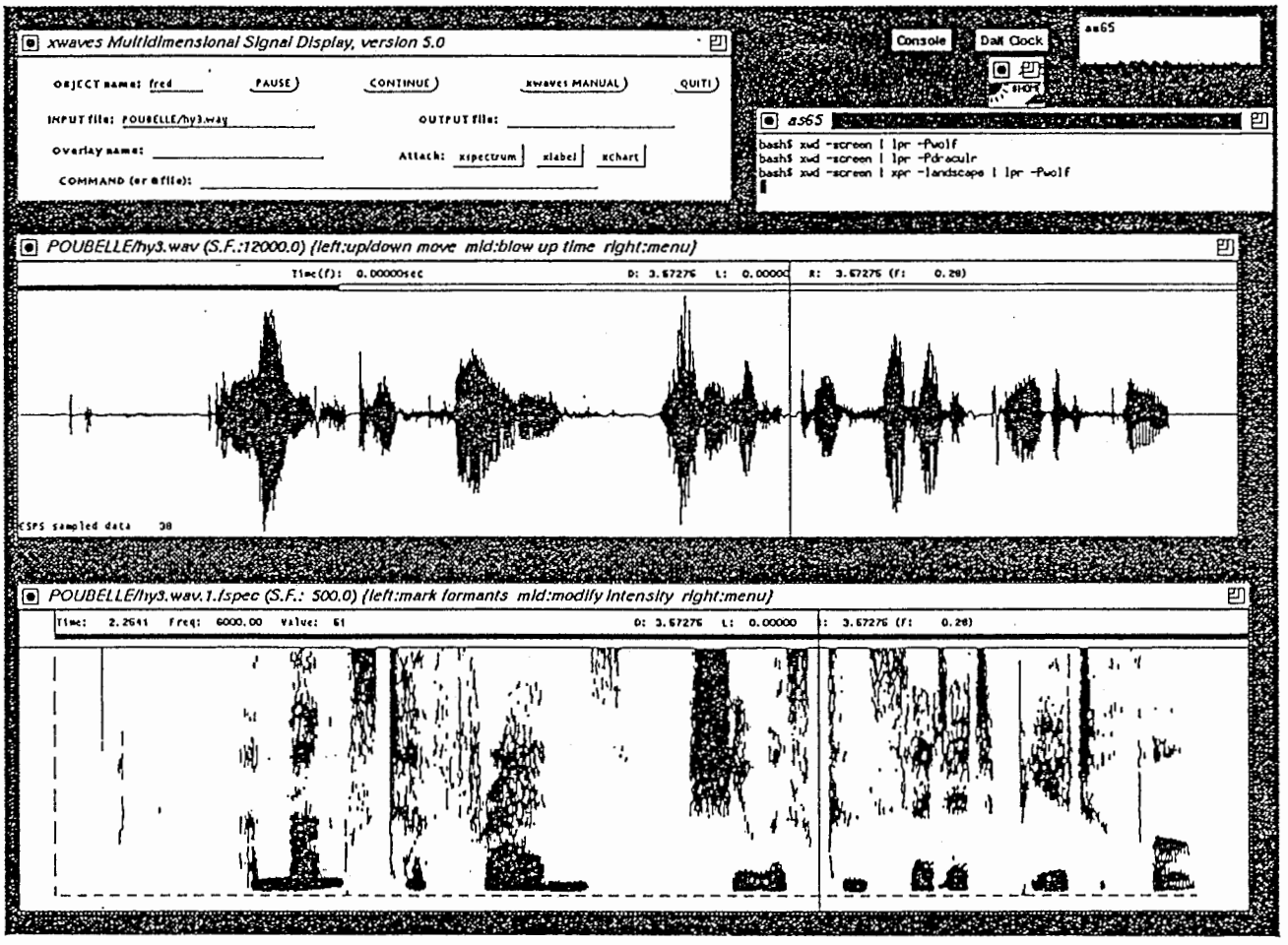


Fig 7. *speech waveform and spectrogram display with xwaves+*

xwaves+ is a large package designed for speech processing. More precisely, it allows to display, analyse, edit and play on loudspeakers the speech audio files (*cf. Fig 7*). It was very important to actually see and listen to the natural and synthesized speech

waveforms in order to get a certain intuition or feeling about the kind of problems encountered : discontinuities, distortion, jumps in power and pitch, all are to be avoided if posible, but some matter more to the quality of the output than others.

Therefore, **xwaves+** served at first as a tutorial, in order to learn about speech signals, and the as a tool for verifying the output of my module, the quality and flaws of the synthesized speech.

2.2.3 the pitch marks

It is important to provide the speech signal with a measure of time and periodicity. This measure is provided by pitch marks.

For every opening/closing cycle of the cords, the corresponding pitch mark matches the highest peak in the waveform during the corresponding time interval, usually from 5 to 15 milliseconds. Thus, the distance between marks becomes an indicative of the current fundamental frequency. During unvoiced speech, however, the vocal cords do not vibrate, and two options are possible. Either "fake" pitch marks are created for unvoiced speech at a fixed rate (10 milliseconds), or pitch marks are restricted to voiced speech.

Pitchmarks are particularly important to the PSOLA method (*cf. § 3.4*), and their accuracy will affect the quality of the synthesized speech produced. They can be automatically obtained using signal processing techniques, but correction by an experimented human labeller is highly desirable. Since we deal with digital data, precision is limited by the sampling rate used, usually 12 or 16 kHz, which is more than enough. Another, more direct method is to detect the vibrations of the vocal cords using a larynxometer. The latter method seems to be more reliable, but again human labelling, although very time-consuming, is the ideal solution.

So far, the two types of pitch marks sets available at ITL are, 1) marks by signal processing, extended to unvoiced speech, and 2) marks by vocal cords tracking, restricted to voiced speech. No handtuning was performed.

2.2.4 the databases

A number of speech databases are available for use with *CHATR*. Each database concerns a particular speaker, and comprises a set of features. The raw data is the speech waveform, obtained from a selected speaker reading a text or list of words in a special noise-proof room, using high quality audio equipment, and sampling at 12 or 16 kHz. Afterwards, the raw data is processed in order to produce additional information needed by the system; The final database should, for concatenative synthesis, contain the following :

- **speech waveform**, the raw digital data
- **pitch marks**, the measure of periodicity
- **labels**, as every word is divided into a set of phonemes
- **vector quantization coefficients**, coordonates in the acoustic space for every individual phoneme, comprising pitch, power, cepstral coefficients, etc

As might be expected, building a speech database is a long and difficult task. Also, the quality of the database will be reflected in the final synthesized speech. A noisy recording, for example, will distort all the processing, inaccurate labels will cause abrupt transitions, and so on. A list of the diferent databases available for *CHATR* is given in Appendix 3, along with some additional details.

2.3 Speech synthesis

2.3.1 articulatory synthesis

As stated before, speech is the outcome of a complex mechanism. Therefore, a good knowledge of such mechanism should allow the recreation of human speech. Such is the principle of articulatory speech : to model faithfully the mechanical motions of the articulators, and the resulting distributions of volume velocity and pressure in the lungs, larynx, and vocal and nasal tracts. This modelling turns out to be very difficult, for various reasons. While the dimensions of the human vocal apparatus can easily be measured, tracking their movement during speech production is another matter. Also, phenomena of a random nature, like the turbulence encountered in the production of most consonants, are very complex. As a result, articulatory synthesis requires several orders of magnitude more computation than other types of synthesis, and has so far only produced less than satisfactory results.

2.3.2 formant synthesis

Formant synthesis takes a humbler approach. The goal is to approximate the speech waveform using a set of rules formulated in the acoustic domain, simpler than that of articulatory synthesis. The principle is shown in Fig 8. A first step consists in approximating the different sources of sound energy separately ; the resulting output is characterized in the frequency domain by *S(f)*. Estimating the transfer function of the vocal tract, *T(f)*, is the second step. Finally, we take into account the directional sound propagation from the head, represented by *R(f)*.

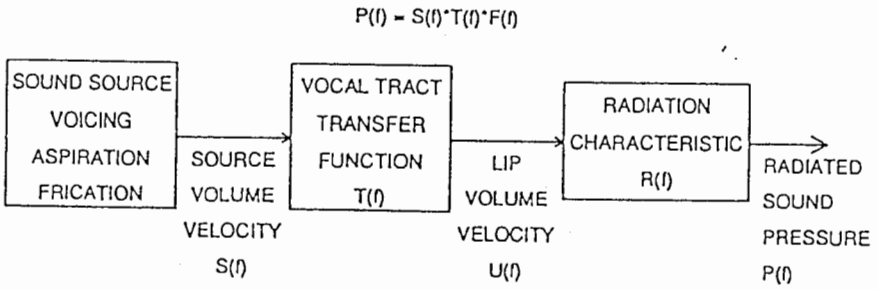


Fig 8. *formant synthesis*

Such a system requires several dozen parameters at every stage that must be tuned accurately. As opposed to concatenative synthesis, it does not require a large database of real speech. But, since it builds its output from scratch, it is easy for the listener to detect

the machine behind. Formant synthesis is possible with *CHATR*, but this will not be discussed here it, as it lies outside the subject of this report.

2.3.3 concatenative synthesis

This is the main type of synthesis performed by *CHATR*, and the one that concerns us most. The idea is to select, extract and reassemble bits of real speech in order to make new words and sentences. It is done as follows :

1. select database, enter input text

2. translate text into a list of phonemes

3. predict pitch, duration and power for every phoneme

4. search the database for the best approximations of the target prosody

5. modify the selected units, in order to match the target prosody

6. concatenate the modified units

Many variants exist, depending on the type of database and unit (diphones, non-uniform units). The raw data might be the MEL-cepstrum coefficients (*cf § 3.2*) of the original speech, and the database might be labeled as phonemes or diphones. Each class has its advantages and drawbacks. In any case, signal processing corresponds to steps 5 & 6. In the following chapter, we analyse more closely an example of a concatenative speech synthesis system, available in *CHATR*, before addressing the core of this project.

Chapter 3

The CHATR system

3.1 Presentation

This is only a very condensed presentation of *CHATR*, a generic speech synthesis system developed at Dept 2 of ITL. For a much more detailed presentation, the reader should consult the *CHATR version 0.7* manual, written by Dr. Black.

CHATR was designed in a way that easily allows a large number of different modules to interact in a defined way. Multiple modules performing equivalent tasks, such as duration prediction and waveform synthesis, are included within the same environment, allowing close direct comparison. *CHATR's* main use in ATR will be within a speech translation system —speech recognition in language *A*, automatic translation, speech production in language *B*— currently under development. The basic system is written in C, while input and output are through a very simple Lisp system.

Many researchers at Dept 2 of ITL have participated in the constant elaboration and upgrading of *CHATR*, mainly Drs Black, Campbell and Taylor. The purpose of this project was to contribute a new module to the system, thus enlarging the choice of algorithms, and hopefully to improve its performance.

3.2 Unit selection

The speech databases vary in length (*cf Appendix 3*), but each one has a great number of examples for every class of phoneme, from a few dozen for seldom used phonemes, to over a thousand for certain vowels. Therefore, selecting the right example requires some

thought, all the more since the quality of the output voice depends greatly on it : badly selected units make for bad synthesis, good units sound almost like real human speech.

Actually, two selecting functions are implemented, but that number should increase. The simpler one starts at the beginning of the utterance, and looks for the longest matching unit stream. This is repeated until the end of the utterance. In doing so, we obtain a low number of breaks (but not the optimal low), where clicks and jumps in the output speech can occur. The second function, however, is less naive, and more time consuming. It involves a cost function C(s), whose different weights can be set by the user at run-time, and which is shown below.

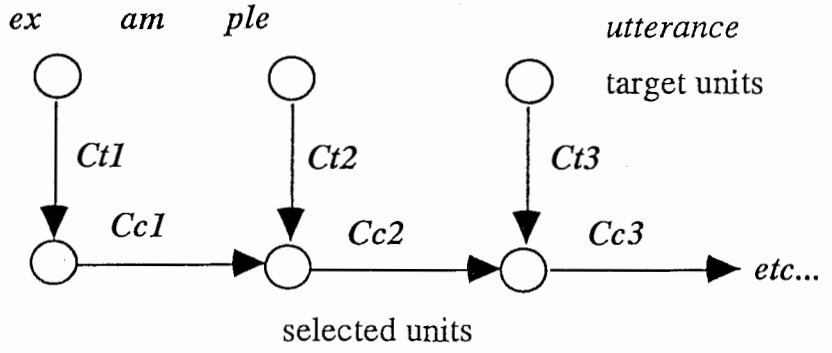


Fig 9. *cost function for unit selection*

The total cost function is the sum of two terms, the continuity cost and the target cost, *Cc(u)* and *Ct(u)* respectively. The first term measures the smoothness of the resulting utterance, and includes a mel-cepstrum mesure of the consecutive units. A low value for *Cc(u)* means that clicks and jumps in the synthesized utterance are few. The second term reflects how far from the target prosody is the selected stream of units.

Thus, at every stage the *n* matching streams of units with the lowest cost are selected, and this is repeated until the end of the utterance. The reason for selecting more than one possible stream at a time is the following. First, the choice of a stream is conditioned by the previous one, because of the continuity term in the cost function. Therefore, different initial streams result in different utterances, and the best (in terms of cost) initial stream does not always lead to the best overall utterance. This requires much processing, and as a result about 60% of the CPU time spent on synthesis is devoted to unit selection, when the second function is selected. This should be reduced as much as possible. Also, the weights involved in the cost function require a long and difficult training, and could use some optimisation, as it affects radically the quality of the output voice. Recent work by Dr. Hunt of Dept 2 is heading in that direction.

Unit selection provides the signal processing module with its main input. So far, each module has been designed separately, but in my view unit selection should take into account the processing performed afterwards (*cf § 6.2*).

3.3 The RUC module

In this section, we present the original signal processing module used in non-uniform unit concatenation, called RUC (for Random Unit Concatenation). It was written in May 1994 by Hélene Valbret, then an invited researcher at ITL and a PhD student at the ENST.

The *RUC* module is based on the PSOLA algorithm presented hereafter, and is roughly schematized in Fig 10. As an input, it receives the units selected as explained in the previous chapter, as well as the target pitch and duration estimated by the corresponding linguistic modules for the current utterance (target power should be available in the very near future).

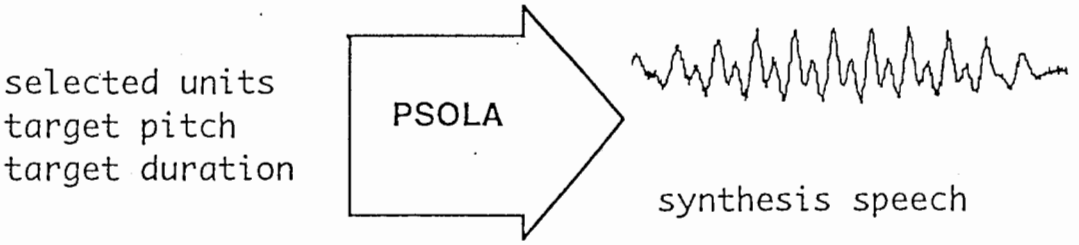


Fig 10. *the RUC module*

So, signal processing is the last step in the process, after all the phonetic and linguistic analysis, has taken place. It plays an important part, since physical manipulation of the waveform can perform the modifications dictated by previous modules, but also additional processing is possible in order to improve the naturalness of the result. However, *RUC* had a number of flaws, which had to be corrected :

1. processing in *RUC* was limited to prosody modification and concatenation, using a single, fixed algorithm. Other possibilities were not explored.

2. the synthesized speech produced by *RUC* was not satisfactory, to the point that the system gave better results when the only signal processing performed was plain (DUMB) concatenation. This could not be caused by the algorithm, but by the implementation, which had to be faulty.

3. the code itself was "badly" written ; comments were scarce, the structure was unnecessarily complex. As a result, direct changes by other researchers after Dr Valbret had left were much too time-consuming, and almost impossible.

3.4 The PSOLA method

PSOLA stands for Pitch Synchronous OverLap and Add, and designates an algorithm for performing prosodic modifications in concatenative synthesis. The principle is presented in Fig. 11, and more details can be found in [3]. Short-Term (ST) signals are the foundation of this method. They are obtained by windowing the speech signal for every pitch mark, using an assymetrical Hanning window. The use of Hanning windows

introduces some distortion in the frequency domain, but it ensures the continuity of the signal at the border of the ST signal.

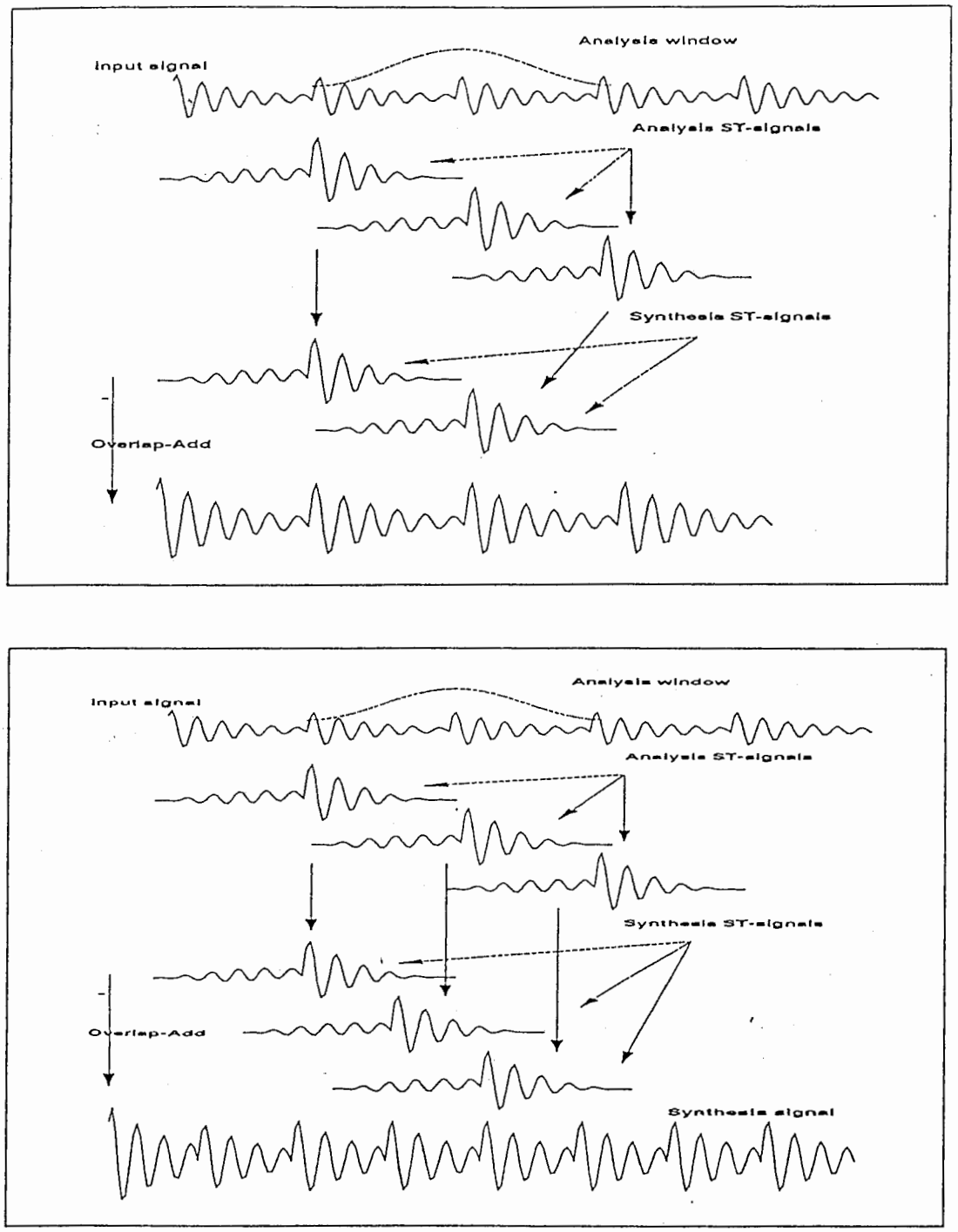


Fig 11. *the PSOLA algorithm*
upper panel) duration modification ;
lower panel) pitch modification

As shown above, duration modifications are obtained by eliminating or duplicating ST signals. For example, an increase in duration of 20% will require the duplication of one ST signal for every four copied. The spacing between the ST signals in the final wave is the same as in the original one. The result can be considered as equivalent to a real wave of the same length.

Pitch modifications are not as simple. The method consists in modifying the time delay between the ST signals in the final wave, thus mimicking a change in the fundamental frequency : setting the ST signals further apart will amount to lowering the pitch, while seting them closer will amount to a raise in pitch. It is important to note, however, that *the resulting wave is not equivalent to a real wave of the desired pitch, but only an attempt to imitate it.* When pitch varies for a given phoneme in real pitch, both the fundamental frequency and the formants change, and such changes are visible in a spectrogram. The PSOLA algorithm can only try to make the listener believe that he or she is hearing a different pitch.

Since its presentation 5 years ago [*3*], PSOLA has been adopted by most laboratories and companies working on concatenative synthesis, and has given good results. This explains the main goal of this project : to implement the PSOLA algorithm in ITL's synthesizer.

Chapter 4

The XPSOLA unit

4.1 Objectives

The *XPSOLA* module has four main goals. Receiving as input the selected units and the target prosody, we want to build the best possible output, in terms of quality of the synthesized voice. This requires that we :

1. perform the prosodic modifications necessary to match the target pitch and duration for every unit. To do so, the PSOLA algorithm will be implemented, but other methods are acceptable.

2. concatenate the units, in a way that minimizes clicks and other noticeable noises that damage the voice quality.

3. use any other kind of procesing to that same effect : to improve the naturalness o the synthesized speech.

4. implement signal processing methods that might be useful in the future, whether for prosodic modifications and concatenation or for some other area.

Tasks 1) and 2) are common to the previous RUC module, while tasks 3) and 4) are new contributions to the system. The only constraint is the implementation of the PSOLA algorithm ; concerning the rest, I was free to experiment.

The new module, however, was expected to yield better results than RUC. If that was the case, *XPSOLA* would take its place as *CHATR's* signal processing module.

4.2 Overview of the unit

4.2.1 architecture

All the functions, data types, data and declarations concerning *XPSOLA* were regrouped in 9 files, in the *ruc* directory. *CHATR* being a large system with a large number of function and variable names, all the new functions written for *XPSOLA* included the *xps_* prefix, making confusions with functions from other modules unlikely. The global architecture is as follows :

xfilters.c 's only purpose is to store for the time being the bank of low-pass FIR filters used in the bands model. This is temporary (*cf § 4.3.3*).

xharmo.c contains the harmonic coefficients (*cf § 4.3.2*) for one speaker's whole database. This is temporary.

xio.c is mostly concerned with input, output and initialisation : reading the waveforms and pitch period files from the database, initializing and freeing the main variables, assembling and returning the final wave.

xmath.c has the routines necessary for FFT-based processing, cepstral analysis, windowing, power estimation and voicing detection

xmisc.c is a set of miscellaneous low-level functions used frequently in the other files.

xmod.c contains the functions relative to the bands and hybrid models for splitting the signal into a harmonic and a stochastic component.

xpros.c is the main file. All the signal processing functions are found here, as well as the mapping and target pitch marks functions.

ruc.c previously written by Helene Valbret and modified to accomodate the new *XPSOLA* unit, has the top-level functions commanding the whole processing.

xruc.h has the declarations of functions visible to other files.

xtop.h has the data types, and most of the constants and macros.

4.2.2 data representation

As a large system, *CHATR* manipulates a large number of data types. Most of them deal with linguistic representations, and are not well adapted to signal manipulation. The original signal processing unit has its own set of data types, but they were not considered for many reasons. First of all, information was not easily accessible, nor clearly presented. Furthermore, the original system is not very flexible, and cannot accomodate easily new algorithms. And finally, since the new *XPSOLA* unit was bound to replace the original one, and would be relatively independent from the rest of the system, changing the data

representation would imply no additional work. Therefore, a new set of data types was created for the *XPSOLA* unit.

The new representation was designed to be clear, concise and flexible. Two top-level data types hold all the information. The first one, *Scheme*, records all the options selected by the user at run-time, and feeds them to the signal processing functions when needed. The second type, *Synthesis,* holds all the data concerning the sampled speech waveform. Both types are structures, and additional fields can be added as the need arises. Figures 12 & 13 give a more detailed description.

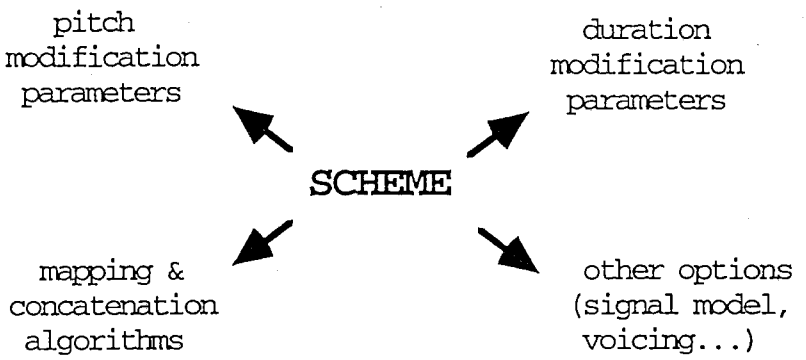


Fig 12. *Scheme data type*

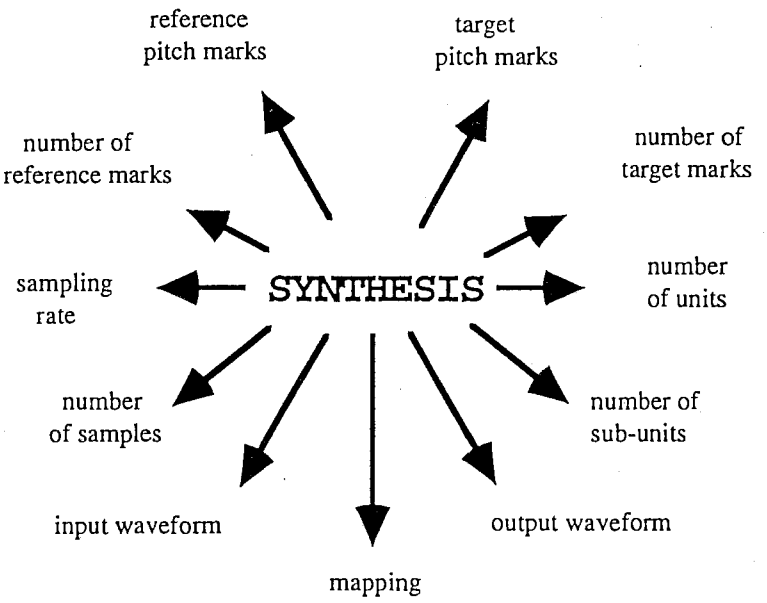


Fig 13. *Synthesis data type*

Every time the module is called, one variable of each type is created. Since *Scheme* contains information vital to the rest of the module, it is initialised before any kind of processing begins, and remains unchanged through the execution. Synthesis, on the contrary, is at the core of the processing, and its contents are modified constantly. Most of

its fields are pointers, and memory allocation must be carefully monitored. Both types are declared as structures.

4.3 The signal model

4.3.1 noise & harmonics

Speech is usually classified into voiced and unvoiced speech. Voiced speech corresponds mainly to the vowels, nasals and liquids, that is, phonemes when the vocal cords are set in motion, vibrating at a certain rate, and giving a pseudo-periodic character to the acoustic waveform. Unvoiced speech, on the contrary, is usually related to the remaining phonemes : fricatives, affricatives, stops, and pauses. Note, however, the existence of "hybrid" phonemes, such as the voiced stops (/b/, /g/, /d/), or the voiced fricatives (/v/, /z/, /ð/).

But, as the latter example shows, this classification is not entirely satisfactory. Voiced speech is not a 100% periodical, but has also an "unvoiced" component. This "unvoiced" or noisy component, no matter how small in amplitude when compared to the periodic component, contributes greatly to the quality of the sound, as later evidence will show (cf 4.3.2). In fact, purely periodical sounds have a very distinctive ring, more related to machines than to living beings. However the same does not always apply to unvoiced speech : the voiceless fricatives /s/ and /f/ present spectrograms lacking any trace of harmonics, with a fair amount of energy in the higher frequencies (/s/), and resembling white noise (/f/, particularly) [*5*].

Therefore, the best way to describe the signal would be to separate the speech signal into a noisy and a harmonic component, both present simultaneously, rather than one at a time when the voiced/unvoiced model is used. The possible advantages in terms of direct applications shall be discussed afterwards.

Modelisation of the noisy component in speech has been a non-trivial task [*9*]. Its origins, however, have been traced, leading to the following classification. Other kinds of noise might result from similar conditions, or from pathological cases, and do not concern us.

- *relaxed vocal cords, obstruction in the vocal tract* : a stream of air is forced through a narrowing somewhere along the vocal tract (e.g., between the teeth and lips for the voiceless labiodental /f/), creating turbulence at the point of constriction.

- *vocal cords relaxed and set together, arytenoid cartilages open* : this creates a small opening at the glottis, creating friction noise. This is the case of whispered speech, in particular.

- *vocal cords vibrating, obstruction in the vocal tract* : the noise is caused by the constriction in the vocal tract but, as opposed to the first category, is also modulated by the vocal cords : voiced fricatives, voiced stops, /r/.

- *vocal cords vibrating, set apart* : the vocal chords no longer interrupts the air stream, but cause audible air leaks, resulting in a breathy voice.

But this does not tell us how to extract the noisy component from speech, nor does it tell us what exactly we mean by noise. As there are a number of possible methods of performing the separation noise/harmonics, each one implying a certain definition of

noise, Nick Campbell suggested I try two methods, presented in the coming sections : the hybrid model, and the bands model. The motivation behind it is two-fold. On one side, the PSOLA method implies some periodicity in the signal ; applying it to both the periodic and aperiodic components of speech might seem inappropriate. The answer lies in splitting the signal in two, using PSOLA on the harmonic part and a better suited type of processing on the noisy part.

On the other side, it might be useful for future applications to implement and test algorithms that perform such a splitting. I am for my part convinced that the noisy component plays a very important role in the naturalness of human speech, which is the main, elusive goal in speech synthesis.

4.3.2 hybrid model

Recently, the CNET laboratories have come up with a "hybrid model" permitting greater prosodic modifications with PSOLA systems [*4*]. It consists of decomposing the signal into a harmonic plus a noise component. The harmonic component is represented in every pitch period by a sum of sinusoids with frequencies that are calculated as multiples of the pitch frequency. The maximum frequency of these harmonics is fixed at a constant level, in the range from 3 to 4 kHz. The noise component is represented by a parameter LPC model, the LPC analysis being performed on the residue which results of the substraction of the harmonic component from the original signal.

The method implemented in XPSOLA differs from the original method for two reasons. First of all, the objectives are not the same. Also, the proposed method has a heavier calculation cost, and some minimal distortion is introduced. The *Ak* and *Bk* coefficients are deducted from the scalar product between the signal, windowed over the current pitch period, and a bank of sine and cosine waves with the above cited frequencies. Thus this harmonic component corresponds to *(1)*,

$$s_h^i(n) = \sum_{k=0}^{K^i} A_k^i \cos(k.\omega_o^i) + B_k^i \sin(k.\omega_o^i) \quad (1)$$

where, *Ki* sets the upper frequency limit, constant for a given speaker, and higher for female speakers than for male speakers. The fundamental frequency is calculated as in *(2)* where *pitch i* is the ith pitch period. The noise component, in this case, is simply equal to the residual, so that both components add up to the original wave.

$$\omega_o^i = \frac{2.\pi}{(pitch^i + pitch^{i+1})} \quad (2)$$

This algorithm was succesfully implemented. As Fig. 14 shows, most energy is contained in the harmonic component. However, the noise component is more intelligible : consonants are perfectly restituted, and even vowels , though dim, are not distorted.

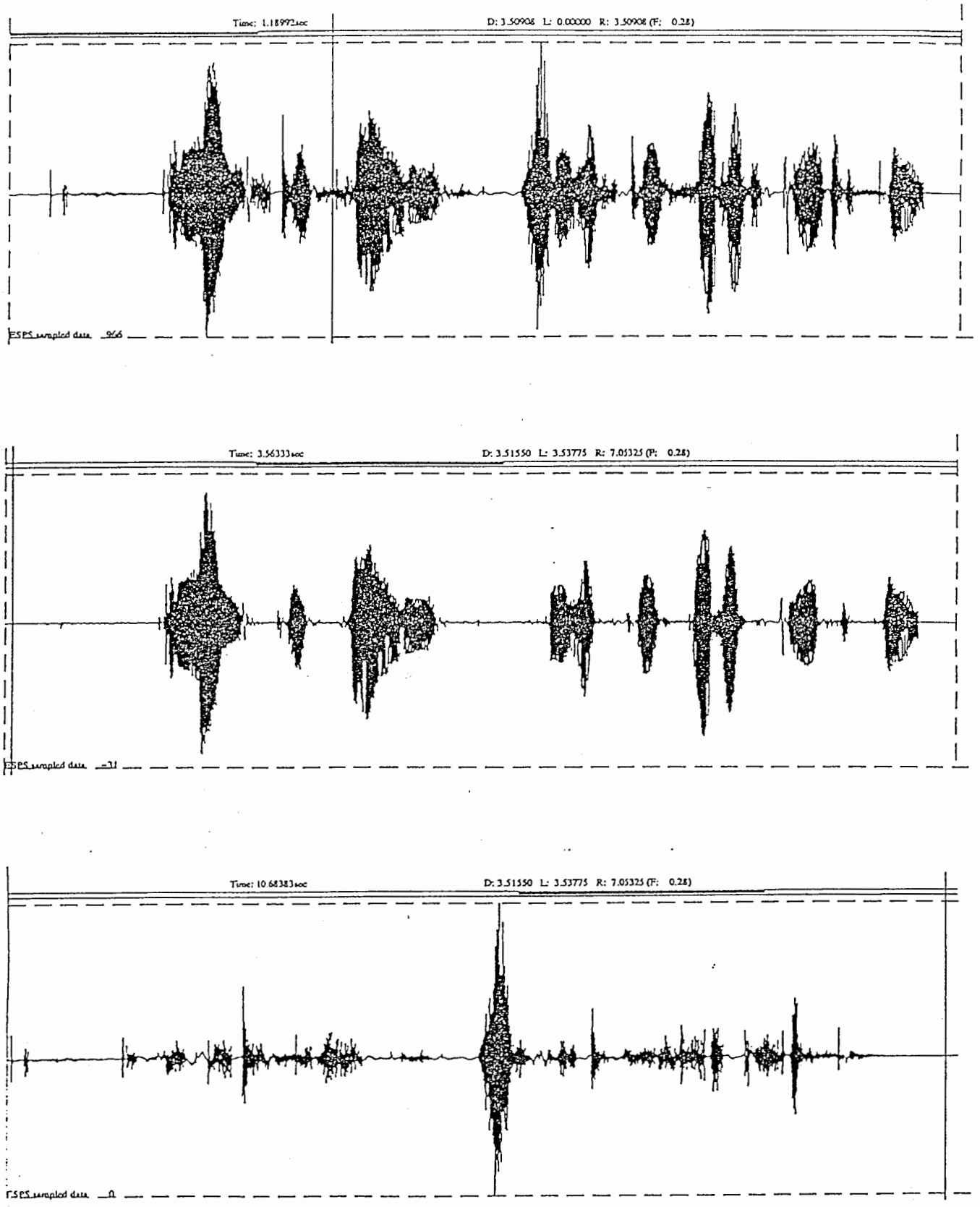


Fig 14. *hybrid model decomposition on the classic utterance "Amongst her friends, she was considered beautiful", 1) original, harmonic component, noise component, 2) original wave, 3) harmonic component, 4) noise component.*

I believe this constitutes a very interesting result. The separation of noise produced by turbulence, obstruction, etc, from the semi-periodic signal in voiced speech is very good, and the time complexity of the whole method is not too high. A question mark still hangs on the way to fix the upper frequency limit for the harmonics. Rather than using a common fixed ceiling, each speaker could have his own ceiling, which could even vary from phoneme to phoneme.

4.3.3 bands model

This method was conceived by Dr. Campbell, and put to work in the course of the project. It stems in part from the following observation : the power spectrograms of natural speech have a certain structure, presenting for example darker stripes corresponding to the formants of voiced phonemes. These stripes, corresponding to frequencies multiple of the fundamental frequency, should account for a rather periodic portion of the signal. *We wish then to detect, for every bandwidth, whether the portion of signal can be considered as voiced or noisy, and to regroup them accordingly.*

Filtering was performed using low-pass FIR filters. Here a trade-off between bandwidth on one side, and speed as well as number of samples needed on the other side, had to be reached. Lower degree filters have a larger transition width (width necessary to ramp from passband to stopband), and that can limit the number of bands used. The final compromise consisted in a set of 200-degree lowpass filters, calculated with MATLAB using the following commands :

```
b = k * sinc(k * (-100:100));
b = b. * hamming(201)';
```

Multiplication by a Hamming window greatly reduces the ringing, and makes for "cleaner" filtered signals. The bandpass filters were simply obtained by substracting two lowpass filters with different cutoff frequencies *k1* and *k2*. Here, *k* represents the normalised cutoff frequency, $0 < k < 1$, and the values used correspond , with a sampling rate of 16 Hz, to the following bandwidths :

0 - 400 Hz	*400 - 800 Hz*	*800 - 1200 Hz*
1200 - 1600 Hz	*1600 - 2000 Hz*	*2000 - 2400 Hz*
2400 - 2800 Hz	*2800 - 3200 Hz*	*3200 - 3800 Hz*
3800 - 4400 Hz	*4400 - 5200 Hz*	*5200 - 6400 Hz*
6400 - 8000 Hz		

Therefore, 13 filtered signals were obtained from the original signal. The last filter, though, was not used ; in order to minimize distortion, the corresponding signal was obtained by substracting all the other filtered signals from the original. The next step, voicing detection for every band, involved the following formula and algorithm:

*Voicing = Ni * rms * ph * (C1 * ac + C2 * cep + C3 * zc), where*

Ni = normalisation coefficient, per band
rms = mesure of power (root mean square)

ph = phoneme information (highest for vowels, lowest for unvoiced consonants)
ac = measure of autocorrelation, based on FFT
cep = measure of periodicity using cepstrum coefficients
zc = number of zero-crossings per time-unit
C1, C2, C3 = normalizing coefficients

```
for every pitch period

   for i in NumBands

      filter signal S with ith bandpass filter -> Si

      calculate voicing degree Voicing = ...

      if Voicing > 0.5

           harmonic component += Si

      else

           unvoiced component += Si
```

This method, however, was soon abandoned in favour of the harmonic model. The following problems, although not critical, demanded special attention, and the investment in time was not judged desirable. The three main reasons for doing so were 1) voicing detection for signals corresponding to different bands of the frequency domain was not an easy task, and would have required some time for finely tuning the coefficients, 2) the filters required 200 previous samples, which are not available at the beginning of units, and 3) the final pair of voiced and unvoiced signals will most likely present discontinuities at every pitch mark, since every filtered signal *Si* is liable to change from voiced to unvoiced and vice-versa at every pitch period.

4.3.4 noise processing

Whereas the logical approach for the harmonic or voiced component is PSOLA, for the reasons stated above, the noise component demands a different kind of processing.

The notion of pitch implies a certain semi-periodic process with a fundamental frequency. This is not compatible with a noisy signal. In fact, it can be easily verifyied that very noisy phonemes such as /s/ and /ʃ/ are perceived as pitch-less sounds, e.g. there is no such thing as a high pitch /s/. Therefore, we only have to be concerned with duration, and the kind of processing required is simplified.

When no splitting is performed on the signal, the following algorithm is used on unvoiced sections in each unit (voicing is defined in § 4.7.2). Here, *m* is the original length of the portion, and *n* is the target length :

```
if m > n

     cut (m - n) from the middle section

else
```

```
repeat
        duplicate the middle 20 % section
until length = n
```

As can be seen, this comes down to simply cutting or pasting. Processing is performed in the middle of the unvoiced section, because it can be asumed that the current phoneme is in a rather stationnary mode during this interval, whereas the borders are often transition periods between different sounds. If that is the case, distortion will be minimal if we cut or duplicate bits of speech that are much alike. This assumption, however, is not always true, and some precautions have to be taken in order to avoid unwanted clicks (*cf § 4.7.2*). In general, though, results were satisfactory, and the joins could not be detected when listening to the resulting waveform.

When the signal has been split into two components, things are a little more complicated. We want to modify the noise component in order to match the target length, but synchronisation of some sort with the voiced component must also be ensured. Indeed, both components are correlated, since they are produced simultaneously, and treating them as completly independent signals could be risky.

Two different approaches were taken, both based on pitch marks. Marks can be relied on as a measure of time, and if the shift between the two components is kept approximately at a friction of one pitch period, we can expect to have sufficient synchronisation. The first method reproduces the algorithm previously shown, in the pitch period level. Samples in the middle of each period are cut or duplicated as required by the target pitch marks. The second method performs a linear intrapolation on every period.

Unfortunately, prosodic modifications using a split signal were not fully tested, as most time was devoted to other, more important parts of the module. I think, however, that it should give good results.

4.4 Prosodic modification

4.4.1 pitch

Two pitch modification algorithms were implemented. Both are attempts at fooling the listener, giving the impression that the pitch has changed ; close inspection shows that such is not the case. Pitch variation for a voiced phoneme is caused by changes in the fundamental frequency. The transfer function might or might not vary in the process, depending on the speaker, but it always does so in a non-linear way. We will see the modifications performed by the algorithms do not correspond to that.

The first method is PSOLA, presented in § 3.4. If we consider the frequency domain, the windowing and time-shifting performed by PSOLA modify the original spectrogram in a way that differs greatly from the natural pitch variation. Explicit details and mathematical theory about this point is found in [3]. The other algorithm implemented for pitch modification consists in a linear intrapolation, performed on every pitch period. If we call *Wo* and *Wi* the output and input waves,

$$\omega_o[j] = \alpha.\omega_i[j.k+1] + (1-\alpha).\omega_i[j] \qquad (3)$$

$$\alpha = j.k - E(j.k)$$
$$k = \text{scaling factor}$$

Again, this second method can be interpreted in the frequency domain as another linear interpolation, which does not correspong to the real, non-linear event. Tests showed that PSOLA was clearly superior, as intrapolation sometimes produced very artificial-sounding speech, but not completly satisfactory. Seemingly, pitch modification is not an easy task.

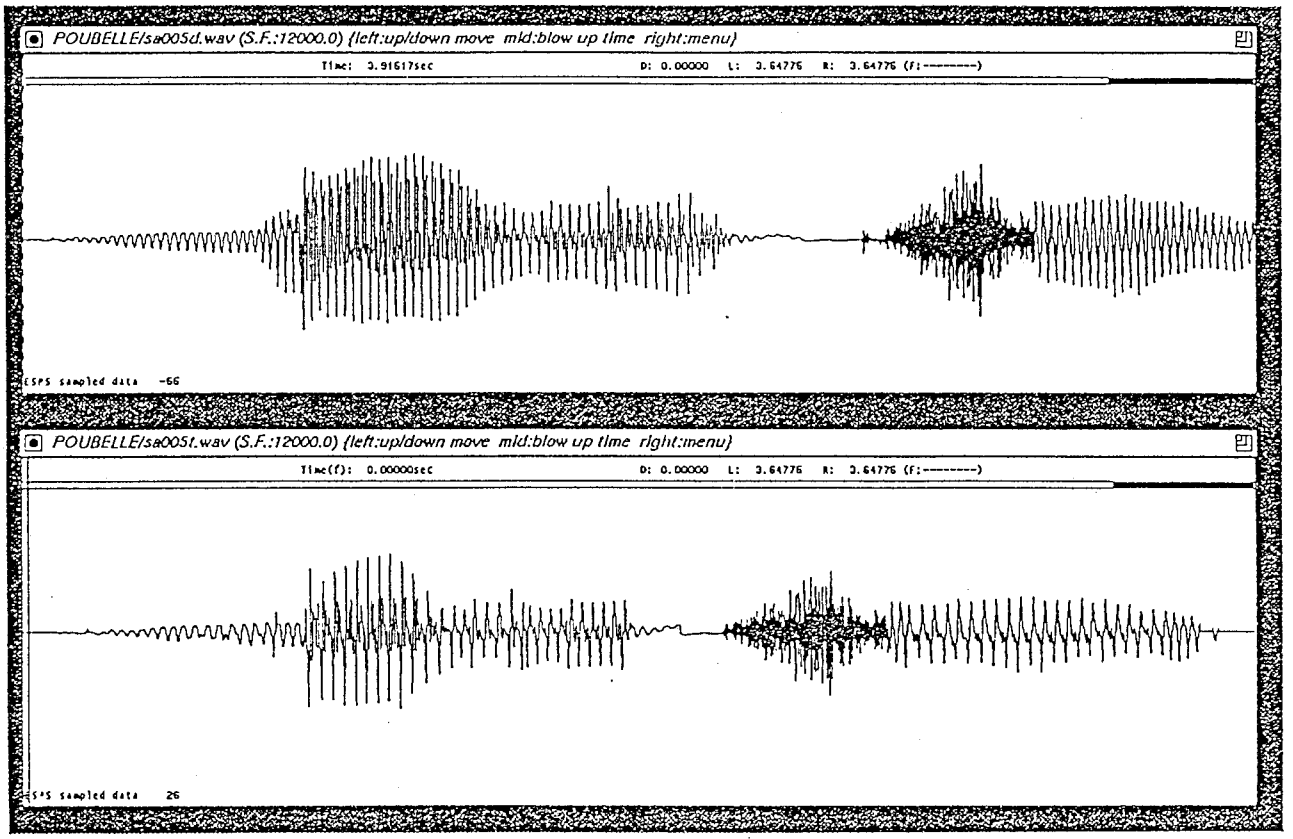


Fig 15. *PSOLA pitch modification*
upper panel : original ; lower panel : modified

4.4.2 duration

Duration modification is performed by cutting or duplicating whole pitch periods. The problem to solve here are the discontinuities that can occur at the joins. Because the signal is modified in a way that parts of the wave that were originaly many milliseconds apart are suddenly put together, both the slope and the sample value of the wave at such points are most likely to be different, and any such discontinuity will undoubtly be noticed by the listener. In order to deal with that, three algorithms were implemented. Besides PSOLA, one method consists in removing the discontinuities by performing a weighted average, as shown below.

```
for j = -15 to -1

    Wo[j] = (-j/15).Wi[j] + ((15+j)/15).M

for j = 0 to 4

    Wo[j] = (j/4).Wi[j] + ((4-j)/4).M
```

where $$M = \frac{\sum_{-15}^{-1}\omega[i]}{30} + \frac{\sum_{0}^{4}\omega[i]}{10}$$

Here, *Wo* and *Wi* are respectively the output and input signals, centered at the pitch mark where the current join takes place. The weighting is assymetrical, in the sense that the five samples to the right of the pitch mark weight as much as the 15 samples to the left. This is because the signal is typically weakest just before the mark, that is, left of the mark. Averaging the signal introduces distortion, which we want to minimize. This kind of distortion is also present, to a lesser extent, in PSOLA : adding windowed ST signals means that the output will somehow be averaged and flattened. That can be prevented only if the pitch marks are reliable enough.

The final method is called *select cut point.* has the following advantage over the two previous methods : no averaging is done to ensure the continuity, and the original signal shape is preserved. If, rather than joining bits of the signal at exclusively pitch marks, a better pair of join points can be found, additional processing will not be necessary. Join points should fulfill the following conditions, 1) slope and sample value should be as close as possible, and 2) the lapse of time between them should be equal to the lapse of time between the corresponding pitch marks, i.e. equal to the pitch period. Thus the algorithm for *select cut point* :

```
for j = 0 to m

    D[j] = sq(W[pi-j] - W[pi+1 - j])

Cost[j] = D[j-1] + D[j] + D[j+1]

find j, 0≤j≤m, minimizing Cost[j]
```

where *sq()* is the square function, *Pi* the ith pitch mark, *j* the shift, in number of samples to left of the pitch mark, and *m* the maximum shift tolerated, fixed at one pitch period. The shift that minimizes the cost points to the place where the join is to be made. As could be expected, PSOLA and *select cut point* give the best results. The resulting duration modifications introduce almost no distortion.

Summing up, duration modification produces less distortion than pitch modification. All the algorithms described had a complexity cost of the same order of magnitude (*O(n)*), though PSOLA requires slightly more CPU time than the rest. Therefore, output quality is the only criteria, and the best choices are PSOLA for pitch, and either PSOLA or *select cut point* for duration.

4.5 Mapping

4.5.1 overview

Mapping acts as the heart of the module. It is responsible of analysing the target prosody and the selected units, and deciding precisely for every unit which processing is required in order to reach the desired output. Mapping can also be described as the intelligent interface between the prosody generation from the linguistics modules and the signal processing routines.

Target duration is generated for every phoneme in the utterance, corresponding to a sub-unit in the stream. It is stored in its corresponding field in the structure. Pitch, on the other side, has a different format. It is produced as an array of integers, each value a fundamental frequency estimation in Hertz, at fixed intervals (usually an estimation every 5 milliseconds). In unvoiced sections, the prosody generating modules create fake values, in order to make the whole f0 contour continuous. A function, *xps_Periods(t, ...)*, puts all this information into an easier to use format : it returns the pitch period length, in milliseconds, estimated at instant *t* in the utterance. At this point, we might be tempted to use this information directly, but there are reasons to be cautious :

unreliable pitch marks : we rely completly on them for measuring the fundamental frequency of the selected units. And while errors in the pitch mark databases are uncommon, they do exist. A single flawed mark is enough to deteriorate the speech signal if we are not careful.

unvoiced zones : near the voiced/unvoiced transitions, the f0 estimation can be slightly unaccurate.

micro-prosody : selected units have their own natural intonation ; forcing them to a different one might sometimes produce unwanted effects.

synchronisation : the fundamental frequency estimation is not exactly synchronized with the actual utterance.

4.5.2 algorithms

The kind of output we expect from the mapping is :

unvoiced signal	target length / original length (float) *for every sub-unit*
voiced signal	copying decision : cut, copy, or duplicate target f0 / original f0 (float) *for every pitch period*

Unvoiced signals are easy to process, since we need not bother with pitch. For voiced ones, for the reasons cited in the previous section, some precautions are needed.

There are 3 different approachs at mapping prosodic modifications for voiced signals. The algorithm is :

```
for every pitch period i

        P'i = 1 / f0i
or
        P'i = Pi . (1/N) Σ_j^N 1/f0_j
or
        P'i = (1 / f0i + Pi . (1/N) Σ_j^N 1/f0_j) / 2
```

where `P'i` and `Pi` design, respectively, the target and original pitch period length. The first approach, exact matching, is dangerous, and can sometimes give bad results for the reasons cited above. The second approach, using the average pitch period value, preserves the original intonation of the unit, shifting the pitch by the same value for every period, and ignores most of the information passed by the f0 generator. The third method, a mix of the previous two, is perhaps the best.

Duration modification for voiced sections is as follows. Pitch modification has to be considered here, because raising the pitch of a signal will make it shorter, and vice versa. The variable used is :

$$dmodif = \frac{target\_length * target\_pitch}{original\_length * original\_pitch} \quad (5)$$

Finally, the algorithm for cutting/duplicating is shown below. As can be seen, I avoid cutting and duplicating near the borders of a unit, because I think transitions between phonemes are too important and fragile to be modified. Also, at most one every two periods can be duplicated/copied. This is an additional precaution against excessive prosodic modifications. That way, duration can at most be doubled or halved, which I think should be enough. If more was required, that might mean the unit was not properly selected.

```
aux = 0

for every pitch period i

    aux = aux - (1 - dmodif i)

    if abs(aux) < 1
    or previous period was cut/dupplicated


        period must be copied

    if aux > 1

        period must be dupplicated
```

```
        aux = aux - 1

if aux < -1

        period must be cut

        aux = aux + 1
```

4.6 Unit concatenation

4.6.1 goals

Each selected unit is extracted from a certain word, in a certain utterance with a certain syntax. This means that every unit stems from different vocal trat configurations and intonation patterns ; concatenation is bound to create discontinuities. Consecutive units, with similar power, and whose waveforms are continuous at the join, can still sound wrong because *no human being could have pronounced them*, as the airflow, tongue and lips position, etc, can not change that fast.

This is a difficult problem, that can best be addressed by unit selection, carefully choosing the unit boundaries, so that the shape of the waveform on both sides of every join is similar. Signal processing can, however, improve the result when concatenating the final units.

The following sections present briefly a set of concatenation methods. The next chapter, though, presents all the other ways of improving the quality of the synthesized speech.

4.6.2 temporal methods

Here the problem is very similar to duration modification : we want a continuous waveform, without jumps in the signal or the slope. Two comments are necessary, though. First, concatenation is done at pitch marks. Otherwise, we would interrupt the periodicity of the signal during voiced speech, introducing a phase shift. Second, in most cases a whole pitch period of signal will be dropped for every pair of concatenated units. Keeping the whole units and making good joins are not compatible.

The three methods implemented are DUMB, *select cut point* and PSOLA. The first approach consists in concatenating without any kind of processing. The other two are inspired from the prosodic modification algorithms. select cut point finds a good point for concatenation, while PSOLA builds a middle period by adding the right and left sides, respectively, of the ST signals of the left side and right side (with respect to the join point) units. In both cases, a whole pitch period is lost.

Listening tests show that *select cut point* gives the best results. Adapting PSOLA to concatenation is not very succesful because, contrary to the duration modification context, the ST signals are not similar to the point that overlapping and adding does not result in averaging and flattening of the signal.

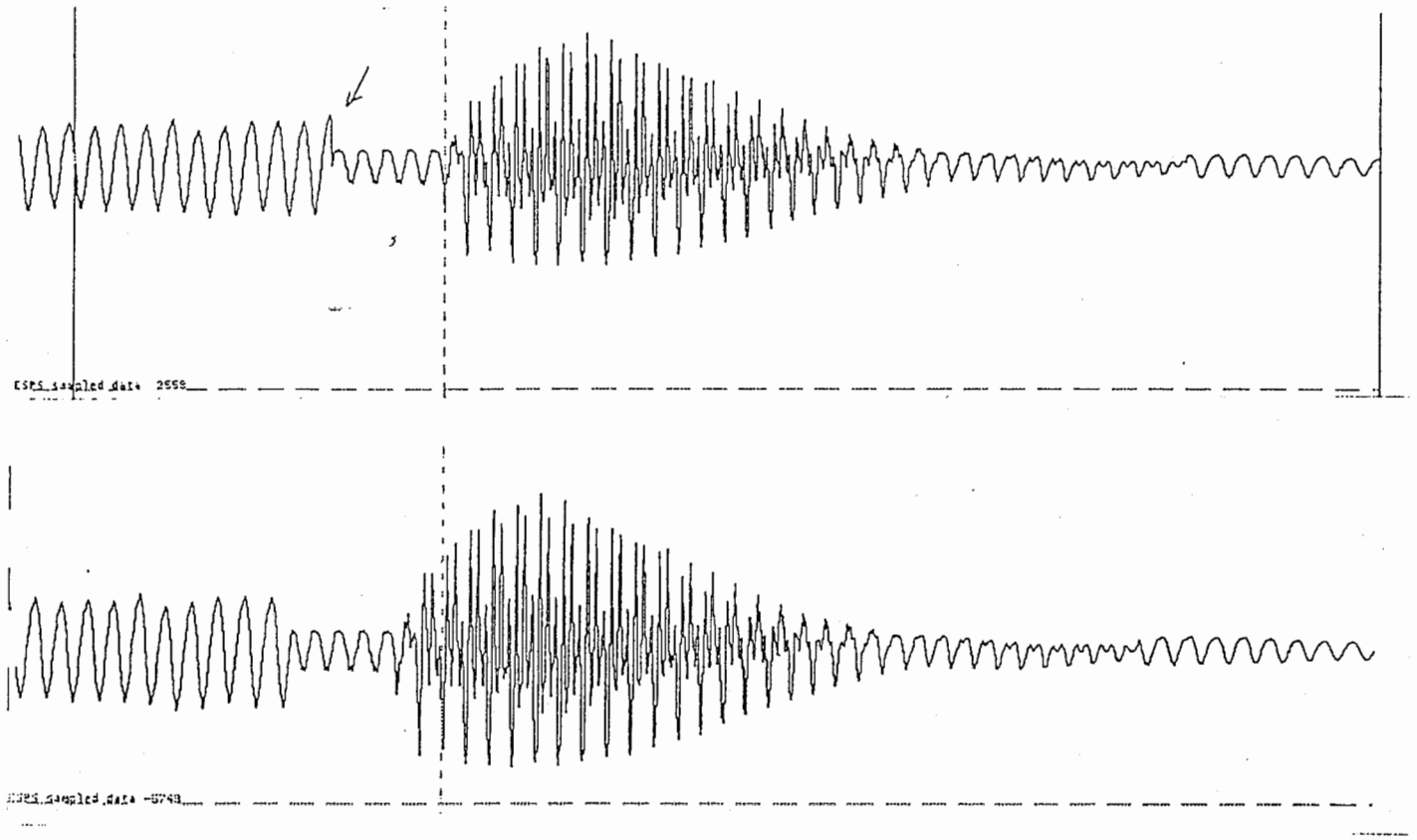


Fig 16. *select cut point concatenation*

4.6.3 spectral methods

The idea behind the following algorithms is to build a transition signal between the units to be concatenated, in order to smoothen the transition between phonemes. Nothing really convincing was obtained, because the algorithms used are quite simple and naive, and the task requires more advanced processing and knowledge, such as a reliable distance measure for waveforms. The first algorithm is based on the following formula :

$$\text{iFFT}\left(\frac{\text{FFT}(\text{signal\_1})+\text{FFT}(\text{signal\_2})}{2}\right) \quad (6)$$

This is a crude attempt at averaging two pitch periods in the frequency domain, using Fourier transforms. Power and length were averaged between the two periods. The second algorithm attempted the same thing, in the temporal domain. Both failed.

4.7 Other options

4.7.1 power modifications

By power we mean the energy of the speech signal, which can be measured by a *root mean square* of the waveform. It plays an important role in prosody, i.e. power is part of stress, and affects the meaning of the sentence. There is so far no power estimation module in CHATR, although that should change. Nevertheless, power modification turned out to be an easy process, that improved considerably the quality of speech.

Power can vary greatly between different occurences of the same phoneme. It varies also during any given utterance. But, with the exception of plusives, power usually rises and falls in a continuous way. Power discontinuities are very rare, and in the speech synthesis context, unwanted. Again, plusives are not concerned.

As selected units are concatenated, the envelope of the resulting signal is likely to show several discontinuities. The same will happen with power. Therefore, an algorithm allowing to erase those discontinuities was implemented.

```
for every unit i

    rms1 = last half of previous sub-unit
    rms2 = first half of first sub-unit
    rms3 = current unit
    rms4 = average value for the unit's phonemes
    rms5 = last half of last sub-unit
    rms6 = first half of following sub-unit

    calculate the 3 scaling coefficients

    build scaling window

    signal = signal * window
```

The scaling window is a set of coefficients, one per sample : a 1.3 value makes for a power increase of 30%, and so on. This window has 3 zones, with smooth, sygmoid-like transitions in between. The middle zone covers all but the borders of the unit, and the corresponding coefficients are meant to drive the power of the unit, halfway between its current value and the average value defined in the database for the unit's phonemes. This tends to produce a kind of speech with little power variation. As a result, it might be slightly monotonous, but synthesized speech will be more predictable and stable.

The two other zones correspond to the first half of the first phoneme, and the last half of the last phoneme of the unit. Here we want to ensure a certain power continuity with neighbouring units. Scaling is done in a way that reduces power gaps by half. For more details, consult functions *xps_INDEX*, *xps_WIN_POW* and *xps_POWER_PRO* in the program.

It is also possible to feed the XPSOLA module with target power values. Completion of the power estimation module with make testing possible.

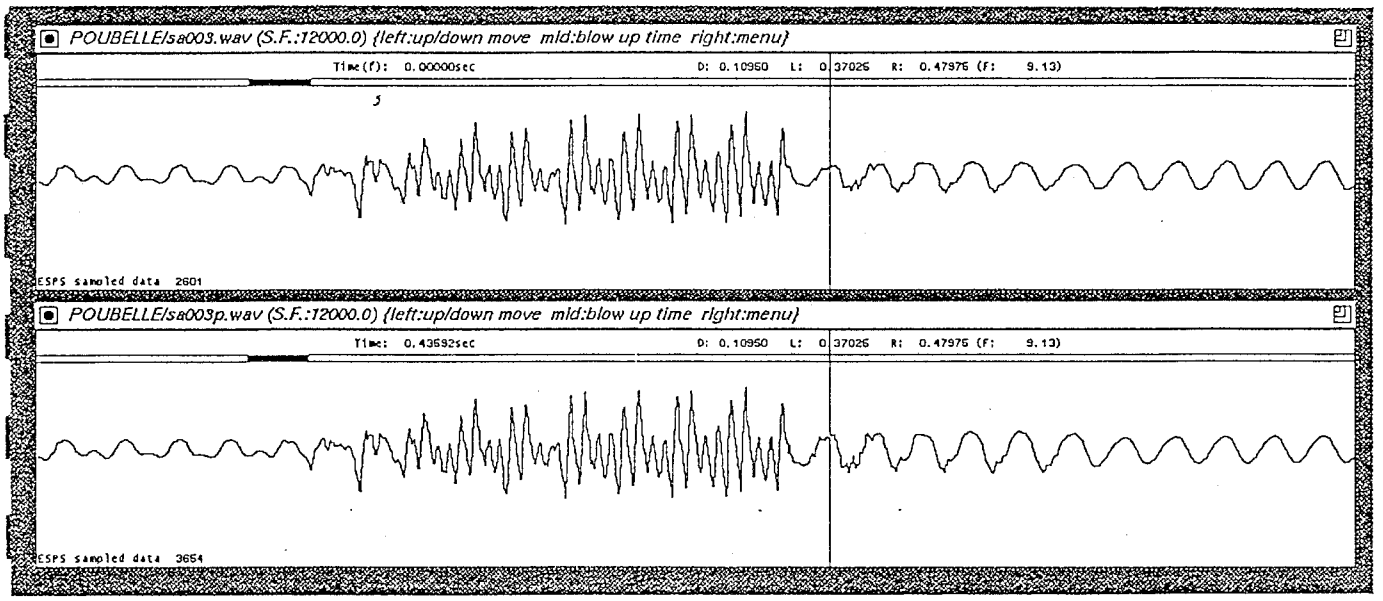


Fig 17. *power modification*

4.7.2 final modifications

To close the chapter, let us briefly mention two last aspects of the processing performed on the synthesized speech, *fading* and short units. Fading consists in repeating the last 3 pitch periods of the utterance's last unit, with diminishing power. This simple algorithm is designed to avoid sentences that end abruptly, because the last selected unit is not silent, but loud, and sudden cuts should be avoided. Concerning short units, my opinion is that, under a certain threshold (5 milliseconds), it is best to drop them from the final wave. Given the kind of processing performed so far, very short units can not be properly treated, and introduce very noticeable clicks.

Other, less important options, not mentioned in this chapter, are briefly explained in Chapter 5.

Chapter 5

Implementation

5.1 Software constraints

As any large software system designed by several people, CHATR features some common rules that programmers must respect. Although CHATR is flexible, it is impossible to be flexible enough for all users. Therefore, as any person contributing a new module to the system, I had to abide by the following dos and do-nots :

- **never call `exit` or `printf`** : there are many modules in CHATR, too many things to print, and too many conditions were it is impossible to continue execution. Rather than exiting the program, or filling the screen with messages, specially made functions like `list_error` and `P_Message` should be used.

- **never use absolute path names or machine dependent functions** : CHATR is designed to be used on many different machines and platforms, so the system must remain portable.

- **do not add unnecessary functions to the name space** : since the number of functions in the systems runs in the thousands, each module should keep its functions static, or else use a given prefix to avoid confusions and visibility problems. The prefix adopted for XPSOLA is *xps_*.

- **always use CHATR functions to access CHATR structures** : internal aspects of CHATR can change and improve. It is much easier to fix one general function than hundreds of random little functions that do similar tasks all wrongly.

The previous rules being observed, the XPSOLA module was linked to the rest of the system. Also, I was encouraged to add many comments in the code I produced, for

obvious reasons. A new entry in `chatr_vars.c`, a kind of on-line help for *CHATR* users, was added, describing the basic options and the correct syntax.

In line with every other module in *CHATR*, *XPSOLA* was debugged using software that spots hard-to-find memory errors [12]. Given the very large amount of data the system is bound to manipulate, no memory corruption can be tolerated. Otherwise, it would be bound to crash. Purify allows to avoid errors like overwriting, writing on unallocated memory, reading uninitialized memory and forgeting to free the memory used when leaving the system. All this is done at run-time, through a very convenient graphic interface.

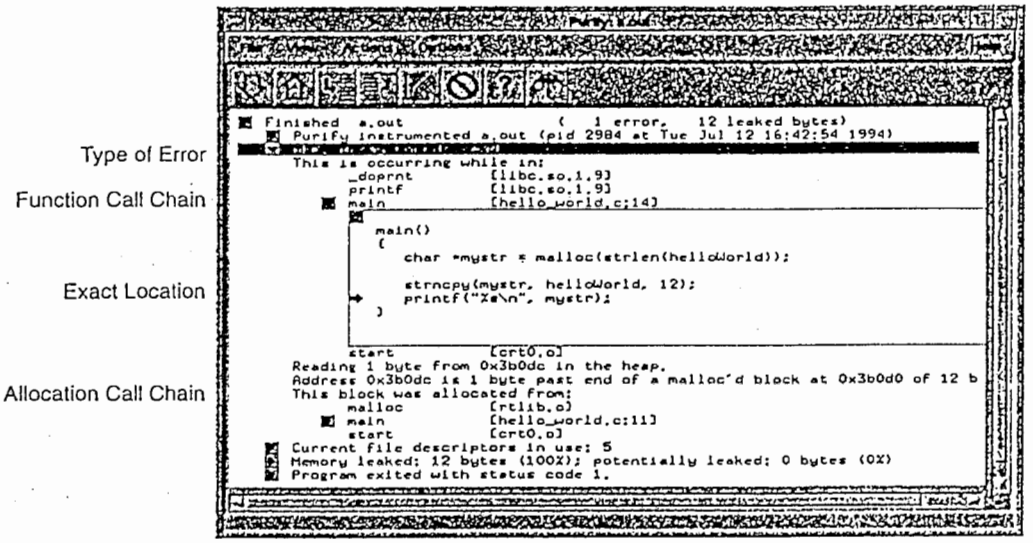


Fig 18. *Purify window*

5.2 Testing

Testing of the module went as follows. The first step consisted in testing that the system will not crash with any given utterance or database, and that no memory errors are detected by **Purify**. The second step is about listening and displaying the output waveforms, looking for and correcting possible mistakes. The whole testing process was performed twice on the module, once for the original type of pitch marks, where it was completed satisfactorily, and once for new version accommodating both the original and the new types. This second testing process was not fully completed (*cf § 6.2*).

This, however, was not enough to detect errors easily. Nor could one tell which flaws in the output signal were due to bugs, to the selected units, or to the algorithms. Therefore, a special testing procedure was developed inside XPSOLA. When used to test the processing, it gave a clear view of what exactly was being performed on the signal. When the signal model was our concern, it simply displays the noisy and harmonic components; this was shown in Chap 4.3.3. The algorithms for these two routines are :

```
for i in [0, 7]
   display original unit OUi
fill 1000 blank samples
apply prosodic modifs, concatenation -> processing Pr
for i in [0, 7]
   display processed unit PUi
fill 1000 blank samples
replace original units with sawteeth
apply same processing Pr to new sawteeth units
for i in [0, 7]
   display processed unit SWTi
```

Figures 12 shows the results from the *test processing* routine, using xwaves+. Some comments are necessary to explain the unusual waveform. On the right end, the processed sawteeth are displayed. The first sawtooth is the longest, and corresponds to a silence, as can be seen on the other two waveforms. Pauses can be made longer by filling them with 0s, thus the "hole" in the middle of the sawtooth. The fourth unit corresponds to an unvoiced consonant, so shortening the unit can be achieved by simply dropping the middle samples, thus the discontinuity. These last two modifications might seem radical, but the listener will not notice any disturbance. The second, third and eighth sawteeth correspond to voiced phonemes : the distortion is caused by the windowing involved in the extraction of short term signals. A closer inspection will reveal their sine-like shape.

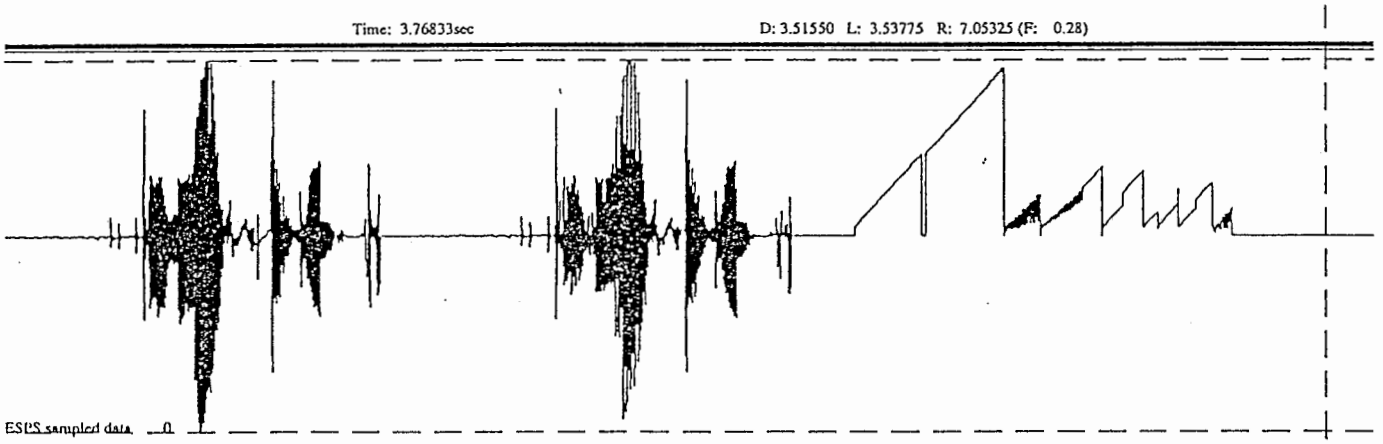


Fig. 19 *test processing*

Since in most cases every database (recorded speech or pitch marks) was built separately, sometimes by diferent persons using different conventions and tools, the system must cope with this variability : formulas for power may differ, some marks in the database may be defective (i.e. pitch marks going "backwards"), etc. Also, some units might be unexpectedly long, or unexpectedly short. In the first stage of testing, we hope to encounter all those extreme cases, and deal with them effectively.

The "modus operandi" was to run CHATR with **Purify** over a whole database, using the command `test_nameofspeaker`. This consisted in, for every utterance in the selected database, synthesizing it after removal from the set of possible units. The listening stage was performed mostly for the English databases, because my poor level in Japanese might have influenced my judgment. Displaying the waveform using **xwaves+** made possible to spot eventual bugs, and helped to trace them back to the code.

5.3 Evaluation

The quality of a synthesized waveform is relative to the listener. Each individual might perceive the naturalness of the artificial speech in a different wave, save for the most obvious mistakes. Therefore, the best way of evaluating an algorithm in a speech synthesis system is to present a representative number of examples to a sufficiently large group of "naive" subjects, who grade each utterance.

This kind of test was not conducted while evaluating the module's performance, though, since it requires time and people, and often the difference between algorithms was big enough to allow to distinguish a clear winner. Therefore, evaluation consisted simply in listening, together with Dr, Black, to a few utterances, spot the problems, and choose the best algorithms. The algorithms that proved to be superior are :

pitch modification : PSOLA better than intrapolation

power modification : smooth better than none

pitch marks : PM_VOICED_ONLY over PM_ALL_MARKED

In other areas, the situation is less clear. These would require more attention. In general, all the algorithms had an equivalent cost, so speed is not an issue when deciding which algorithm should be preferred. The choice for default options was then made by picking the clear winners when possible, and picking the most standard or reliable algorithm if not. These default options are presented in the next section.

5.4 User's guide

In this section, the practical aspects of the XPSOLA unit will be presented. As shown in Chapter 4, there are a number of parameters, each having a default value, that might be set by the user at run-time. Therefore, no modifications to the code are required when switching from one algorithm to another.

Usually, CHATR is called using a script, containing a number of instructions and parameters, each addressing a particular area in the processing. For prosodic modifications, concatenation, and signal processing as a whole, the corresponding variable is called `Concat_Method`. Currently, possible values are :

XPSOLA	remember ?
PSOLA	previous version, implemented by Hélène Valbret
NUUCEP	Nuutalk cepstrum resynthesis
PS_Simple	PSOLA with cepstrum cutting
DUMB	simple concatenation, no prosody modifications
DUMB+	concatenation at zero crossings, no prosody modifications
NULL	empty wave

When XPSOLA is selected, an additional variable named `xpsola_params`, containing all the desired options, migth be set. Options that are not set by the user, or that are set to unacceptable or out-of-range values, will be assigned to their default values. Input is case-independent. Information thus stored into `xpsola_params` is then passed on to the global variable scheme, used in the XPSOLA unit (see Appendix 1). The list of options is :

P_thres	default value : 0.1 pitch modifications under (P_thres * 100) % will be ignored, i.e. not accomplished, during the processing. possible values : greater or equal to 0
D_thres	default value : 0.1 duration modifications under (P_thres * 100) % will be ignored, i.e. not accomplished, during the processing. possible values : greater or equal to 0
P_method	default value : PSOLA this selects the pitch modification algorithm used. possible values : intrapol, psola
D_method	default value : PSOLA this selects the duration modification algorithm used. possible values : dumb, select, psola
P_ceil	default value : 0.33 pitch modifications are limited to +/-(P_ceil * 100)%. possible values : greater or equal to 0
D_ceil	default value : 0.66 duration modifications are limited to +/-(D_ceil * 100)%. possible values : greater or equal to 0
concat	default value : SELECT this selects the concatenation algorithm used.

possible values : dumb, select, psola, cepstral, extrapol, extrapol2

power default value : SMOOTH
this selects the power modification algorithm used.
possible values : none, target, smooth

contour default value : AVERAGED
this selects the mapping algorithm used.
possible values : averaged, target, mixed

model default value : NONE
this selects the model used for noise/harmonics decomposition, and eventually the algorithm for noise processing.
possible values : none, hybrids, bands, hybrid+, bands+

test default value : NONE
this selects the testing routines.
possible values : none, model, processing

stops default value : NONE
this selects the kind of modifications performed on stops.
possible values : none, pitch

tiny default value : DROP
this selects whether units under 2 periods long are used or not.
possible values : drop, keep

voicing default value : PHONEME
voicing decision for PM_ALL_MARKED pitch marks.
possible values ; phoneme, database

The defaults were chosen in a rather conservative way : the simplest, most reliable, but not necesarily best, algorithms are used. However, they can easily be changed, by modifying the function *xps_set_xpsola_params* in the ruc.c file (see Appendix 1). A possible change could be changing power from NONE to SMOOTH, and contour from AVERAGED to MIXED. An example of a script setting these variables could be :

```
(speaker_sally)                          ;;selecting the sally database

(Parameter Concat_Method XPSOLA)

(set xpsola_params '(( model HYBRID+)
                     (test model)        ;; case independent
                     (tiny KEEP)))
```

Chapter 6

Conclusion

6.1 Summary

This report presented a new module to be implemented in version 7 of CHATR, the speech synthesis system developed at Department 2 of ITL. It is based on the PSOLA algorithm for prosody modification, but it offers many other possibilities as well. It provided CHATR with its first signal processing unit, needed to fully exploit the prosody generation capabilities of the system, and to improve the overall quality of the output synthesized speech. It also allowed to test new techniques concerning pitch mark generation and signal splitting into voiced and unvoiced components.

For future users it presented the potential of the system, as well as the way to exploit it. Future programmers found the different algorithms presented in detail.As with the rest of the CHATR system, the XPSOLA module is bound to evolve with time, and comments as well as contributions are expected.

6.2 Fields for improvement

As with any system, this module can be improved in several ways. As the new XPSOLA module is fairly young, future users are invited to make the changes they deem necessary. Possible fields for improvement fall into one of the following categories.

1. First of all, the final stage of debugging should be finished. The way to do this is to run CHATR using purify, and to try all the examples for one or more databases, with

different options. This was completed succesfully with the previous version of the module, before adding the modifications needed for treaiting the new pitch marks type (cf 2.2.3).

2. The module was conceived in order to offer different options, and speed was not a priority. Even though signal processing only takes about 15% of the CPU time needed for synthesizing an utterance, it is still important to reduce as much as possible the processing time if CHATR is to work as an almost real-time system. Moreover, the percentage of time spent on unit selection is bound to fall in the future. So optimisation of some kind is necessary.

3. The optimal set of options, that is, PSOLA pitch and duration modification, power smoothening, and "select cut point" concatenation, can be deemed satisfactory : the sound quality is fair, and no clicks or major distortion introduced. These algorithms are straightforward and reliable. The mapping, however, is more complex, and can certainly be improved.

4. Also, some sort of distance measure between waveforms would be welcome. It could be used in particular for concatenation and for unit selection, helping to determine the best points for joining consecutive units, or even for creating "fake" frames to smoothen the join.

5. Finally, a comment on unit selection for stops. The burst is a very sensitive segment, and can easily be corrupted, with very noticeable effects. Therefore, it migth be wise, not only to avoid prosody modifications on stops, but also to make sure that the burst is well in the middle of the selected unit, and not shared between two consecutive ones.

6.3 Discussion

A number of comments can be made on the previous sections.

1. Each individual has a particular voice, recognized easily by humans, but not by automatic machines. Difference in voice quality between speakers has long interested researchers, and can be traced back to three causes [2]. First of all, the overall dimension of the vocal tract, and the relative proportions between the supra-glottal cavities (laryngeal, oral, nasal,...) present important variations from one speaker to the other. As a result, formant frequencies differ from speaker to speaker. The second factor is linked to the glottal excitation, ie the vocal cords (an opera singer is said to have "good" vocal chords). Finally, the third factor is linked to the person's speaking habits, dictated by his dialect, and social environment, making for a particular prosody.

The latter is linked with linguistics, and does not concern us here. However, when using the hybrid model for splitting the signal into a harmonic and a noisy components, we have direct access to the harmonics of the speech signal, and thus of the different formants. *It would then be possible to map the formants of one speaker to those of a target speaker.* Such mapping might be performed using neural nets, for example, trained for that particular pair of speakers, over the whole database. It might be the case that each phoneme type will require a specifically trained net, or a a single net will be able to trace all the phonemes, effectively representing the overall differences between the vocal tracts.

Two questions remain open. On one side, the noisy component carries a considerable amount of information, and will probably require some sort of processing too. The nature of such processing has yet to be determined. On the other side, we don't

know to what extent the difference in glottal excitation has been addressed, nor how it really affects the voice quality.

I think this idea is worth trying. XPSOLA offers a good starting point, and is flexible enough to accommodate many different test routines and waveform manipulations. If succesful, this method would allow modification of the voice of one speaker without modifying its prosody, ie its particular accent (Scottish, English, Australian, upper class,...).

2. Unit selection should take into account the signal processing that follows : for any given utterance, the units selected for DUMB concatenation (no modifications whatsoever on the units) may not be the best ones for PSOLA processing, power modification and "select cut point" concatenation. Although this requires a new series of testing, some fair guesses can be made.

Since duration is relatively easy to modify, and the results are rather good, duration should play only a minor role during unit selection. Power too can be modified so as to eliminate power bursts. Pitch, however, is more difficult to change. Another kind of problem is the shortest units, usually two or three pitch periods long, that are less than ideal for signal processing. Therefore, while training new weights for unit selection, it would be useful to have in mind that :

- *Wd*, weight linked to duration, shoukd be low
- *Wr*, weight linked to rms (power), should also be low
- *Wp*, weight linked to pitch, should be high
- very short units (ie, < 5 milliseconds) should be penalised

Acknowledgements

I wish to thank all those who helped me in many ways to carry out this project :

- Nick Campbell, who agreed to be my *Directeur de Stage*. His experience and knowledge were extremely valuable.

- Alan W. Black, who greatly contributed to this project. He took the time to introduce me to many diferent aspects of CHATR in particular, and of speech processing in general.

- Lee Yang-Hee, for the many presentations of his research on cepstral analysis.

- Andrew Hunt, for his sound advice in speech processing and computer science.

- Masahiro Nishimura, the department's systems engineer, who always solved any of our problems with either hardware or software.

- Kevin Lenzo, who explained certain aspects of speech-related digital signal processing, in particular the fft-related functions.

- Yoko Shibata and Chieko Kohshima, thanks to whom my adaptation to a new environment turned out to be a very enjoyable experience.

- Norio Higuchi and all the other members of the PEGASUS group.

Finally, I wish to thank Ms. Dorizzi, who agreed to supervize this internship and to be part of the jury, as well as Shuko Lei and Satoshi Ikeda, Japanese teachers at the INT, who are responsible for my discovering this wonderful country.

```
/*=========================================================================*/
/*           A T R Interpreting Telecommunications Labs                    */
/*                                                                         */
/*=========================================================================*/
/*                 CHATR Speech Synthesis System                           */
/*                        Christian Lelong                                 */
/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                      Input - Output  Library                            */
/*                                                                         */
/*                                                                         */
/*                            Feb 1995                                     */
/*                     Copyright (C) 1994,1995                             */
/*     ATR Interpreting Telecommunications Research Laboratories           */
/*                      All rights reserved.                               */
/*=========================================================================*/


#include "xruc.h"

#define F0_FRAM 5


extern int SPAN;


Small_pm *make_small_pm(int npm);
void free_small_pm(Small_pm *pm);


/*************************************************** Classify a Phoneme **/
/*   Read a phoneme type :                                                 */
/*      This simply returns a value according to a classification table,   */
/*      telling the phoneme type of sub-unit 'su'.                         */
/*************************************************************************/


int xps_PHONEM(struct Sub_Unit *su)

{
    int aux, output, vowel, cons;


    vowel = SC(su->target, Segment)->vowel;
    cons = SC(su->target, Segment)->c_type;


    switch (cons)    {

        case  STOP    :  output = (vowel==1) ? 1 : 7;
                         break;
        case  FRIC    :  output = (vowel==1) ? 1 : 5;
                         break;
        case  AFFRIC  :  output = (vowel==1) ? 1 : 4;
                         break;
        case  NASAL   :  output = (vowel==1) ? 1 : 3;
                         break;
        case  LIQUID  :  output = (vowel==1) ? 1 : 2;
                         break;
        case  CLOSURE :  output = (vowel==1) ? 1 : 6;
                         break;
        default       :  output = (vowel==1) ? 1 : 8;      /* silence */
                         break;
                      }

   return output;
```



```
}


/************************************************** Initialization **/
/*   Set the main variable :                                          */
/*      This creates a Synthesis variable suited for the input utterance, */
/*      and frees sufficient memory for all of its fields. It also reads  */
/*      some useful information from the utterance. No more memory allo-  */
/*      cation is needed afterwards, except for the waves and marks.      */
/*************************************************************************/


Synthesis *xps_INIT_ALL(Utterance utt)

{
   int units=0, npm, i=0;
   struct Unit *punit;
   P_Marks pmark;
   Synthesis *S;
   Stream u;


   S = xalloc(1, Synthesis);
   S->srate = udb_current->wave_sample_rate;


   for (u = UNITSTREAM(utt); u!=SNIL; u = SC_next(u))

     units++;                                   /* count number of units */


   S->nunits = units;
   S->w1 = xalloc(units, short *);
   S->w2 = xalloc(units, short *);
   S->ref = xalloc(units, Marks *);
   S->tar = xalloc(units, Marks *);
   S->map = xalloc(units, Map *);
   S->nrfpm = xalloc(units, int);
   S->ntgpm = xalloc(units, int);
   S->nsamp = xalloc(units, int);
   S->nsubu = xalloc(units, int);



   return S;

}



/************************************************ Reference Pitch Marks **/
/*   Read the reference marks :                                       */
/*      This reads pitch marks from 'punit' if they are preloaded, or from */
/*      the pitch mark files if they aren't (much slower), for a unit of a */
/*      given 'rank' in 'S'. 'npm' has the number of marks for the unit.   */
/*      Not all the fields are initialized.                              */
/*************************************************************************/


Marks *xps_Get_Pitchmarks(struct Unit *punit, int *npm, int srate)

{
    int start, end, i, k, scale, new, auxnpm;
    unsigned char trash;
    FILE *fid;
```

```
Small_pm *spm;
Marks *pm;
char line[500];
float shift1, shift2, begin, aux;

if ((punit->whole_pm != NULL))              /* pitch marks preloaded ! */
{
    spm = punit->whole_pm;
}
else                                        /* read appropriate files */
{
    if ((fid = fopen(punit->pitch_mark_file, "r")) == NULL)
    {
        P_Error("Failed to get pitchmarks from %s",punit->pitch_mark_file);
        list_error(On_Error_Tag);
    }

    spm = make_small_pm(num_lines(fid)+1);              /* allocation */
    spm->npm -= 1;          /* +1 in case we need to add one at zero */
    fscanf(fid, "%f %c", spm->pos, &trash);
    if (spm->pos[0] == 0)
        for (i=1; i<spm->npm; i++)                      /* reading */
            fscanf(fid, "%f %c", spm->pos+i, spm->voice+i);
    else
    {                                       /* make sure it begins at 0 */
        spm->pos[1] = spm->pos[0];
        spm->pos[0] = 0;
        spm->voice[0] = 0;                          /* most common case */
        for (i=1; i<spm->npm; i++)                      /* reading */
            fscanf(fid, "%f %c", spm->pos+1+i, spm->voice+i);
        spm->npm += 1;
    }
    fclose(fid);
}


i = 0;                                              /* all in milisec */
while ((i<spm->npm-1)&&(spm->pos[i] < punit->start)) i++;
start = (end = i);
while ((i<spm->npm)&&(spm->pos[i] <= punit->start + punit->length)) i++;
end = i - 1;
                                                    /* 'inner' marks */
if (end - start < 1)
{                                       /* not enough marks found */
    pm = xalloc(2+punit->num_sub_units, Marks);
    pm[0].nsamp = 0;
    pm[1].nsamp = punit->length *srate/1000;
    pm[0].voice = (pm[1].voice = 0);
    if (punit->whole_pm == NULL)
        free_small_pm(spm);                 /* 2 fake marks created */
    *npm = 2;
    return pm;
}

shift1  =  spm->pos[start] - punit->start;              /* positive */
auxnpm = end - start + 1;
pm = xalloc(auxnpm+punit->num_sub_units+2, Marks);      /* precaution */

for (i=1, aux=spm->pos[start], k=0 ; i-k<auxnpm; i++)
{
    new = (int) (spm->pos[start+i]-aux) * srate / 1000;
    if (i == 1)
    {
       pm[i].nsamp = new;
       pm[i].voice = 0;
    }
```



```
        if ((i>1)&&(new>pm[i-1-k].nsamp))
         [                                  /* check for corrupted marks */
           pm[i-k].nsamp = new;
           pm[i-k].voice = 0;
         ]
        if ((i>1)&&(new <= pm[i-1-k].nsamp))   /* k ==  number of bad marks */
         [
           auxnpm = auxnpm -1;
           k++;
         ]
    }

    pm[0].nsamp = (int) (shift1 * srate / 1000);            /* useful... */
    pm[0].voice = 0;
    *npm = auxnpm;

    if ((punit->whole_pm == NULL))
      free_small_pm(spm);

    return pm;

}


/********************************************** Reference Pitch Marks **/
/*   Read the reference marks :                                       */
/*      This calls function xps_Get_Pitchmarks, and then finishes the job, */
/*      for a unit of 'rank' in 'S'. All the fields are initialised.  */
/*************************************************************************/


void xps_READ_REFERENCE(Synthesis *S, int rank, Stream u, Scheme sch)

{
   int nsubu, i, j, k, size, start, type, npm, rk, rk2;
   int boundary, shift, aux, rate, temporary;
   struct Unit *punit;
   struct Sub_Unit *psubu;
   Marks *pmark, *pm;

   punit = SC(u, Unit);
   pmark = xps_Get_Pitchmarks(punit, &npm, S->srate);
   nsubu = punit->num_sub_units;
   rate = S->srate / 1000;

   temporary = pmark[0].nsamp;                      /* used in READ_WAVE */
   pmark[0].nsamp = 0;

   if (udb_current->pm_type==PM_VOICED_ONLY)   /* these need some pre-proc. */
    {
      for (i=0; i<npm; i++)
      {
         pmark[i].boundary = 0;
         pmark[i].voice = (npm > 2) ? 1 : 0;          /* most often true */
      }
      for (i=0; i<nsubu+1; i++)                     /* additional marks ? */
       {
         aux = xps_Find_Boundary(punit, i, pmark, npm);
         if (aux < 0)                                          /* one needed */
          {
            boundary = (i==nsubu) ? punit->length : punit->sub_units[i].start;
            boundary = boundary * rate;
            aux = -aux -1;                          /* rank of new mark */
            xps_Push_Marks(pmark+aux, npm - aux);
            pmark[aux].nsamp = boundary;
            pmark[aux].voice = 0;
```

```
          pmark[aux].boundary = 1;
          npm++;
        }
       else
         pmark[aux].boundary = 1;
    }
}


S->nrfpm[rank] = npm;
S->ntgpm[rank] = npm;
S->ref[rank] = pmark;
S->tar[rank] = xalloc(2*npm, Marks);                /* just in case ! */
S->map[rank] = xalloc(npm, Map);
S->nsubu[rank] = nsubu;


for (i=(k=0); i<npm; i++)
  {
    S->ref[rank][i].nsamp = pmark[i].nsamp;             /* precaution */
    S->ref[rank][i].rank = i;
    S->ref[rank][i].forbid = 0;
    S->ref[rank][i].boundary = (udb_current->pm_type==PM_VOICED_ONLY) ?
                               pmark[i].boundary : 0;
    S->ref[rank][i].voice = pmark[i].voice;
    S->ref[rank][i].prev = (i == 0) ? NULL : S->ref[rank]+i-1;
    if (i >= S->nrfpm[rank]-1)
      S->ref[rank][S->nrfpm[rank]-1].next =  NULL;
    else
      S->ref[rank][i].next = S->ref[rank]+i+1;

  }

S->nsamp[rank] = S->ref[rank][S->nrfpm[rank]-1].nsamp + 1;

S->ref[rank][0].boundary = 1;
S->ref[rank][S->nrfpm[rank]-1].boundary = 1;
psubu = punit->sub_units;                               /* initialize */
type = xps_PHONEM(psubu);
for (k=0; k<S->nrfpm[rank]; k++)
     S->ref[rank][k].phoneme = type;

if (nsubu > 1)
{
  rk2 = 1;                                  /* two routines for two types */
  rk = (aux = 0);                            /* of pitch marks, different */
  size = punit->sub_units[0].length * rate;
  while (S->ref[rank][rk2].boundary == 0) rk2++;
  while (S->ref[rank][rk].nsamp+20 <= size) rk++;
  for (i=1; i<nsubu; i++)                                /* locate boundaries */
   {                                                  /* & read phoneme types */
     psubu = punit->sub_units + i;
     size = MAX(0,psubu->length) * rate;                        /* in samples */
     start = MAX(0,psubu->start) * rate;
     type = xps_PHONEM(psubu);

     if (udb_current->pm_type==PM_VOICED_ONLY)

       for (j=rk2; ((j<S->nrfpm[rank])&&(aux<=i)); j++)
       {
          S->ref[rank][j].phoneme = type;
          aux +=  S->ref[rank][j].boundary;
       }
     else
     {
       for (j=rk; ((j<S->nrfpm[rank]) &&
```



```
                ((S->ref[rank][j].nsamp+20) <= (start + size))); j++)

          S->ref[rank][j].phoneme = type;

        S->ref[rank][rk].boundary += 1;                        /* new border */
        if ((j-1 <= rk)||(j==S->nrfpm[rank]))
          S->nsubu[rank] -= 1;                      /* too small a sub-unit */
      }
        rk = MAX(rk, j-1);
        aux--;
        rk2 = j-1;
    }

  }

 if (udb_current->pm_type==PM_ALL_MARKED)                      /* voicing ? */
 for (i=0; i<npm; i++)
     S->ref[rank][i].voice = (sch.voicing == 0) ?       /* based on phoneme */
     1-NOISY(S->ref[rank][i].phoneme) : pmark[i].voice;         /* or on db */

 aux = S->nsamp[rank] ;

  for (i=0, size=0; i<nsubu; i++)
    size += MAX(0, punit->sub_units[i].targ_length);

  size = MAX(aux, size * rate);
  S->w1[rank] = xalloc(2*size, short);
  S->w2[rank] = xalloc(2*size, short);             /* extra room provided */

  S->w1[rank][0] = shift;                  /* to account for lost samples */

  S->ref[rank][0].nsamp = temporary;                     /* see beginning */

}



/************************************************************** Mapping **/
/*   Create the mapping :                                                */
/*       This creates an f0 estimation for the whole utterance, and then   */
/*       deducts the prosodic modifications to be performed on every unit, */
/*       by filling the 'map' field in 'S'. Three possible different ways. */
/************************************************************************/


void xps_MAPPING(Synthesis *S, Scheme sch, Utterance utt)

{
   int i, j, k, l, total, start, end, nsubu, rnpm, rlength, back, shift, *F0;
   int size, maxsize, tarsize,  npm, length, last, min, max;
   int start2, end2, limit, je, js, flag, vlength;
   float dmodif, pmodif, *periods, rep, scale, f0ref, f0tar, aux;
   float pmodif1, pmodif2, dmodif1, dmodif2, voiced;
   Marks *current, *next;
   struct Unit *punit, *temp;
   struct Sub_Unit *psubu;
   Stream u;


   tarsize = (flag = ( start = ( end = (back = 0))));
   F0 = make_F0(utt, &total, F0_FRAM, FALSE);                /* target f0 */
   maxsize = total * F0_FRAM;                          /* target msec length */
   periods = xalloc(1000, float);                        /* 1000 periods max */

   for (u=UNITSTREAM(utt); u!=SNIL; u=SC_next(u))
     {
```

```
      punit = SC(u, Unit);
      for (j=0; j<punit->num_sub_units; j++)
         tarsize += MAX(0, punit->sub_units[j].targ_length);
   }

scale = ((float) maxsize)/tarsize;          /* make sure that sizes fit */


for (u=UNITSTREAM(utt),i=(back=(start=0)); u!=SNIL; u=SC_next(u))
 {
   punit = SC(u, Unit);
   current = S->ref[i];
   js = 0;

   for (j=0; j<punit->num_sub_units; j++)
    {
      je = js;
      dmodif2 = 0;
      while (current[++je].boundary == 0);
      nsubu = (j==0) ? S->ref[i][0].boundary - 1 : 0;
      nsubu += current[je].boundary;
                                                  /* prosody input */

      length = (int) (scale*MAX(0, punit->sub_units[j].targ_length));
      rlength = MAX(0, punit->sub_units[j].length);
      end = MIN(start + (int)(scale*length+0.5), total * F0_FRAM);
      xps_PERIODS(F0, total, start, end, &npm, periods);
      rnpm = je - js + 1;
                              /* we try to account for unvoiced speech */
      if (udb_current->pm_type == PM_VOICED_ONLY)
      {
         vlength = (k = 0);
         while ((k<rnpm-1)&&(current[js+k].voice == 0)) k++;
         start2 = (current[js+k].nsamp-current[js].nsamp)*S->srate/1000;
         start2 = start2 + start;
         rnpm -= k;
         k = 0;
         while ((k<rnpm-1)&&(current[je-k].voice == 0)) k++;
         end2 = (current[je].nsamp-current[je-k].nsamp)*S->srate/1000;
         end2 = end - end2;
         rnpm = MAX(2, rnpm-k);
         xps_PERIODS(F0, total, start2, end2, &npm, periods);
         npm = MAX(2, npm);
      }

      pmodif1 = ((float)rlength*npm)/(length*rnpm);       /* 1st method */
      dmodif1 = ((float) npm) / rnpm;

      if ((rnpm < 3)||(npm < 3))            /* no use having pitch modifs */
      {
        pmodif1 = 1;
        dmodif1 = length / (float)rlength;
      }

      for (k=js; k<je; k++)
      {
         max = MIN(k+1, S->nrfpm[i]-1);
         min = MAX(0, k-1);
                                          /* ref and target f0 values */
         if (xps_Is_Alone(current + k, SPAN))
         {
           f0ref = S->ref[i][max].nsamp - S->ref[i][min].nsamp;
           f0ref = (max-min) * S->srate / f0ref;
           l = back + 1000*scale*length*(S->ref[i][k].nsamp -
                          S->ref[i][js].nsamp)/(S->srate*rlength);
           l = MIN(total-1, l/F0_FRAM);
```



```
           f0tar = ( F0[l] + F0[MIN(l+1, total-1)] ) / 2;

           pmodif2 = f0tar / f0ref;                    /* 2nd method */
           dmodif2 = pmodif2 * length / rlength;
         }
         else
         {
          pmodif2 = 1;                    /* no pitch modif if unvoiced */
          dmodif2 = length / rlength;
         }

         switch (sch.contour)
         {
           case 1 :  pmodif = pmodif1;
                     dmodif = dmodif1;
                     break;
           case 2 :  pmodif = pmodif2;
                     dmodif = dmodif2;
                     break;
           default : pmodif = (pmodif1 + pmodif2) / 2;
                     dmodif = (dmodif1 + dmodif2) / 2;
                     break;
         }

         pmodif=MAX(1-sch.P_ceil,MIN(1+sch.P_ceil,pmodif));     /* limit */
         dmodif=MAX(1-sch.D_ceil,MIN(1+sch.D_ceil,dmodif));     /* limit */


         if (S->ref[i][k].phoneme == 7)            /* special for stops */
         {
            pmodif = (sch.stops == 1) ? pmodif : 1;
            dmodif = 1;
            S->ref[i][k].forbid = 1;
         }

         back += (0.5 + 1000*length*scale*(S->ref[i][je].nsamp -
                     S->ref[i][js].nsamp)) / (S->srate * rlength);

         S->map[i][k].rank = k;
         S->map[i][k].P_modif = pmodif;
         S->map[i][k].D_modif = dmodif;
         S->map[i][k].R_modif = 0;
      }


     if ((current[je].next == NULL ) || (je >= S->nrfpm[i]-1))
      {
        js = 0;                                          /* next unit */
        S->map[i][S->nrfpm[i]-1].rank = S->nrfpm[i]-1;
        S->map[i][S->nrfpm[i]-1].P_modif =  S->map[i][S->nrfpm[i]-2].P_modif;
        S->map[i][S->nrfpm[i]-1].D_modif = S->map[i][S->nrfpm[i]-2].D_modif;
        S->map[i][S->nrfpm[i]-1].R_modif = 0;
        i++;
      }

      else   js = je;                                    /* next sub-unit */

   }

}


                                                /* final lap : field 'repeat' */
for (i=0; i<S->nunits; i++)
{
  if (S->nrfpm[i] >1)
```

```
    {
      if (sch.contour == 1)
         rep = (S->map[i][0].D_modif > 1) ? 0.5 : -0.5;
      else if (sch.contour == 3)
            rep = (S->map[i][0].D_modif > 1) ? 0.25 : -0.25;
         else rep = 0;

      for (j=0, flag=0; j<S->nrfpm[i]; j++)
      {
        limit = ((j<2) || (j>(S->nrfpm[i]-3))||(S->ref[i][j].forbid == 1));
        aux = ( Abs(1-S->map[i][j].D_modif) > sch.D_thres ) ?
                                              S->map[i][j].D_modif : 1;
        rep += (aux - 1);
        S->map[i][j].D_modif = aux;
        S->map[i][j].P_modif = ( Abs(1-S->map[i][j].P_modif) > sch.P_thres ) ?
                                              S->map[i][j].P_modif : 1;

        if (flag==1)
        {
          flag = 0;
          S->map[i][j].repeat = 0;      /* never two consecutive dupl/elim */
         }
        else if (rep>1)                                     /* duplication */
            {
              rep -= 1;
              S->map[i][j].repeat = (limit == 1) ? 0 : 1;
              flag = 1;
             }
            else if (rep<-1)                                /* elimination */
                 {
                   rep += 1;
                   S->map[i][j].repeat =
                      ((limit == 1)||(S->ref[i][j].boundary != 0)) ? 0 : -1;
                   flag = 1;
                 }
                else   S->map[i][j].repeat = 0;
      }
    }
   }

   xfree(F0);
   xfree(periods);

}



/************************************************** Create Target Periods **/
/*   Make target periods :                                                 */
/*     Given the set of f0 estimates every F0_FRAM mseconds, and the start */
/*     and end of the current interval (a sub-unit), this calculates and   */
/*     returns a set of values for the 'n' target pitch periods.           */
/*************************************************************************/


void xps_PERIODS(int *f0,int total,int start,int end,int *n,float *periods)

{
   int f0start, f0end, npm=0, nf0=0, i;
   float previous=0, current;


   f0start = start / F0_FRAM;                       /* limits on the f0 list */
   f0end = MIN(end / F0_FRAM + 1, total);

   for (i=f0start; i<f0end; i++)
```



```
    {
      if (f0[i]<=0)  f0[i] = 100;                                  /* 100 Hz */

      current = (1000/(float)f0[i] + nf0*previous) / (nf0+1);   /* in msec */

      if (((nf0+1)*F0_FRAM) > current)
        {
          periods[npm] = current;                              /* new mark */

          if (npm<499) npm++;                                 /* precaution */
          nf0 = 0;
          current = 0;                                            /* reset */
          previous = 0;
        }

      else                                              /* one more is needed */
        {
          previous = current;
          nf0++;
        }
    }

   if (npm == 0) *n = 0;
   else
     *n = (((nf0+1)*F0_FRAM) > 2*current) ? npm+2 : npm+1;

}


/************************************************** Read Wave File **/
/*   Read the waveform :                                            */
/*      This reads into the waveform files, for a unit of a given 'rank'   */
/*      in 'S'. The field 'nsamp' in 'S' is initialised.            */
/*********************************************************************/


void xps_READ_WAVE(Synthesis *S, int rank, Stream u)
{
    P_Wave wave, output;
    int i, aux, shift, size;
    char *encoding;
    struct Unit *unit;

    if (S->nrfpm[rank] > 1)
    {
    encoding = udb_current->wave_encoding;
    unit=SC(u,Unit);

    output = make_wave();

    if ((unit->filetype != NULL)&&(strcmp(unit->filetype,"raw")==0))
        wave=get_raw_sub_wave
            (unit->wave_file, S->srate, encoding, unit->start, unit->length);

    else wave=get_sub_wave
        (unit->wave_file, unit->filetype, unit->start, unit->length);


    if (wave==NULL)
    {
        P_Error("Read_Wave : unable to read waveform");
        list_error(On_Error_Tag);
    }

    if ((S->srate!=wave->samp_rate)||(output->byte_order!=wave->byte_order))
    {
```

```
        P_Error("Read_Wave : units of incompatible types");
        P_Error("srates : %d %d ", output->samp_rate,wave->samp_rate);
        P_Error("word sizes : %d %d ", output->word_size, wave->word_size);
        P_Error("byte orders : %d %d", output->byte_order,wave->byte_order);
        list_error(On_Error_Tag);
    }

    aux = S->nsamp[rank] ;

    for (i=0, size=0; i<unit->num_sub_units; i++)
        size += MAX(0, unit->sub_units[i].targ_length);

    size = MAX(aux, size * S->srate / 1000);

    bzero( S->wl[rank]+aux, (2*size-aux)*sizeof(short));
    size = MIN(S->ref[rank][0].nsamp, wave->num_samp-aux);
    if (size<0)
    {
      S->nsamp[rank] += size;
      aux = S->nsamp[rank];
      S->ref[rank][S->nrfpm[rank]-1].nsamp += size;
      size = 0;
    }
    xps_strcopy(S->wl[rank], wave->wave+size, aux, 1);

    S->ref[rank][0].nsamp = 0;

    free_pwave(wave);
   }



}


/******************************************************* Power Processing **/
/*  Power Smoothening :                                                    */
/*     This is performed before any other processing on the waves. We try  */
/*     here to erase power discontinuties between units by measuring &     */
/*     modifying their rms. This happens before prosodic processing.       */
/*     Depending on the scheme, different approaches can be taken.          */
/*************************************************************************/


void xps_POWER_PRO(Synthesis *S, Utterance utt, Scheme sch)

{
   int i, j, k, nunits, nsubu, nsub, size, index, previ, cur, next, toto,Start;
   float rfpow, tgpow, aux, scale1[3], *window;
   float *scale, a, b[4], prev=1;
   double log10, daux;
   Marks *start, *end, *first, *pm1, *pm2;
   struct Unit *pu;
   struct Sub_Unit *psu;
   Stream u;


   u = UNITSTREAM(utt);
   nunits = S->nunits;
   window = xalloc(5*S->srate, float);                    /* 5 seconds max */
   scale = xalloc(BUF_SIZ, float);

   for (i=0; i<nunits; i++)
   {
     pu = SC(u, Unit);
```



```
     psu = pu->sub_units;
     nsubu = S->nsubu[i];

   switch (sch.power)
     {

       case 1 :  break;                                  /* nothing is done */


       case 2 :  for (j=0; j<nsubu; j++)                 /* scaling factor */
                   {
                     rfpow = pu->sub_units[j].power;   /* log(rms), maybe ? */
                     tgpow = pu->sub_units[j].targ_power;
                     scale[j] = (tgpow == 0) ? 0 : tgpow - rfpow;
                     scale[j] = sqrt(exp(scale[j]));
                   }

                 k = -1;
                 start = S->ref[i];

                 while ((start != NULL)&&(start->next != NULL))
                  {
                       k += start->boundary;
                       end = start + 1;
                       while (end->boundary == 0) end = end->next;

                       while (start->rank < end->rank)
                        {
                         S->map[i][start->rank].R_modif = scale[k];
                         for (j=start->nsamp; j<start->next->nsamp; j++)

                           S->wl[i][j] = S->wl[i][j] * scale[k];  /* scaling */

                         start = start->next;
                        }

                   if ((start == NULL)||(start->prev == NULL))
                    {
                      P_Error("POWER_PRO : wrong number of sub-units ");
                      break;
                    }

                  }

                 break;


        default : if ((nunits > 1)&&(S->nrfpm[i] > 1))
                   {
/*
**   we want to raise the loudness level halfway to the mean value.
**   the needed scaling factor is stored in a[].
**   near the border, we want to get closer to the next/previous
**   sub-unit's loudness level. b[] has the scaling for that.
**   power in stops is not modified with normal marks.
*/

                     daux = exp(xps_LOG_POW(S, i));                    /* left */
                     aux = exp(xps_MEAN_POWER(utt, S, i));
                     a = (float)((daux + aux) / (2 * daux));
                     b[0] = (b[3] = 0);

                     /****************************************** normal marks */

                     if (udb_current->pm_type == PM_ALL_MARKED) {
                     if ((i != 0)&&(S->nrfpm[i-1] > 1))
```

```
 {
   previ = i-1;
   first = S->ref[previ];
   index = xps_INDEX(first, 1);
   start = first + index;
   index = S->nrfpm[previ] - index -1;
   b[0] = xps_RMS(S->wl[previ], start, index);
 }

first = S->ref[i];
index = xps_INDEX(first, -1);
b[1] = xps_RMS(S->wl[i], first, index);

index = xps_INDEX(first, 1);
start = first + index;
index = S->nrfpm[i] - index -1;
b[2] = xps_RMS(S->wl[i], start, index);

if ((i<nunits-1)&&(S->nrfpm[i+1] > 1))
 {
  next = i+1;
  first = S->ref[next];
  index = xps_INDEX(first, -1);
  b[3] = xps_RMS(S->wl[next], first, index);
 }}

 /****************************************** larynx marks */

 else {
if ((i != 0)&&(S->nrfpm[i-1] > 1))
 {
   Start = xps_INDEX_bis(S->ref[i-1], 1);
   index = S->nsamp[i-1] - Start;
   b[0] = xps_RMS_bis(S->wl[i-1], Start, index);
 }


first = S->ref[i];
Start = xps_INDEX_bis(first, -1);
b[1] = xps_RMS_bis(S->wl[i], 0, Start);

Start = xps_INDEX_bis(first, 1);
index = S->nsamp[i] - Start;
b[2] = xps_RMS_bis(S->wl[i], Start, index);

if ((i<nunits-1)&&(S->nrfpm[i+1] > 1))
 {
  next = i+ 1;
  Start = xps_INDEX_bis(S->ref[i+1], -1);
  b[3] = xps_RMS_bis(S->wl[next], 0, Start);
 }}

/*******************************************************/

                                        /* scaling factors */

scale[0] = ((i == 0)||(b[0]<=0)||(b[1]<=0)) ?
                                1 : prev/(2*prev-1);
scale[1] = sqrt(a);
scale[2] = ((i==(nunits-1))||(b[2]<=0)||(b[3]<=0)) ?
                              1 : (1 +sqrt(b[3]/b[2]))/2;

scale[0] = MAX(0.66, MIN(1.5, scale[0]));   /* precaution */
scale[1] = MAX(0.8, MIN(1.3, scale[1]));
scale[2] = MAX(0.66, MIN(1.5, scale[2]));
```



```
                if (udb_current->pm_type == PM_ALL_MARKED)
                {
                 first = S->ref[i];
                 pm1 = S->ref[i]+xps_INDEX(first, -1);
                 pm2 = S->ref[i] + xps_INDEX(first, 1);
                }
                else
                {
                 first = S->ref[i];
                 pm1 = xalloc(2, Marks);                    /* modification */
                 pm2 = pm1 + 1;                               /* boundaries */
                 pm1->nsamp = xps_INDEX_bis(first, -1);
                 pm2->nsamp = xps_INDEX_bis(first, 1);
                 pm1->prev = first;
                 pm2->next = first + S->nrfpm[i]-1;
                }

                 xps_WIN_POW(&scale[0], pm1, pm2, window);/* scaling window */
                 size = S->nsamp[i];

                if (udb_current->pm_type == PM_ALL_MARKED) {
                 for (j=pm1->rank; j<=pm2->rank; j++)
                  {
                    toto = S->ref[i][j].nsamp - pm1->nsamp;
                    S->map[i][j].R_modif = window[toto];
                  }}

                 for (j=0; j<size; j++)
                    S->wl[i][j] = S->wl[i][j]*window[j];

                prev = scale[2];
                }
               break;

     }

   u = SC_next(u);


  }

   xfree(window);

}




/************************************************** Output Speech Wave ***/
/*   Final waveform :                                                    */
/*      After the prosodic modifications have been performed in 'S', this  */
/*      builds the output wave by concatenating the units and building a   */
/*      Wave * variable.                                                   */
/************************************************************************/


P_Wave xps_FINAL_WAVE(Synthesis *S, Scheme sch)

{
    int i, j, size=0, end, additional, units, marks;
    int index, new_index, shift, total;
    short *curr, *p;
    P_Wave waveform;
    Marks *tgpm;
```

```
    float k, step;

    units = S->nunits;
    for (i=0; i<units; i++)

        size += S->nsamp[i];

    waveform = make_wave();     /* building output wave */
    waveform->wave = xalloc(size+2*S->nsamp[units-1], short); /* spare room */
    waveform->num_samp = size;
    waveform->samp_rate = S->srate;

    curr = waveform->wave;
    index = 0;

    if (units == 1)

      for (i=0; i<S->nsamp[0]; i++)   waveform->wave[i] = S->wl[0][i];

    else
    for (i=0; i<(units-1); i++) /* unit concatenation */
    {
        new_index = xps_CONCAT(S, i, waveform, index, sch.concat);
        index = new_index-1;                /* the "-1" removes some UMRs */
    }                                                   /* gomen ne ? */

    marks = S->ntgpm[units-1];  /* final fading */

    if ((marks != 0)&&(S->tar[units-1][marks-1].phoneme != 8))
    {
        additional = MIN(3, marks);
        tgpm = S->tar[units-1];
        j = S->tar[units-1][marks-additional].nsamp;
        shift = (tgpm+marks -1)->nsamp - (tgpm + marks - additional)->nsamp;
        shift = MIN(shift, S->nsamp[units-1]/3);
        step = 0.5 / shift;
        k = 1;                  /* fading lasts to the end */

        p = waveform->wave + index - shift;


        for (i=0; i<shift; i++)
        {
            p[i] = (short)( p[i] * k);
            k = k-step;
        }

        p = waveform->wave + index;

        for (i=0; i<shift; i++)
        {
            p[i] = (short)( p[i-shift] * k / (k + 0.5));
            k = MAX(0, k-step);
        }

     }  else shift = 0;

    waveform->num_samp += shift;

    for (i=new_index + shift; i<new_index + 2*shift; i++)

        waveform->wave[i] = 0;

    waveform->num_samp += shift;
```



```
    for (i=0; i<S->nunits; i++) /* memory freeing */
    {
        xfree(S->wl[i]);
        xfree(S->w2[i]);
        xfree(S->ref[i]);
        xfree(S->tar[i]);
        xfree(S->map[i]);
    }

    xfree(S->wl);
    xfree(S->w2);
    xfree(S->ref);
    xfree(S->tar);
    xfree(S->map);
    xfree(S->nsamp);
    xfree(S->nrfpm);
    xfree(S->ntgpm);
    xfree(S->nsubu);


    return (waveform);

}


/*********************************************************** Thingie Number 1 */

void free_small_pm(Small_pm *pm)
{

    if (pm != NULL)
    {
        xfree(pm->voice);
        xfree(pm->pos);
        xfree(pm);
    }
}

/*********************************************************** Thingie Number 2 */

Small_pm *make_small_pm(int npm)
{
    Small_pm *pm;

    pm = xalloc(1,Small_pm);
    pm->voice = xalloc(npm,unsigned char);
    pm->pos = xalloc(npm,float);
    pm->npm = npm;

    return pm;
}
```

```
/*=========================================================================*/
/*            A T R Interpreting Telecommunications Labs                   */
/*                                                                         */
/*-------------------------------------------------------------------------*/
/*                 CHATR Speech Synthesis System                           */
/*                       Christian Lelong                                  */
/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                    Mathematical Library                                 */
/*           Fourier Transforms, Cepstrum Analysis, etc                    */
/*                                                                         */
/*                                                                         */
/*                         Feb 1995                                        */
/*                    Copyrigth (C) 1994, 1995                             */
/*     ATR Interpreting Telecommunications Research Laboratories           */
/*                       All rights reserved.                              */
/*=========================================================================*/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "xruc.h"

#define C1 1
#define C2 1
#define C3 1
#define C_MIN 2
#define C_MAX 18
#define ICEP_NUM 3
#define CEP_ORD 32

float icep_coef[ICEP_NUM];

/* Spectral Analysis */

static void xps_four1(float *data, int nn, int isign);


/*************************************************************** Power of 2 **/
/*   returns the smallest power of 2 greater or equal than input 'a'.      */
/*****************************************************************************/


int xps_ispower2(int a)

{
   int b=1, c;

   c = a;
   while ((a = a >> 1) != 0) b = b << 1;
   if (c>b) b = b << 1;

   return b;
}


/*********************************************************** Sygmoid Function **/
/*   Sigmoid aproximation :                                                  */
/*       f(x)=0 for x<0, =1 for x>1, degree-4 polynomial used for [0,1]      */
/*****************************************************************************/


float xps_Sygmoid(float x)

{
   float y;
```



```
   if (x<0) y=0;
   else if (x<0.5)  y=8*x*x*x*x;
        else if (x<1)  y=1-8*(1-x)*(1-x)*(1-x)*(1-x);
             else y=1;

   return y;

}



/************************************* Hanning / Hamming / Hybrid  Window **/
/*   assymetric windows :                                                  */
/*      'dim' samples long, maximum value 'scale' at sample 'middle', and  */
/*      Hanning/Hamming mode is selected by 'type' = 1 / 2.                */
/*      'type' 3 returns a rectangular window featuring smoothened edges   */
/*     (X^2-like), and 'middle' is the length of the flat part.            */
/*****************************************************************************/


float *xps_Window(int dim, int scale, int middle, int type)

{
    int l_half, r_half, edge, i, tempr;
    float *win, left_pi, rigth_pi, a, b;


    win = xalloc(dim,float);


    switch (type)    {

      case  3 : edge=(dim-middle)/2; /* hybrid window */
        for (i=0; i<dim; i++)  {

            if (i<edge/2)                   win[i]=scale*(i/edge)*(i/edge);
            else if (i<edge)                win[i]=scale*(1-i/edge)*(1-i/edge);
            else if (i<(edge+middle))       win[i]=scale;
            else if (i<(1.5*edge+middle))   win[i]=scale*(1-(i-edge-middle))*(1-(i-edg
e-middle));
            else                            win[i]=scale*(i-2*edge-middle)*(i-2*edge-m
iddle);
        }
        break;


      default : if (type==1) { a=0.5; b=0.5;} /* hanning or hamming */
      else { a=0.54; b=0.46;};
        l_half = middle;
        r_half = dim +1 - middle;
        left_pi = 8.0 * atan(1.0) / (2 * l_half);
        rigth_pi = 8.0 * atan(1.0)/ (2 * r_half);

        for (i=0; i < middle; i++)
            win[i] = (a + b*cos(left_pi * (float)(i - middle))) * scale ;
        for (i=(dim-1); i >= middle; i--)
            win[i] = (a + b*cos(rigth_pi * (float)(i - middle))) * scale ;
        win[middle-1] = scale;
        break;
    }

    return (win);

}
```

```
/*********************************************** Complex Type Conversion **/
/* convert complex numbers to polar representation :                      */
/*                  'list' of 'length' cartesian input complex numbers    */
/*************************************************************************/


Polar *xps_Cart2Pol(Complex *list, int length)

{
   int i;
   Polar *out;


   out = (Polar *)xalloc(2*length, float);
   for (i=0; ((i<length)&&(list!=NULL)); i++)
    {
      out[i].scale = sqrt (list->real * list->real + list->imag*list->imag);
      if (list->real == 0) out[i].phase = atan(1.0) * 2 * sign (list->imag);
      else out[i].phase = atan (list->imag / list->real);
      list++;
    }

   return (out);

}


/************************************************************ Get the RMS **/
/*  Read a period's power :                                               */
/*     This gets the normalized rms value for 'n' pitch periods from the  */
/*     input waveform beginning at 'first' ; processing starts at 'pm'.   */
/*************************************************************************/


float xps_RMS(short *wave, Marks *pm, int n)

{
    double rms=0, aux;
    Marks *end;
    int i=0;


    if (n == 0) return (-1);    /* warning */

    end = pm;
    while ((end->next != NULL)&&(i++<n)) end = end->next;

    if ((pm == NULL)||(pm->next == NULL))
    {
        P_Error("\n RMS : invalid pitch marks \n");
        list_error(On_Error_Tag);
    }
    else
    {
        for (i=pm->nsamp; i<end->nsamp; i++)
        {
            aux = (double)wave[i] ;
            rms += aux * aux;
        }
        rms =sqrt( rms / (end->nsamp - pm->nsamp + 1));
    }

    return ((float)rms);

}
```



```
/************************************************************ Get the RMS **/
/*  Read a period's power :                                               */
/*     This gets the normalized rms value for 'size' samples from the     */
/*     input waveform beginning at 'start'. For larynx pitch marks.       */
/*************************************************************************/


float xps_RMS_bis(short *wave, int start, int size)

{
    double rms=0, aux;
    int i;

    if (size < 0)
    {
       start = start + size;
       size = -size;
    }

    for (i=start; i<size; i++)
     {
       aux = (double)wave[i] ;
       rms += aux * aux;
     }
    rms =sqrt( rms / (size + 1));

    return ((float)rms);

}


/************************************************************ Read Power **/
/*  Read a sub-unit's power :                                             */
/*     This returns ln(E[s^2]), the average of the log of the square, for */
/*     the nth unit in 'S'.                                               */
/*************************************************************************/


float xps_LOG_POW(Synthesis *S, int n)

{
   int i, size;
   Marks *start, *end;
   float logpow, aux;


   start = S->ref[n];
   end = start + S->nrfpm[n] - 1;

   if (((start == NULL)||(start->next == NULL))&&(n<S->nunits))
     {
       P_Error("\n LOG_POW : rank of unit exceeds number of units ");
       return 0;
     }

   logpow = 0;
   size = end->nsamp - start->nsamp;

   for (i=start->nsamp+100; i<=end->nsamp-100; i++)/* to avoid interference */
    {                                              /* from neighbouring units */
      aux = S->wl[n][i] * S->wl[n][i] + 1.0;
      logpow += (double)aux;
    }

   logpow = MAX( 1, logpow / (size-200));
   logpow = 0.5 * log(logpow);
```

```
   return logpow;

}


/************************************************************ Power Window **/
/*  Tailor-made Window :                                                    */
/*    This returns a window allowing smooth power modifications in a given  */
/*    unit. 's' has the three layers of the window : the target value in    */
/*    between, close to the mean, and the target values at the edges close  */
/*    to the neighbouring sub-units's levels. Marks 'pm1' & 'pm2' are the   */
/*    left-side and rigth-side marks of the current sub-unit.               */
/*************************************************************************/


void xps_WIN_POW(float *s, Marks *pm1, Marks *pm2, float *w)

{
    int siz1, siz2, siz3, siz4, totsiz, i;
    Marks *beg, *end;


    if ((pm1==NULL)||(pm2==NULL))
    {
        P_Error("\n WIN_POW : unsuitable marks ");
        list_error(On_Error_Tag);
    }

    beg = pm1;
    while (beg->prev != NULL) beg = beg->prev;
    end = pm2;
    while (end->next != NULL)  end = end->next;

    siz1 = pm1->nsamp - beg->nsamp; /* window borders */
    siz3 = end->nsamp - pm2->nsamp;
    totsiz = end->nsamp - beg->nsamp + 1; /* window size */

    /*
     **   If the sub-unit is less than four pitch periods long, we go for
     **   a smooth modification in power, in order to meet the expected value
     **   at the edge. If the sub-unit is longer, we raise it closer to the
     **   mean power level of its class, and tune it at the edge as before.
     **   If the current unit, however is made of only one short (<4 periods)
     **   sub-unit, the power modification is a weigthed sum of the power
     **   level at its edges.
     */


    if ((siz1+siz3) >= totsiz)
        for (i=0; i<totsiz; i++)
            w[i] = (s[2]*i + s[0]*(totsiz-i)) / totsiz;
    else                                                /* long sub-unit */
    {
        for (i=(siz1); i<(totsiz-siz3); i++)
            w[i] = s[1];                                /* middle section */

        siz2 = siz1/2;
        siz1 = siz1 - siz2;
        siz4 = siz3/2;
        siz3 = siz3 - siz4;

        for (i=0; i<siz1; i++)
            w[i] = s[0]+(s[1]-s[0])*(1-cos((PI*i)/(2.0*siz1)))/2;

        for (i=siz1; i<siz1+siz2; i++)
```



```
            w[i] = s[0]+(s[1]-s[0])*(1-sin((PI*(i-siz1))/(2.0*siz2)))/2;

        for (i=(totsiz-siz3-siz4); i<(totsiz-siz4); i++)
            w[i] = s[1]+(s[2]-s[1])*(1-cos((PI*(i-totsiz+siz3+siz4))/(2.0*siz3)))/2;

        for (i=(totsiz-siz4); i<totsiz; i++)
            w[i] = s[1]+(s[2]-s[1])*(1-sin((PI*(i-totsiz+siz4))/(2.0*siz4)))/2;

    }

}


/************************************************** FFT auxiliar function **/
/* Numerical Recipes :                                                     */
/*    'data' => input/output of size 'nn', 'isign' = 1/-1 for fft/ifft     */
/*************************************************************************/


static void xps_four1(float *data, int nn, int isign)

{
   int n,mmax,m,j,istep,i;
   double wtemp,wr,wpr,wpi,wi,theta;
   float tempr,tempi;

   n=nn << 1;
   j=1;

   for (i=1;i<n;i+=2)
     {
        if (j > i)
          {
             swap(data[j],data[i], tempr);
             swap(data[j+1],data[i+1], tempr);
          }
        m=n >> 1;
        while (m >= 2 && j > m)
           {
              j -= m;
              m >>= 1;
           }
        j += m;
     }

   mmax=2;
   while (n > mmax)
     {
        istep=2*mmax;
        theta=6.28318530717959/(isign*mmax);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2)
          {
             for (i=m;i<=n;i+=istep)
               {
                    j=i+mmax;
                    tempr=wr*data[j]-wi*data[j+1];
                    tempi=wr*data[j+1]+wi*data[j];
                    data[j]=data[i]-tempr;
                    data[j+1]=data[i+1]-tempi;
                    data[i] += tempr;
                    data[i+1] += tempi;
```

```
                }
                wr=(wtemp=wr)*wpr-wi*wpi+wr;
                wi=wi*wpr+wtemp*wpi+wi;
            }
        mmax=istep;
    }
}


/************************************************** FFT main function **/
/* Numerical Recipes :                                                 */
/* 'data' => input/output of size 'n', 'isign' = 1/-1 for fft/ifft     */
/***********************************************************************/


void xps_realft(float *data,int n, int isign)

{
   int i,i1,i2,i3,i4,n2p3;
   float c1=0.5,c2,h1r,h1i,h2r,h2i;
   double wr,wi,wpr,wpi,wtemp,theta;

   theta=3.141592653589793/(double) n;
   if (isign == 1)
     {
        c2 = -0.5;
        xps_four1(data,n,1);
     }
   else
      {
         c2=0.5;
         theta = -theta;
       }
   wtemp=sin(0.5*theta);
   wpr = -2.0*wtemp*wtemp;
   wpi=sin(theta);
   wr=1.0+wpr;
   wi=wpi;
   n2p3=2*n+3;
   for (i=2;i<=n/2;i++)
     {
        i4=1+(i3=n2p3-(i2=1+(i1=i+i-1)));
         h1r=c1*(data[i1]+data[i3]);
         h1i=c1*(data[i2]-data[i4]);
         h2r = -c2*(data[i2]+data[i4]);
         h2i=c2*(data[i1]-data[i3]);
         data[i1]=h1r+wr*h2r-wi*h2i;
         data[i2]=h1i+wr*h2i+wi*h2r;
         data[i3]=h1r-wr*h2r+wi*h2i;
         data[i4] = -h1i+wr*h2i+wi*h2r;
         wr=(wtemp=wr)*wpr-wi*wpi+wr;
         wi=wi*wpr+wtemp*wpi+wi;
      }
   if (isign == 1)
     {
        data[1] = (h1r=data[1])+data[2];
        data[2] = h1r-data[2];
      }
   else
      {
         data[1]=c1*((h1r=data[1])+data[2]);
         data[2]=c1*(h1r-data[2]);
         xps_four1(data,n,-1);
       }
}
```



```
/************************************************ Auto Correlation Function **/
/*  Numerical Recipes :                                                      */
/*                        'V''=> input/ouput vectors of size 'm'             */
/*****************************************************************************/


void xps_acf(float *V,int m)

{
    float *C, *eptr, tmp ;
    int n, size;

    size = xps_ispower2(m);
    n=size>>1;
    FFT( V, size) ;
    tmp = V[1] ;
    V[1] = 0 ;

    for ( C=&(V[0]), eptr=C+size ; C<eptr ; C++ ) {
        *C = ( sq(*C) + sq(*(++C)) ) / n ;
        *C=0;
    }
    V[1] = sq( tmp ) / n ;

    iFFT( V, size) ;

    for (n=0; n<size; n++)  V[n] = V[n] / size;

}


/*************************************************************** Cepstrum **/
/*   usual cepstrum : input 'data' of size 'length'                        */
/*****************************************************************************/


void xps_Cepstrum (short *data, float *cep, int length)

{
   int size, i;

   size=xps_ispower2(length);
   bzero(&(cep[0]),size*sizeof(float));
   xps_strcopy(&(cep[0]), &(data[0]), length, 2);
   FFT(cep,size);                              /* power spectrum */

   for (i=0; i<(size/2); i++)  {

          cep[i] = sqrt( cep[2*i]*cep[2*i] + cep[2*i+1]*cep[2*i+1]);
          if (cep[i] != 0) cep[i]  = log(cep[i]);
          cep[size-i] = cep[i];
      }

   iFFT(cep,size);

   for (i=0; i<CEP_ORD; i++)
      cep[i]=cep[i] *2/size;

}


/************************************************** Improved Cepstrum **/
```

```
/*  Improved cepstrum :                                                    */
/*    Better aproximation of the spectral enveloppe. May need some tuning. */
/*    You migth want to check cepanaf.c in NUUTALK or ask Lee-Sensei or    */
/*    Satoh-San for details, eventual corrections & improvements.          */
/*************************************************************************/


void xps_Icepstrum (short *data, float *icep, int length)

{
   int size, i, j;
   float *aux, *ps, *win;


   for (i=0; i<ICEP_NUM; i++)                        /* improved cep. coefs */
                  icep_coef[i] = 1.5;

   size = xps_ispower2(length);
   aux = xalloc(MAX(CEP_ORD,size),float);
   bzero(aux, CEP_ORD*sizeof(float));
   ps = xalloc(size, float);
   bzero(ps,size*sizeof(float));
   xps_strcopy(ps,data,length,2);
   FFT(ps,size);                                                 /* spectrum */


   for (i=0; i<(size/2); i++)      {

         ps[i] = sqrt( ps[2*i]*ps[2*i] + ps[2*i+1]*ps[2*i+1]);
         if (ps[i] != 0)
            ps[i] = log(ps[i]);
         ps[size-i-1] = ps[i];                          /* power spectrum */
      }

   xps_strcopy(icep,ps,size,4);

   iFFT(icep,size);                                              /* cepstrum */
   for (i=0; i<size/2; i++)
      icep[i]=icep[2*i] / size;
   for (i=0; i<size/2; i++)
         icep[size-i] = icep[i];

   if (ICEP_NUM>0)
    {
      win=xps_Window(CEP_ORD, 1,CEP_ORD/2, 2);
      for (j=0; j<ICEP_NUM; j++)
       {
         bzero(aux, size*sizeof(float));
         for (i=0; i<CEP_ORD; i++)                        /* windowing */
          {
            aux[i] = win[i] * icep[i];
            aux[size-i-1] = aux[i];
          }

         FFT(aux,size);                                        /* fft */
         for (i=0; i<size/2; i++)
            aux[i] = aux[2*i];
         for (i=0; i<size/2; i++)
            aux[size-i-1] = aux[i];

         for (i=0; i<size; i++)            /* substract power spectrum */
            aux[i]= ps[i] - aux[i];

         iFFT(aux,size);                                       /* ifft */
         for (i=0; i<size/2; i++)
            aux[i] = aux[2*i]/size;
```



```
         for (i=0; i<size/2; i++)
            aux[size-i-1] = aux[i];

         for (i=0; i<CEP_ORD; i++)                    /* coefficients */
            aux[i]= aux[i] * icep_coef[i];

         for (i=0; i<CEP_ORD; i++)                  /* add to cepstrum */
            icep[i] += aux[i];
        }

      for (i=0; i<CEP_ORD; i++)
         icep[i] = icep[i] * win[i];

   xfree(win);

    }

   xfree(aux);
   xfree(ps);


}



/*************************************************** Voicing Detection **/
/*  Fuzzy voicing detection :                                            */
/*      From input 'data', an estimation of the degree of voicedness based */
/*      on rms, autocorrelation, cepstrum and unit-type information. 'k' is */
/*      the scaling factor for the rms (1 for the whole spectrum, >1 if  */
/*      only a given bandwidth is fed.                                   */
/*************************************************************************/


float xps_Is_Voiced(short *data, int size, int unit, float k)

{
   int i, num_zc, phoneme;
   float ac, rms, cep, voice, zc, *aux;

   k = MAX(1, k);
   ac=(rms=(cep=0));
   aux = xalloc(2*size, float);
   bzero(aux, 2*size*sizeof(float));
   xps_strcopy(aux,data,size,2);
   xps_acf(aux, size);

                                                 /* vowel, nasal or liquid */
     phoneme = ((unit==1)||(unit==2)||(unit==3)) ? 5 : 1;


   for (i=0; i<size; i++)    {
       ac += sq( aux[i]);              /* measure of periodicity */
       rms += sq(data[i]);             /* measure of power */
    }
   ac = sqrt(ac)/size;
   rms = sqrt(rms)/(float)size;

   bzero(aux, 2*size*sizeof(float));
   xps_strcopy(aux,data,size,2);
   xps_Icepstrum(data,aux ,size);

   for (i=C_MIN; i<C_MAX; i++)         /* measure of periodicity */
     cep += sq(data[i]);
   cep = cep/(float)size;
```

```
   num_zc = xps_Zero_Cross(data, size, &i, &i);    /* measure of noise */
   zc = (float)num_zc / size;

   voice = 0.00000001 * k * rms * phoneme * ( C1*ac + C2*cep + C3/zc);
   voice = xps_Sygmoid(voice);

   return (voice);

}


/************************************************** Cut Point Selection **/
/*  Pick the best-suited frames for unit concatenation :                  */
/*     Among frames of left unit in 'w1' and frames  of rigth unit in 'w2'  */
/*     pick the pair that minimizes cepstral distance ; 'pm1' and  'pm2'    */
/*     have the corresponding pitchmarks, pointing to the border marks ;    */
/*     'n1' and 'n2' have the number of periods available. Cuts near the    */
/*     edges are prefered. At most 10 periods per unit are considered.      */
/*************************************************************************/


Joint xps_Cut_Point(short *w1, short *w2, Marks *pm1, Marks *pm2)

{
   int i, j, k, n1=0, n2=0, size1[10], size2[10];
   int fra1, fra2, diff, imax, imin, jmax, jmin, lostsamp=0;
   Marks *aux1=pm1, *aux2=pm2;
   float *icep1, *icep2, cur_dif, cepdist[10][10];
   Joint cut_p;



   icep1 = xalloc(BUF_SIZ, float);
   icep2 = xalloc(BUF_SIZ, float);

   diff = 10000 * CEP_ORD;                          /* dumb default value */

   while ((aux1->prev != NULL)&&(n1<10))
     {  aux1 = aux1->prev;                                  /* left unit */
        size1[n1] = aux1->next->nsamp - aux1->nsamp;
        n1++;
      }

   while ((aux2->next != NULL)&&(n2<10))
     {  aux2 = aux2->next;                                  /* rigth unit */
        size2[n2] = aux2->nsamp - aux2->prev->nsamp;
        n2++;
      }

   aux1 = pm1->prev;
   aux2 = pm2;

   for (i=0; i<n1; i++)                              /* estimate cepstral */
    {                                               /* distances per frame */
       xps_Icepstrum(w1+aux1->nsamp, icep1,size1[i]);
       for (j=0; j<n2; j++)
        {
           cur_dif = 0;
           xps_Icepstrum(w2+aux2->nsamp, icep2,size2[j]);

           for (k=0; k<CEP_ORD; k++)

             cur_dif += sq(icep1[k]-icep2[k]);

           cepdist[i][j] = sqrt(cur_dif)/CEP_ORD;
```



```
           aux2 = aux2->next;
        }

        aux2 = pm2;
        aux1 = aux1->prev;
     }


     for (i=0; i<n1; i++)

        for (j=0; j<n2; j++)                          /* find minimal average */
         {                                               /* cepstral distance */
              imin = MAX(0, (i-1));
              imax = MIN((n1-1),(i+1));
              jmin = MAX(0, (j-1));
              jmax = MIN((n2-1),(j+1));

                                  /* we punish joints that are too distant */
                                  /* from the border of the original units */

              cur_dif = 2*cepdist[i][j]+cepdist[imin][jmin]+cepdist[imax][jmax];
              cur_dif = cur_dif * (2 - cos((i+j)*3.1416/20));

             if (cur_dif<diff)
              {  fra1=i;
                 fra2=j;
                 diff=cur_dif;
              }
         }


    for (i=0; i<fra1; i++)                                /* estimating lost duration */
       lostsamp += size1[i];

    for (j=0; j<fra2; j++);

       lostsamp += size2[j];

    lostsamp = lostsamp / udb_current->wave_sample_rate;


    cut_p.nfra1 = fra1;       /* two closest frames, and consequent duration */
    cut_p.nfra2 = fra2;
    cut_p.dur = lostsamp;
    cut_p.distance = diff;

    xfree(icep1);
    xfree(icep2);

    return(cut_p);

}
```

```
/*=========================================================================*/
/*           A T R Interpreting Telecommunications Labs                    */
/*                                                                         */
/*=========================================================================*/
/*                CHATR Speech Synthesis System                            */
/*                      Christian Lelong                                   */
/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                 Miscellaneous Functions Library                         */
/*                                                                         */
/*                                                                         */
/*                           Feb 1995                                      */
/*                     Copyright (C) 1994,1995                             */
/*     ATR Interpreting Telecommunications Research Laboratories           */
/*                     All rights reserved.                                */
/*=========================================================================*/


#include "xruc.h"

#define FIL_BEG 0.025
#define FIL_END 0.5

extern int SPAN;

void xps_push(short *buffer, int n, int m, int offset);
void xps_flush(short *buffer, int size, int offset);
void xps_strcopy(void *out, void *in, int length, int mode);
void xps_Push_Marks(Marks *pm, int npm);
int xps_Is_Alone(Marks *pm, int span);
int xps_Find_Boundary(struct Unit *punit, int i, Marks *pmarks, int npm);
void xps_Filter_Samples(int k, short *data, int start, int length, short *out);
void xps_Band_RMS(int filter, short *data, int l, int mode, float *rms);
int xps_Zero_Cross(short *data, int length, int *start, int *end);
int xps_Test_Voicing(struct Unit *pu, Marks *pm, int npm,int time, int permax);
float xps_MEAN_POWER(Utterance utt, Synthesis *S, int n);
int xps_INDEX(Marks *pm, int type);
int xps_INDEX_bis(Marks *pm, int type);

/****************************************************** Flushing a Buffer **/
/*   buffer flushing :                                                     */
/*       this will shift to the left (Firt In First Out, or FIFO) all the  */
/*       'size' samples of the 'buffer' by 'offset' values. the buffer will */
/*       be fed with zeros on its rigth side.                              */
/*************************************************************************/


void xps_flush(short *buffer, int size, int offset)

{
    int i;

    if (((buffer+size-1)==NULL)||(offset>size))
    {
        P_Error ("\n flush : sizes are not consistent \n");
        list_error(On_Error_Tag);
    }

    for (i=0; i<(size-offset-1); i++)
        buffer[i]=buffer[i+offset];

    for (i=(size-offset); i<size; i++)
        buffer[i]=0;

}
```



```
/****************************************************** Pushing a Buffer **/
/*   buffer pushing :                                                      */
/*      this will shift to the rigth all the samples, from the 'n'th to the */
/*      'm'th sample, by 'offset' values. No values are lost, provided that */
/*      'buffer' has enough spare room.                                    */
/*************************************************************************/


void xps_push(short *buffer, int n, int m, int offset)

{
    int i;

    if (((buffer+m+offset)==NULL)||(n>m))
    {
        P_Error ("\n push : sizes are not consistent \n");
        list_error(On_Error_Tag);
    }

    for (i=m; i>n; i--)
        buffer[i+offset]=buffer[i];

}

/****************************************************** Vector Duplication **/
/*  Copying for shorts and floats :                                        */
/*      copies first 'length' elements of string 'in' to string 'out', and */
/*      does the necessary type conversions : 'mode' = 1 / 2 / 3 / 4  for  */
/*      short to short / short to float / float to short / float to float  */
/*      also, 'mode' 5 for Complex type ; if 'length' = 0, all             */
/*      elements in 'in' are copied.                                       */
/*************************************************************************/


void xps_strcopy( void *out, void *in, int length, int mode)

{
   int i=0, j=length;
   short *a, *b;


   if (j==0) j=BUF_SIZ;

   switch (mode) {

       case  2  :   while ((i++<j) && (in!=NULL))
                      {
                        *((float *)out) = (float) *((short *)in);
                        out += sizeof(float);
                        in += sizeof(short);
                      }
                    break;

       case  3  :   while ((i++<j) && (in!=NULL))
                      {
                        *((short *)out) = (short) *((float *)in);
                        out += sizeof(short);
                        in += sizeof(float);
                      }
                    break;

       case  1  :   a = (short *) out;
                    b = (short *)in;
                    while ((i++<j) && (in!=NULL))
                                                  *a++ = *b++;
                    break;
```

```
      case  4  :   while ((i++<j) && (in!=NULL))
                     {
                       (float *) out = (float *)in;
                       out += sizeof(float);
                       in += sizeof(float);
                     }
                   break;

      case  5  :   while ((i++<j) && (in!=NULL))
                     {
                       (Complex *) out = (Complex *)in;
                       out += 2*sizeof(float);
                       in += 2*sizeof(float);
                     }
                   break;

    }

}

/*********************************************************** Push Pitch Marks */
/*  Shift Marks :                                                             */
/*     This shifts 'npm' pitch marks one space to the right.                  */
/*****************************************************************************/


void xps_Push_Marks(Marks *pm, int npm)

{
   int i;

   for (i=npm; i>0; i--)
    {
      pm[i].nsamp = pm[i-1].nsamp;
      pm[i].voice = pm[i-1].voice;          /* other fields uninitialized */
      pm[i].boundary = pm[i-1].boundary;
    }
}

/*********************************************************** Isolated Marks */
/*  Voiced isolated marks :                                                   */
/*     This applies to voiced_only pitch marks. Pitch modifications are       */
/*     done only on sets of at least 5 voiced consecutive pitch marks. This   */
/*     checks for the existence of this set. 'span' is the maximum number     */
/*     of miliseconds between two consecutive marks.                          */
/*****************************************************************************/


int xps_Is_Alone(Marks *pm, int span)

{
   int i=0, rate;
   Marks *aux;

   rate = udb_current->wave_sample_rate / 1000;
   aux = pm->next;
   while ((aux!=NULL)&&(aux->nsamp - pm->nsamp < rate * span)&(i<5))
   {
     i++;
     pm = aux;
     aux = pm->next;
   }

   if (i < 5) return 0;
   else       return i;                                  /* not alone */
```



```
}

/*********************************************** Find Sub-Unit Boundary */
/*   Find boundary mark :                                                */
/*      This is for PM_VOICED_ONLY marks. For every sub-unit boundary in a  */
/*      unit, we want to know if its neighbourhood is unvoiced. If so, we   */
/*      have to create a 'fake' mark. Voiced means there's a mark less than */
/*      SPAN msec away from the boundary.                                   */
/*****************************************************************************/


int xps_Find_Boundary(struct Unit *punit, int i, Marks *pmarks, int npm)

{
   int j, nsubu, boundary, span, rate, closest, aux;
   Marks pm;

   nsubu = punit->num_sub_units;
   i = MAX(0, MIN(nsubu, i));
   if (i == nsubu) boundary = punit->length;
   else boundary = punit->sub_units[i].start;
   rate = udb_current->wave_sample_rate / 1000;
   boundary = boundary * rate;                          /* all in samples */
   span = SPAN * rate / 2;

   j = 0;                                              /* marks on both sides */
   while ((j<npm-1) &&  (pmarks[j].nsamp < boundary))
    j++;

   aux = MAX(j-1, 0);
   if (abs(pmarks[j].nsamp-boundary) > abs(pmarks[aux].nsamp-boundary))
     closest = aux;
   else closest = j;                                    /* the closest mark */


   if (abs(pmarks[closest].nsamp-boundary) < span) return closest;
   else
    if (pmarks[closest].nsamp >= boundary) return -(closest+1);
    else return -(closest+2);

}

/*************************************************************** Filtering **/
/*   filtering the waveform :                                                */
/*     'length' samples from signal in 'data' are filtered by 'filter'       */
/*     starting from sample number 'start', 'out' is at least BUF_SIZ long   */
/*****************************************************************************/


void xps_Filter_Samples (int k, short *data, int start, int length, short *out)

{
    int i, j, l, stop, size;
    float aux;

    for (i=start; i<length; i++)
     {
        aux=0;
        stop = MIN( FIL_ORD, start);

        if (stop == FIL_ORD)
          for (j=0; j<stop; j++)
             aux += Filter_Bank[j][k] * *(data + (i - j));
        else
         {
```

```
          size = length - start;
          for (j=0; j<FIL_ORD; j++)
           {
             l = i - j;
             while (l < 0) l += size;
             aux += Filter_Bank[j][k] * data[l];
           }
        }

      out[i-start]=(short)aux;
    }

  bzero((out+length-start), (BUF_SIZ - length+start)*sizeof(short));

}

/*****************************************************************Band-Pass RMS **/
/*   set / get rms in a given band-width :                                       */
/*     given a band-pass 'filter', get/set ('mode' 1 / 0) from the fil-          */
/*     tered signal 'data' of 'l' samples the normalised value of the 'rms'      */
/*****************************************************************************/


void xps_Band_RMS(int filter, short *data, int l, int mode, float *rms)

{
  int i;
  float power, scale;
  short aux[BUF_SIZ];

  power=0;
  xps_strcopy(aux, data, l, l);
  xps_Filter_Samples(filter, data, l, l, aux);
  for (i=0; i<l; i++) power += sq(aux[i]);
  power /= l;
  switch (mode)
  {
    case '0' : scale = sqrt((*rms) / power);
               for (i=0; i<l; i++)
                 *(data++) = scale * aux[i];
               break;

    default  : *rms = power;
               break;
  };

}


/************************************************** Get the Zero-crossings **/
/*  for input 'data' of 'length' samples, finds the first and last zero-     */
/*  crossings, and returns their total number.                               */
/*****************************************************************************/


int xps_Zero_Cross(short *data, int length, int *start, int *end)

{
  short *p1, *p2;
  int i, cross, total;

  p1 = data;
  cross = *p1++;
  i = 0;
  while ((i<length)&&((cross * *p1++) > 0))
      i++;                                  /* detect the 1st zero crossing */
```



```
  if (i == length-1)  return 100;                              /* none found */

  if (Abs(*p1) > Abs(*(p1 -1))) *start=i-1;
  else *start=i;

  p2 = data + length - 1;
  cross = *p2--;
  i = length -1;
  while ((cross * *p2--) > 0) i--;        /* detect the first zero crossing */
  if (Abs(*p2) > Abs(*(p2 +1))) *end=i+1;
  else *end=i;

  total = 1;                              /* count all the zero crossings */
  i = *start;
  cross = *p1;

  while (i != *end)   {

     while ((i<length)&&((cross * *p1++) > 0)) i++;

       if (i == length-1) goto END;     /* only one zero crossing */

       cross = *p1;
       total++;
   }

  total++;                                /* count the last one */

  END : return total;

}


/************************************************************ Look for Marks */
/*  Voicing test :                                                           */
/*     When larynx-derived pitch marks are used, voicing at a given time is */
/*     defined by the presence of pitchmarks, no more than 'permax' samples */
/*     apart. If they exist, this returns '1' or '-1', and '0' otherwise.   */
/*****************************************************************************/


int xps_Test_Voicing(struct Unit *pu, Marks *pm, int npm, int time, int permax)

{
  int distance, i, left, rigth;
  Marks *current;


  if ((time > pu->length)||(time < 0))                    /* bad arguments */
                                       return 0;
  if ((pm == NULL)||(npm == 0))                           /* unvoiced unit */
                                       return 0;
  i = 0;
  while ((i < npm)&&(time > (pm+i)->nsamp)) i++;        /* 2 closest marks */
  left = abs((pm+i-1)->nsamp - time);
  rigth = abs((pm+i)->nsamp - time);
  distance = (i == 0) ? abs(pm->nsamp - time) : MIN(left, rigth);

  if (distance > permax)    return 0;
  else  if (left>rigth)   return 1;                     /* rigth is closer */
        else return -1;                                  /* left is closer */

}

/************************************************************ Mean Power **/
```

```
/*  Average Sub-Unit power :                                                */
/*     This returns, for a given unit, an estimate of its  average power,  */
/*     defined by : log(s^2), averaged over every sub-unit. Usual values   */
/*     range from 5 to 9, depending on the phoneme type, and the speaker.  */
/**************************************************************************/


float xps_MEAN_POWER(Utterance utt, Synthesis *S, int n)

{
   int i, j, k, total;
   Stream u;
   struct Unit *pu;
   struct Sub_Unit *psu;
   float power;


   power =(total = 0);                  /* could be better ... */
   u = UNITSTREAM(utt);

   for (i=0; i<n; i++)  u = SC_next(u);

   pu = SC(u, Unit);
   psu = pu->sub_units;

   while (total < S->nsubu[n])
    {
      for (i=0; i<udb_current->num_nus_phones; i++)

        if (streq(psu->name, udb_current->nus_phones[i].name))

           j = i;          /* sub-unit belongs to ith class */

      power += udb_current->nus_phones[j].pow_mean;
      total++;
      psu++;
    }

   power = power / S->nsubu[n];

   return(power);

}


/*********************************************************** Index in a Unit */
/*  Index for power processing :                                            */
/*    This simply returns the index of the pitch for a given unit, where the */
/*    power modification is supposed to begin (rigth end, type == 1) or end  */
/*    (left end, type == -1). This mark is chosen at the middle of the last/ */
/*    first sub-unit, unless the sub-unit is a stop (we don't want to modify */
/*    stops), in which case it returns the limit of the unit.                */
/**************************************************************************/


int xps_INDEX(Marks *pm, int type)

{
   int size;
   Marks *current;


   current = pm;

   if (type == 1)
    {
```



```
      while (current->next != NULL)  current = current->next;
      if (current->prev->phoneme == 7) return current->rank; /* no modifs */
      size = current->rank;
      if (size <2) return current->rank;                /* too short a unit */
      current = current -2;
      while (current->boundary==0) current=current->prev;/* start of subunit */

      return ((int)((size+current->rank+0.5)/2));      /* middle of sub-unit */
    }
   else
    {
      if (current->phoneme == 7) return current->rank;     /* update if nec. */
      current = current->next;
      while (current->boundary == 0) current = current->next;
      size = current->rank;
      if ((size<2)||(current->prev->phoneme == STOP)) return 0;

      return (size/2);
    }
}

/*********************************************************** Index in a Unit */
/*  Index for power processing :                                            */
/*    This simply returns the index of the sample for a given unit where the */
/*    power modification is supposed to begin (rigth end, type == 1) or end  */
/*    (left end, type == -1). Used for larynx pitch marks.                   */
/**************************************************************************/


int xps_INDEX_bis(Marks *pm, int type)

{
   int size, totsize;
   Marks *current;


   current = pm;

   if (type == 1)
    {
      while (current->next != NULL)  current = current->next;
      size = current->rank;
      totsize = current->nsamp;
      current = current -1;
      while (current->boundary==0) current=current->prev;/* start of subunit */

      if (current->phoneme != 7)                                /* a stop ? */
      return ((int)((totsize + current->nsamp)/2));    /* middle of sub-unit */
      else return totsize;
    }
   else
    {
      current = current->next;
      while (current->boundary == 0) current = current->next;
      size = current->nsamp;

      if (pm->phoneme != 7)                                    /* a stop ? */
      return (size/2);                                /* middle of sub-unit */
      else return 1;

    }
}
```

```
/*************************************************************************/
/*          A T R Interpreting Telecommunications Labs                   */
/*                                                                       */
/*************************************************************************/
/*              CHATR Speech Synthesis System                            */
/*                    Christian Lelong                                   */
/*-----------------------------------------------------------------------*/
/*                                                                       */
/*                    Speech Signal Models                               */
/*                                                                       */
/*                         Feb 1995                                      */
/*                  Copyrigth (C) 1994, 1995                             */
/*    ATR Interpreting Telecommunications Research Laboratories          */
/*                    All rights reserved.                               */
/*************************************************************************/



#include <stdio.h>
#include <string.h>
#include <math.h>

#include "alloc.h"
#include "xruc.h"


#define SM 10000
#define Sm 400
#define FIL_STP 0.0015625
#define WMAX 50                                         /* 50 harmonics max */
#define THRES 4
                                    /* Cut_Wo is never used, but tells the */
                                /* cutting frequencies of the loaded filters */

int Cut_Wo[] = {0, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15,
                0.175, 0.2, 0.2375, 0.2875, 0.35, 0.425, 0.5 };

const int BANK[] = {0,16,32,48,64,80,96,112,128,152,184,224,272,320};

const int FMAX[] = {2000, 3000, 4000, 5000, 6000};

static int f_curr = 2;

static float *cosinus[WMAX];
static float *sinus[WMAX];
static float *Ak;
static float *Bk;
static short *harmo;


void xps_Stochastic(Synthesis *S, int rank, short *stoch);
void xps_Harmonic(Synthesis *S, int rank, short *harmonic, Scheme sch);
void xps_H_psola(short *output, Marks *pm, int wnum, int shift);
float *xps_Weigth(Marks *pm, int npm);
void xps_V_Bands(Synthesis *S, int rank);
void xps_NOISE_PRO(Synthesis *S, int rank, int type);
void xps_Hybrid(Synthesis *S, Scheme sch);
void xps_Bands(Synthesis *S);



/************************************ Calculate the Stochastic Component **/
/*   Extraction of the stochastic component :                            */
/*     for input waveform 'signal', whose 'npm' pitch mark set is 'pm', it */
/*     returns the corresponding stochastic component. for more details,  */
/*     check Boeffard & Violaro 's paper on the hybrid model (CNET 1994). */
```



```
/*                                                                       */
/*                    NEVER TESTED  -  NEVER USED                        */
/*                                                                       */
/*************************************************************************/


void xps_Stochastic(Synthesis *S, int rank, short *stoch)

{
   int Snum, Wnum, i, j, k, length, npm, shift;
   float power, h_power, period;
   Marks *mark, *pm;
   short *signal;
   double w;


   npm = S->nrfpm[rank];
   pm = S->ref[rank];
   signal = S->wl[rank];                                  /* original signal */

   power = xps_RMS(signal, pm, npm-1);                  /* average unit power */

   Snum = npm - 2;

   for (i=0, mark = pm->next; i<Snum; i++)
    {
      length = mark->next->nsamp - mark->prev->nsamp;  /* ASsymetric window */
      period = length / (1.0*S->srate);                              /* in msec */
      w = 2*PI/length;                                            /* in samples */
      Wnum = (int) FMAX[f_curr] * period;              /* number of harmonics */
      Wnum = MIN(Wnum, WMAX);

      for (j=1; j<Wnum; j++)
        for (k=0; k<length; k++)
         {
           cosinus[j][k] = (float) cos(w*j*k);       /* fill the 2 matrices */
           sinus[j][k] = (float) sin(w*j*k);
         }

      shift = mark->prev->nsamp;
      for (j=1; j<Wnum; j++)
       {
         Ak[j] = (Bk[j] = 0);
         for (k=0; k<length; k++)
          {
            Ak[j] += signal[shift+k] * cosinus[j][k];     /* scalar product */
            Bk[j] += signal[shift+k] * sinus[j][k];
          }
         Ak[j] = Ak[j] * 2 / length;                       /* normalisation */
         Bk[j] = Bk[j] * 2 / length;
       }

     xps_H_psola(stoch, mark, Wnum, 0); /* short-term windowed harmonic signal */

     mark++;

    }

                                                  /* adjusting limit frequency */
     h_power = xps_RMS(stoch, pm, npm-1);

     if ((h_power/power) > 0.9) f_curr = MAX(0, f_curr-1);

     if ((h_power/power) < 0.2) f_curr = MIN(THRES, f_curr+1);
```

```
}



/************************************* Calculate the Harmonic Component **/
/*   Extraction of the Harmonic component :                              */
/*     for input waveform 'signal', whose 'npm' pitch mark set is 'pm', it  */
/*     returns the corresponding harmonic component. for more details,     */
/*     check Boeffard & Violaro 's paper on the hybrid model (CNET 1994).  */
/*     when required, psola is performed here (makes things faster).       */
/************************************************************************/


void xps_Harmonic(Synthesis *S, int rank, short *harmonic, Scheme sch)

{
   int Snum, Wnum, i, j, k, l, length, npm, shift, shift2;
   float period, *weigth, aux1, aux2;
   Marks *mark, *pm;
   short *signal;
   double w;


/*
**  This is very operation-intensive, and quite slow. So far I haven't
**  thougth of ways to improve it, but I think it would be worthwhile.
*/

   npm = S->nrfpm[rank];
   pm = S->ref[rank];

   weigth = xps_Weigth(pm, npm);                    /* corrective weigthing */
   length = pm[npm-1].nsamp;
   signal = S->wl[rank];

   Snum = npm - 2;

   for (i=0, l=0, mark = pm->next; i<Snum; i++)
    {
      length =mark->next->nsamp - mark->prev->nsamp;   /* assymetric window */
      period = length / (float)S->srate;                          /* in sec */
      w = 2*PI/length;                                  /* in samples */
      Wnum = (int) FMAX[f_curr] * period;            /* number of harmonics */
      Wnum = MIN(Wnum, WMAX);

/*
**   We haven't designed yet a way of seting the maximum number of harmonics,
**   Wnum, so that it adaptas to different speakers, speech styles and even
**   phonemes. Should be worth it. For the time being, an intermediate value.
*/


                                /* an average of 15 and N harmonics, where */
      Wnum = (15 + Wnum) / 2;       /* N is the number it takes to go up to */
                                    /* FMAX[f_curr], currently 4000 Hz */

      for (j=1; j<Wnum; j++)
        for (k=0; k<length; k++)
          {
            cosinus[j][k] = (float) cos(w*j*k);      /* fill the 2 matrices */
            sinus[j][k] = (float) sin(w*j*k);
          }

      shift = mark->prev->nsamp;
      for (j=1; j<Wnum; j++)
       {
```



```
         Ak[j] = (Bk[j] = 0);
         for (k=0; k<length; k++)
          {
            Ak[j] += cosinus[j][k] * (float)signal[shift+k];
            Bk[j] += sinus[j][k] * (float)signal[shift+k];
          }
         Ak[j] = 2*Ak[j] / length;                        /* normalisation */
         Bk[j] = 2*Bk[j] / length;
       }


   if ((sch.P_method == 2) && (sch.D_method == 3)       /* PSOLA rigth away */
            &&(sch.test != 2))
    {
      shift2 = mark->prev->nsamp - S->tar[rank][l++].nsamp;
      if (S->map[rank][i+1].repeat == 0)
        xps_H_psola(harmo, mark, Wnum, -shift2);                    /* copy */
      if  (S->map[rank][i+1].repeat == 1)
       {
        xps_H_psola(harmo, mark, Wnum, -shift2);              /* duplicate */
        shift2 = mark->prev->nsamp - S->tar[rank][l++].nsamp;
        xps_H_psola(harmo, mark, Wnum, -shift2);
       }
      if (S->map[rank][i+1].repeat == -1) l--;                /* eliminate */
    }

   else

   xps_H_psola(harmo, mark, Wnum, 0);


   mark = mark->next;

  }

   for (k=0; k<(pm+npm-1)->nsamp; k++)
    {
      S->w2[rank][k] = S->wl[rank][k] - harmo[k];
      S->wl[rank][k] = harmo[k];
    }


   xfree(weigth);

}


/********************************** Overlap and Add for the Hybrid Model **/
/*  Build st-signals :                                                    */
/*     Builds harmonic signal from the coordonates in the sine/cosine base, */
/*     stored in Ak and Bk, by windowing, overlapping & adding, using assy- */
/*     metrical windows. The signal is centered at mark 'pm'.             */
/************************************************************************/


void xps_H_psola(short *output, Marks *pm, int wnum, int shift)

{
   int length, middle, i, j;
   float *win, aux;


   length = pm->next->nsamp - pm->prev->nsamp;
   middle = pm->nsamp - pm->prev->nsamp;
   win = xps_Window(length, 1, middle, 1);
```

```
   if (pm->prev->prev == NULL)                              /* left limit */
     for (i=0; i<middle; i++)
       win[i] = 1;

   if (pm->next->next == NULL)                              /* rigth limit */
     for (i=middle; i<length; i++)
       win[i] = 1;

   for (i=0; i<length; i++)
    {
      for (j=1, aux = 0; j<wnum; j++)
         aux += Ak[j] * cosinus[j][i] + Bk[j] * sinus[j][i];

      output[pm->prev->nsamp+shift+i] += (short)(win[i]*aux);
    }

   xfree(win);

}


/******************************** Weigthing for Hybrid Harmonic Component **/
/*   Calculate correcting weigthing :                                     */
/*       this will correct the amplitude distortions introduced in the    */
/*       estimation of the harmonic component for the hybrid model, due to */
/*       the fact that windows used are symetrical, and therefore they do */
/*       not add up to one. This returns the inverse, sample by sample, of */
/*       the sum of the windows used.                                     */
/*************************************************************************/


float *xps_Weigth(Marks *pm, int npm)

{
   int i, j, length, middle, size;
   float *output, *win;
   Marks *mark;

   size = (pm+npm-1)->nsamp + 1;
   output = xalloc(size, float);


   for (i=0; i<size; i++)
      output[i] = 0;

   for (i=1, mark = pm->next; i<(npm-1); i++)
    {
      length = mark->next->nsamp - mark->prev->nsamp;
      middle = mark->nsamp - mark->prev->nsamp;
      win = xps_Window( length, 1, middle, 1);

      if (mark->prev->prev == NULL)                          /* left limit */
        for (j=0; j<middle; j++)
          win[j] = 1;

      if (mark->next->next == NULL)                          /* rigth limit */
        for (j=middle; j<length; j++)
          win[j] = 1;

      for (j=0; j<length; j++)
         output[mark->prev->nsamp + j] += win[j];

      mark++;
    }

   xfree(win);
```



```
   for (i=0; i<size; i++)
      output[i] = MAX(0.9, MIN(1.1, 1 / output[i]));

   return (output);

}


/************************************************ Band Voicing Detection **/
/*   Voicing detection :                                                  */
/*      for a given signal 'w' with a 'pm' pitch mark set of 'npm' marks,  */
/*      tests for every bandwidth and for every pitch period the degree of */
/*      voicing ; if positive, the filtered signal is added to 'harm',     */
/*      otherwise it is added to 'stoch'. In order to avoid too many       */
/*      voiced/unvoiced transitions, which could be annoying, changes occur */
/*      over two pitch periods.                                            */
/*************************************************************************/


void xps_V_Bands(Synthesis *S, int rank)

{
   int i, j, l, b[BANDS], start, size, beg, end, phon;
   int k[] = { 20, 20, 20, 20, 20, 20, 20, 20, 13, 10, 8, 7, 7};
   short *buffer, *h, *s;
   float voice;

/*
** b is used to avoid too frequent v/uv transitions ; k allows to
** correct the gap in energy between bands, for voicing detection
*/

   for (i=0; i<BANDS; i++) b[i] = 0;
   start = 0;

   buffer = xalloc(BUF_SIZ, short);
   h = xalloc(BUF_SIZ, short);
   s = xalloc(BUF_SIZ, short);

   for (i=0; i<(S->nrfpm[rank]-1); i++)   {

      size = S->ref[rank][i+1].nsamp - S->ref[rank][i].nsamp + 1;
      bzero(h, BUF_SIZ*sizeof(short));                  /* harmonic component */
      bzero(s, BUF_SIZ*sizeof(short));                /* stochastic component */

      for (j=0; j<BANDS; j++)   {

         beg = S->ref[rank][i].nsamp;
         end = S->ref[rank][i+1].nsamp;
         phon = S->ref[rank][i].phoneme;

         if (j < BANDS-1)
           xps_Filter_Samples(j, S->wl[rank], beg, end, buffer);
         else
           for (l=0; l<size; l++)                              /* so it adds up */
             buffer[l] = S->wl[rank][beg+l] - s[l] - h[l];
         voice = xps_Is_Voiced(buffer, size, phon, k[j]);

                                                             /* classification */
         if (voice>0.5)

             if (b[j] == -1)  {  b[j] == 0;
                                 for (l=0; l<size; l++) s[l] += buffer[l]; }
             else             {  b[j] == 1;
                                 for (l=0; l<size; l++) h[l] += buffer[l]; }
```

```
        else

            if (b[j] == 1)   {  b[j] == 0;
                               for (l=0; l<size; l++) h[l] += buffer[l]; }
            else             {  b[j] == -1;
                               for (l=0; l<size; l++) s[l] += buffer[l]; }

      }


      for (l=0; l<size; l++)   {              /* copy the two final signals */

         S->w1[rank][start+l] = h[l];
         S->w2[rank][start+l] = s[l];
       }

      start += size-1;

   }


}



/*********************************************************** Noise Processing */
/*  Prosodic modifications for unvoiced component :                           */
/*    This applies to the noisy component of the speech signal, whatever     */
/*    the algorithm used to obtain it. 'type' selects the processing.        */
/*****************************************************************************/


void xps_NOISE_PRO(Synthesis *S, int rank, int type)

{
   int i, j, k, diff, rlength, tlength, chunk, total;
   float scale;
   short *signal;
   Marks *rpm, *tpm;
   Map *m;


   m = S->map[rank];
   rpm = S->ref[rank];
   tpm = S->tar[rank];
   signal = S->w2[rank];
   type = (type >3) ? 1 : 0;

   bzero(harmo, S->nsamp[rank]*sizeof(short)); /* we use harmo for the noise */

   for (i=0; i<S->nrfpm[rank]; i++)
    {
      rlength = rpm->next->nsamp - rpm->nsamp;
      tlength = tpm->next->nsamp - tpm->nsamp;

      if (type = 1)                                /* method 1 : cut & paste */
       {
         switch (m->repeat)   {


          case 0 :
LAB:      diff = rlength - tlength;
         if (diff >= 0)                                          /* cutting */
          {
            xps_strcopy(harmo+tpm->nsamp, signal+rpm->nsamp, tlength/2, 1);
            xps_strcopy(harmo+tpm->nsamp+tlength/2, signal+rpm->nsamp+rlength+
```



```
                                  tlength/2 - tlength, tlength-tlength/2, 1);
          }
         else                                                    /* duplicating */
          {
             chunk = rlength / 4;
             xps_strcopy(harmo+tpm->nsamp, signal+rpm->nsamp, 3*chunk, 1);
             total = tpm->nsamp+3*chunk;
             while (diff < 2*chunk)
              {
                xps_strcopy(harmo+total, signal+rpm->nsamp+chunk, 2*chunk, 1);
                diff += 2 * chunk;
                total += 2 * chunk;
              }

             xps_strcopy(harmo+total, signal+rpm->next->nsamp+diff-chunk-
                         rlength+4*chunk, chunk-diff+rlength-4*chunk, 1);
          }
         break;

           case 1 :
          diff = rlength - tlength;
          if (diff >= 0)                                          /* cutting */
           {
             xps_strcopy(harmo+tpm->nsamp, signal+rpm->nsamp, tlength/2, 1);
             xps_strcopy(harmo+tpm->nsamp+tlength/2, signal+rpm->nsamp+rlength+
                                  tlength/2 - tlength, tlength-tlength/2, 1);
           }
          else                                                    /* duplicating */
           {
              chunk = rlength / 4;
              xps_strcopy(harmo+tpm->nsamp, signal+rpm->nsamp, 3*chunk, 1);
              total = tpm->nsamp+3*chunk;
              while (diff < 2*chunk)
               {
                 xps_strcopy(harmo+total, signal+rpm->nsamp+chunk, 2*chunk, 1);
                 diff += 2 * chunk;
                 total += 2 * chunk;
               }

              xps_strcopy(harmo+total, signal+rpm->next->nsamp+diff-chunk-
                          rlength+4*chunk, chunk-diff+rlength-4*chunk, 1);
           }

          tpm = tpm->next;
          tlength = tpm->next->nsamp - tpm->nsamp;
          goto LAB;
          break;

            default :
          break;
          }
        }
       else                                        /* method 2 : interpolation */
        {
          switch (m->repeat)   {
            case 0 :
          scale = tlength / (float)rlength;
          for (j=0; j<tlength; j++)
           {
             k = MAX(0, MIN(rlength-1, (int)(j*scale)));
             xps_strcopy(harmo+tpm->nsamp+j, signal+rpm->nsamp+k, 1, 1);
           }
          break;

            case 1 :
          scale = tlength / (float)rlength;
```

```
        for (j=0; j<tlength; j++)
         {
           k = MAX(0, MIN(rlength-1, (int)(j*scale)));
           xps_strcopy(harmo+tpm->nsamp+j, signal+rpm->nsamp+k, 1, 1);
         }
        tpm = tpm->next;
        tlength = tpm->next->nsamp - tpm->nsamp;
        scale = tlength / (float)rlength;
        for (j=0; j<tlength; j++)
         {
           k = MAX(0, MIN(rlength-1, (int)(j*scale)));
           xps_strcopy(harmo+tpm->nsamp+j, signal+rpm->nsamp+k, 1, 1);
         }
        break;

         default :
        break;
       }
     }

      rpm = rpm->next;
      if (m->repeat != -1) tpm = tpm->next;
      m++;

    }
                                                        /* last sample */
   xps_strcopy(harmo + S->nsamp[rank]-1,
                       signal + S->ref[rank][S->nrfpm[rank]-1].nsamp, 1, 1);

   for (i=0; i<S->nsamp[rank]; i++)

      S->w1[rank][i] += harmo[i];                  /* re-unite the signals */

   xfree(harmo);

}


/************************************************ Hybrid Model Algorithm **/
/*  Hybrid model based synthesis algorithm :                              */
/*    Based on Boeffard & Violaro's 'hybrid model' for separating harmonics */
/*    and the stochastic components in the speech signal ; this is done for */
/*    all the units. When PSOLA is selected, all prosodic modifications are */
/*    done at the same time (it's faster that way).                       */
/*************************************************************************/


void xps_Hybrid(Synthesis *S, Scheme sch)

{
   int i, j, size, sizmax=0, npmax=0, permax=0;
   short *aux;
   float step;

/*
**  This actually differs from the original method, which I don't fully
**  understand. So, this is experimental and open to suggestions. We
**  calculate for every short-term signal, its corresponding fourier
**  coefficients : up to a given frequency, they give us the harmonic
**  component. The rest is stochastic.
*/

   for (i=0; i<S->nunits; i++)
    {
      sizmax = ((S->nrfpm[i]>1)&&(S->ref[i][S->nrfpm[i]-1].nsamp > sizmax))
                                  ? S->ref[i][S->nrfpm[i]-1].nsamp : sizmax;
```



```
      npmax = (S->nrfpm[i]>npmax) ? S->nrfpm[i] : npmax;
      for (j=0; j<S->nrfpm[i]-1; j++)
       permax = MAX(permax, S->ref[i][j+1].nsamp - S->ref[i][j].nsamp);
    }

   harmo = xalloc(sizmax, short);
   Ak = xalloc(permax, float);
   Bk = xalloc(permax, float);

   for (i=0; i<WMAX; i++)
    {
      cosinus[i] = xalloc(2*permax, float);          /* maximum window size */
      sinus[i] = xalloc(2*permax, float);
      for (j=0; j<(2*permax); j++)
       {                                                      /* set to zero */
         cosinus[i][j] = 0.0;                                  /* all coeffs */
         sinus[i][j] = 0.0;
       }
    }

   for (i=0; i<S->nunits; i++)   {

    if (S->ntgpm[i]  > 1)  {

      size = S->nsamp[i];

/*
**   In the original method, this is used to calculate the stochastic
**   component :

                                xps_Stochastic(S, i, aux);


**   Here the harmonic component is calculated. The stochastic signal is
**   merely the difference between the original and harmonic signals. HERE
**   is where we get away from Boeffard & Violaro.
*/
      bzero(harmo, sizmax*sizeof(short));

      xps_Harmonic(S, i, harmo, sch);

   }

   else
     {
       S->w1[i] = NULL;
       S->nrfpm[i] = 0;
       S->w2[i] = NULL;
    }

   }


   for (i=0; i<WMAX; i++)                          /* (vast) memory freeing */
    {
      xfree(cosinus[i]);
      xfree(sinus[i]);
    }

   xfree(Ak);
   xfree(Bk);


}
```

```
/******************************************** Multi Bands Model Algorithm **/
/*  Multi bands model based synthesis algorithm :                         */
/*      Rather than labeling each whole-spectrum frame as either voiced or*/
/*      unvoiced, this method consists in testing the voicing degree for  */
/*      each of a given set of bandwidths, therefore creating a (rather)  */
/*      harmonic signal and a more stochastic one.                        */
/*************************************************************************/


void xps_Bands(Synthesis *S)

{
   int i, j, size, *map, sizmax=0;
   short *aux;
   float step;


   for (i=0; i<S->nunits; i++)

       sizmax = MAX(sizmax, S->nsamp[i]);

   aux = xalloc(sizmax, short);


   for (i=0; i<S->nunits; i++)
    {
     if (S->ntgpm[i]  > 1)
      {
       size = S->nsamp[i];
       xps_strcopy(aux, S->wl[i], size, 1);
       xps_V_Bands(S, i);
      }

     else
      {
       S->w1[i] = NULL;
       S->w2[i] = NULL;
     }
  }

   harmo = aux;

}
```

```
/*-------------------------------------------------------------------*/
/*          A T R Interpreting Telecommunications Labs               */
/*                                                                   */
/*-------------------------------------------------------------------*/
/*                CHATR Speech Synthesis System                      */
/*                        Christian Lelong                           */
/*-------------------------------------------------------------------*/
/*                                                                   */
/*            Prosodic Modification Related Functions                */
/*                                                                   */
/*                           Feb 1995                                */
/*                     Copyrigth (C) 1994, 1995                      */
/*     ATR Interpreting Telecommunications Research Laboratories     */
/*                        All rights reserved.                       */
/*-------------------------------------------------------------------*/


#include <stdio.h>
#include <string.h>
#include <math.h>

#include "alloc.h"
#include "xruc.h"



int vowel2[] =        { 3, 3, 3, 1, 2, 1, 1, 1 };
int liquid2[] =       { 1, 0, 2, 1, 1, 1, 1, 1 };
int nasal2[] =        { 3, 2, 3, 1, 1, 1, 1, 1 };
int affricative2[] =  { 1, 1, 1, 1, 1, 1, 1, 1 };
int fricative2[] =    { 2, 1, 1, 1, 1, 1, 1, 1 };
int closure2[] =      { 1, 1, 1, 1, 1, 1, 1, 1 };
int stop2[] =         { 1, 1, 1, 1, 1, 1, 1, 1 };
int silence2[] =      { 1, 1, 1, 1, 1, 1, 1, 1 };


int *RULE1[8] = { &vowel2[0], &liquid2[0], &nasal2[0], &affricative2[0],
                  &fricative2[0], &closure2[0], &stop2[0], &silence2[0] };


short *xps_ST_Signal(short *data, Marks *pm, int width);
void xps_MAKE_TARGET(Synthesis *S, Scheme sch);
void xps_MAKE_UNIT(Synthesis *S, int rank, Scheme sch);
void xps_MAKE_SUB_UNIT(short *wave, Marks *ref, Marks *tar, Map *m, Scheme sch, short
 *buffer);
void xps_PM1_intrapol(short *wave, Marks *ref, Map *p, short *buffer);
void xps_PM2_psola(short *in, Marks *ref, Marks *tar, int shift, short *out);
void xps_PM3_fft(short *wave, Marks *ref, Map *map, short *buffer);
void xps_DM1_dumb(short *wave, Marks *ref, int type, short *buffer);
int xps_DM2_select(short *wave, Marks *ref, int type, short *buffer);
void xps_DM3_psola(short *wave, Marks *ref, int type, short *buffer);
void xps_all_PSOLA(short *wave,Marks *ref,Marks *tar,Map *map,short *buffer);
void xps_NP(short * wave, Marks *ref, Marks *tar, int type, short *buffer);
int xps_CONCAT(Synthesis *S, int rank, P_Wave pw, int index, int method);



/************************************************** Short Term Signal **/
/*    Short term signal :                                            */
/*       This returns the windowed short term signal corresponding to pitch */
/*       mark 'pm' ; the window is 2*'width' frames long.            */
/*********************************************************************/


short *xps_ST_Signal(short *data, Marks *pm, int width)
```



```
{
   int start, end, left, rigth, size, offset, i, wsize, wleft, wrigth;
   float *win;
   short *output;

                                                  /* boundaries of ST signal */
   if (pm->prev == NULL) start = 0;
   else if ( (pm->prev->prev == NULL)||(width==1)) start = 1;
        else if ((pm->prev->prev->prev == NULL)||(width == 2))  start = 2;
             else start = 3;

   if (pm->next == NULL) end = 0;
   else if ( (pm->next->next == NULL)||(width==1)) end = 1;
        else if ((pm->next->next->next == NULL)||(width == 2))  end = 2;
             else end = 3;

                                                          /* window halves */
   if (start == 0) wleft = 0;
   else if (width > start) wleft = width * (pm->nsamp - pm->prev->nsamp);
        else if (width == 1)  wleft = pm->nsamp - pm->prev->nsamp;
             else if (width == 2)  wleft = pm->nsamp - pm->prev->prev->nsamp;
                  else wleft = pm->nsamp - pm->prev->prev->prev->nsamp;

   if (end == 0) wrigth = 0;
   else if (width > end) wrigth = width * (pm->nsamp - pm->next->nsamp);
        else if (width == 1)  wrigth = pm->next->nsamp - pm->nsamp;
             else if (width == 2)  wrigth = pm->next->next->nsamp - pm->nsamp;
                  else wrigth = pm->next->next->next->nsamp - pm->nsamp;


   wsize = wrigth + wleft + 1;                                 /* window size */

   if (start == 0) left = pm->nsamp;
   if (start == 1) left = pm->prev->nsamp;
   if (start == 2) left = pm->prev->prev->nsamp;
   if (start == 3) left = pm->prev->prev->prev->nsamp;

   if (end == 0) rigth = pm->nsamp;
   if (end == 1) rigth = pm->next->nsamp;
   if (end == 2) rigth = pm->next->next->nsamp;
   if (end == 3) rigth = pm->next->next->next->nsamp;

   size = rigth - left + 1;                                 /* ST signal size */

   output=xalloc(size, short);
   bzero(output, size*sizeof(short));

   win=xps_Window(wsize, width, wleft+1, 1);                /* hanning window */

   offset = pm->nsamp-left;

   for (i= -offset; i<(rigth - pm->nsamp); i++)

      output[i+offset]=(short)(win[i+wleft] * data[ pm->nsamp + i]);



   xfree(win);

   return output;

}


/************************************************ Make The Target Marks **/
/*   Make target pitch marks :                                       */
```

```
/*        This creates for each unit the appropriate set of target pitch   */
/*        marks and updates consequently the field 'nsamp' in 'S'. Target  */
/*        pitch marks take into account reference marks and mapping, but   */
/*        also the scheme 'sch' employed.                                  */
/***************************************************************************/


void xps_MAKE_TARGET(Synthesis *S, Scheme sch)

{
   int i, unit, duration, number, rank, npm, total, offset, flag;
   int diff, curdif, shift, beg, end, j, newduration, nsilpm, k, forgot;
   Marks *auxrf, *auxtg;
   Map *auxmap;



   for (unit=0; unit<S->nunits; unit++)                          /* each unit */
     {
       rank = 0;
       offset = 0;
       total = 0;
       flag = 0;
       npm = S->nrfpm[unit];


       for (i=0; i<(npm-1); i++)                          /* each pitch period */
         {
           auxrf = (S->ref[unit]+i);
           auxtg = (S->tar[unit]+rank);
           auxmap = (S->map[unit]+i);

           if (auxrf->phoneme == 8)                 /* special silence routine */
            {
              j = (flag = 1);
              while (((i+j+1)<S->nrfpm[unit])&&((auxrf+j)->phoneme == 8)
                     &&((auxrf+j)->boundary == 0))
                 j++;
              duration = auxrf[j].nsamp - auxrf[0].nsamp;
              newduration = (int) duration * auxmap[i].D_modif;

              auxtg[0].nsamp = offset;                /* silences fit into one */
              offset += newduration;                       /* big pitch period */
              auxtg[0].rank = rank++;
              auxtg[0].phoneme = 8;
              auxtg[0].boundary = 1;
              auxtg[0].forbid = 0;
              auxtg[0].voice = 0;
              auxtg[0].prev = (rank==1) ? NULL : auxtg-1;
              auxtg[0].next = auxtg+1;

             rank ++;
             total += 1;
             i += j;
             if (i+1 == npm) auxtg[1].next = NULL;
             else i--;                                    /* beginning of next su */
            }
           else
            {
           if (flag == 1)                           /*  that's the way it is... */
            (auxtg - 1)->boundary = 0;
           flag = 0;

           duration = (1/auxmap->P_modif) * ((auxrf+1)->nsamp - auxrf->nsamp);

               switch (auxmap->repeat)
```



```
               {
                 case -1 :   break;                          /* elimination */

                 case 1  :   auxtg->nsamp = offset;          /* duplication */
                             offset += duration;
                             (auxtg+1)->nsamp = offset;
                             offset += duration;
                             auxtg->rank = rank++;
                             (auxtg+1)->rank = rank++;
                             auxtg->phoneme = auxrf->phoneme;
                             (auxtg+1)->phoneme = 0;
                             auxtg->voice = auxrf->voice;
                             (auxtg+1)->voice = auxtg->voice;
                             auxtg->boundary = auxrf->boundary;
                             (auxtg+1)->boundary = auxrf->phoneme;
                             auxtg->forbid = auxrf->forbid;
                             (auxtg+1)->forbid = auxrf->forbid;
                             auxtg->prev = (auxtg->rank == 0) ? NULL : (auxtg-1);
                             (auxtg+1)->prev = auxtg;
                             auxtg->next = auxtg+1;
                             (auxtg+1)->next = auxtg+2;

                             total += 2;
                             break;

                 default :   auxtg->nsamp = offset;                  /* copy */
                             offset += duration;
                             auxtg->rank = rank++;
                             auxtg->phoneme = auxrf->phoneme;
                             auxtg->voice = auxrf->voice;
                             if ((i!=0)&&((auxmap-1)->repeat==-1))
                              { forgot = (auxrf-1)->boundary;
                                auxtg->boundary=MAX(auxrf->boundary,forgot);}
                             else auxtg->boundary = auxrf->boundary;
                             auxtg->forbid = auxrf->forbid;
                             auxtg->prev = (auxtg->rank == 0)?NULL:(auxtg-1);
                             auxtg->next = auxtg+1;

                             total++;
                             break;
               }
         }
     }


     if (npm>1)                                              /* last mark */
       {
         auxtg = auxtg + 1;
         auxtg->nsamp = offset;
         auxtg->rank = rank;
         auxtg->phoneme = (auxtg-1)->phoneme;
         auxtg->voice = (auxtg-1)->voice;
         auxtg->boundary = 1;
         auxtg->forbid = (auxtg-1)->forbid;
         auxtg->prev = auxtg-1;
         auxtg->next = NULL;

         total++;
     }

   S->ntgpm[unit] = total;                    /* number of marks & samples */
   S->nsamp[unit] = S->tar[unit][total-1].nsamp + 1;

   if ((S->nsamp[unit] < S->srate / 100)&&(sch.tiny == 0))
   {
     S->nsamp[unit] = 0;                          /* drop units under 10 msec */
```

```
      S->ntgpm[unit] = 0;
      S->nrfpm[unit] = 0;
    }

  }

}


/*********************************************************** Build a New Frame **/
/*   Build the target signal :                                                  */
/*       Depending on the scheme 'sch' employed, this builds the rankth         */
/*       target unit, from the reference unit, target and reference marks,      */
/*       and mapping, all of which are stored in S. A little messy, maybe.      */
/*******************************************************************************/


void xps_MAKE_UNIT(Synthesis *S, int rank, Scheme sch)

{
   int i, j, nsub, rend, tend, size, offset, memsiz, auxsiz, flag, nref, ntar;
   Marks *ref, *tar;
   Map *m;
   short * buffer;

   if (S->ntgpm[rank] > 1) {

   auxsiz = S->ref[rank][S->nrfpm[rank]-1].nsamp - S->ref[rank][0].nsamp;
   memsiz = MAX(S->nsamp[rank], auxsiz) + S->srate/50;
   buffer = xalloc(memsiz, short);       /* overlap due to psola is possible */
   bzero(buffer, memsiz*sizeof(short));
   nsub = S->nsubu[rank];
   ref = S->ref[rank];
   tar = S->tar[rank];
   m = S->map[rank];
   nref = S->nrfpm[rank];
   ntar = S->ntgpm[rank];
   rend = (tend = 1);
   offset = 0;
   flag = ref[0].voice;

   for (i=0; i<nsub; i++)
    {
      while ((rend<nref)&&(ref[rend].voice == flag)&&
            (ref[rend].boundary == 0))                  ++rend;
      while ((tend<ntar)&&(tar[tend].voice == flag)&&
            (tar[tend].boundary == 0))                  ++tend;

      if (ref->phoneme == 8)                                    /* a pause */
        xps_NP(S->wl[rank], ref, tar, 0, buffer+offset);
      else if (flag == 0)                                /* unvoiced phoneme */
             xps_NP(S->wl[rank], ref, tar, 1, buffer+offset);
           else if ((sch.P_method == 2)&&(sch.D_method == 3))  /* all psola */
                  xps_all_PSOLA(S->wl[rank], ref, tar, m, buffer+offset);
                else                                          /* default */
                  xps_MAKE_SUB_UNIT(S->wl[rank],ref,tar,m,sch,buffer);


      offset += tar[MIN(ntar-1,tend)].nsamp - tar->nsamp;
      if (rend < nref)
        flag = ref[rend].voice;

      ref = ref + rend;                                 /* updating marks */
      m = m + rend;
      tar = tar + tend;
      rend = (tend = 0 );
```



```
    }

   xps_strcopy(S->wl[rank], buffer, offset, 1);          /* new wave ready */

   xfree(buffer);

  }

}


/*********************************************************** Build Target Sub-Unit */
/*  Build a target sub-unit :                                                     */
/*     This routine is used for voiced phonemes, in case either the duration */
/*     or pitch modification algorithms chosen are not PSOLA. This routine   */
/*     is longer and messier than the others.                                    */
/*******************************************************************************/


void xps_MAKE_SUB_UNIT(short *wave, Marks *ref, Marks *tar, Map *m, Scheme sch, short
 *buffer)

{
   int i, j, nrpm, ntpm, type, offset, shift, flag, size, totsiz;
   short *aux1;
   Marks *rpm2;


   nrpm = (ntpm = 1);
   while ((ref[nrpm].next != NULL)&&(ref[nrpm].voice == 1)&&
         (ref[nrpm].boundary == 0))                          nrpm++;
   while ((tar[ntpm].next != NULL)&&(tar[ntpm].voice == 1)&&
         (tar[ntpm].boundary == 0))                          ntpm++;
   nrpm++;                                    /* we stop at boundaries, or */
   ntpm++;                                      /* one past voicing change */
   offset = 0;

   totsiz = tar[ntpm-1].nsamp - tar[0].nsamp + 1;
   aux1 = xalloc(2*totsiz, short);

  switch (sch.D_method)                            /* duration modification */
   {
     case 1: for (i=0; i<(nrpm-1); i++)                              /* use DM1 */
              {
                type = m[i].repeat;
                xps_DM1_dumb(wave, ref+i, type,aux1 +offset);
                offset += (type + 1) * (ref[i+1].nsamp - ref[i].nsamp);
              }
             break;

     case 2 : for (i=0; i<(nrpm-1); i++)                             /* use DM2 */
              {
                type = m[i].repeat;
                xps_DM2_select(wave, ref+i, type, aux1+offset);
                offset += (type + 1) * (ref[i+1].nsamp - ref[i].nsamp);
              }
             break;

     default : for (i=0; i<(nrpm-1); i++)                            /* use DM3 */
              {
                type = m[i].repeat;
                xps_DM3_psola(wave, ref+i, type, aux1+offset);
                offset += (type + 1) * (ref[i+1].nsamp - ref[i].nsamp);
              }
             break;
    }
```

```
offset = ref->nsamp;

rpm2 = xalloc(ntpm, Marks);                              /* auxiliary set */
rpm2[0].nsamp = offset;
rpm2[0].prev = NULL;
rpm2[0].next = rpm2 + 1;
rpm2[0].rank = 0;
rpm2[0].boundary = 0;                                    /* never mind */
for (i=0, j=1; i<nrpm-1; i++)
 {
   type = m[i].repeat;

   switch (type)
    {
      case 0: size = ref[i+1].nsamp - ref[i].nsamp;              /* copy */
              (rpm2+j)->prev = rpm2 +j -1;
              (rpm2+j)->next = (j==ntpm-1) ? NULL : rpm2 +j +1;
              (rpm2+j)->rank = j;
              (rpm2+j)->boundary = (i<nrpm-2)&&(m[i+1].repeat==-1)  ?
                                   i + 2 : i + 1;       /* never mind */
              (rpm2+j)->nsamp = rpm2[j-1].nsamp + size;
              j++;
              break;

      case 1: size = ref[i+1].nsamp - ref[i].nsamp;        /* duplicate */
              (rpm2+j)->prev = rpm2 +j -1;
              (rpm2+j)->next = rpm2 +j +1;
              (rpm2+j)->rank = j;
              (rpm2+j)->boundary = i + 1;    /* never mind */
              (rpm2+j)->nsamp = (rpm2+j-1)->nsamp + size;
              j++;
              (rpm2+j)->prev = rpm2 +j -1;
              (rpm2+j)->next = rpm2 +j +1;
              (rpm2+j)->rank = j;
              (rpm2+j)->boundary = i + 1;    /* never mind */
              (rpm2+j)->nsamp = (rpm2+j-1)->nsamp + size;
              j++;
              break;

      default : break;                                     /* elimination */
    }
 }

offset = 0;

switch (sch.P_method)                                /* pitch modification */
{
  case 1: for (j=0; j<(ntpm-1); j++)                          /* use PM1 */
            {
             i = rpm2[j].boundary;                        /* momentary use */
             shift = tar[j].nsamp;
             xps_PM1_intrapol(aux1+offset, rpm2+j, m+i, buffer+shift);
             offset += ref[i+1].nsamp - ref[i].nsamp;
            }
          break;

  case 2: bzero(buffer, totsiz*sizeof(short));                /* use PM2 */
          shift = tar[j].nsamp;
          for (i=0, j=0, flag=0; j<(ntpm-1); j++)
           {
             i = rpm2[j].boundary;                        /* momentary use */
             xps_PM2_psola(aux1, rpm2+j, tar+j, shift, buffer);
             shift = tar[j].nsamp;
           }
          break;
```



```
   }

   xfree(rpm2);
   xfree(aux1);

}


/****************************************** 1st Pitch Modification Method */
/*   Pitch modification by piece-wise linear intrapolation :              */
/*      This deducts a new waveform from the original, by intrapolation,  */
/*      assuming that the signal is linear between two consecutive samples. */
/*      Since this asumption is wrong, this introduces a little distortion. */
/*****************************************************************************/


void xps_PM1_intrapol(short *wave, Marks *ref, Map *p, short *buffer)

{
    int i, index, newsiz, size;
    float alpha, k;


    k = p->P_modif;
    size = ref->next->nsamp - ref->nsamp;
    newsiz = (int) (size / k);

    if (k != 1)
    {
        if ((wave == NULL) || ((wave+size-1) == NULL)|| (newsiz > BUF_SIZ))
        {
            P_Error("\n PM1_intrapol : wrong waveform size %d", size);
            list_error(On_Error_Tag);
        }

        if (buffer+MAX(size, newsiz) == NULL)
        {
            P_Error("\n PM1_intrapol : too short a buffer %d", newsiz);
            list_error(On_Error_Tag);
        }

        for (i=0; i<newsiz; i++) /* intrapolation */
        {
            index = (int) i * k;
            alpha = i * k - index;
            buffer[i] = (short)( alpha * wave[MIN(index+1, (size-1))]
                                 + (1-alpha) * wave[index]);
        }
    }
    else
        xps_strcopy(buffer, wave, size, 1);

}


/****************************************** 2nd Pitch Modification Method */
/*   Moulines'pitch modification :                                        */
/*      This deducts a new waveform from the original using short term sig- */
/*      nals set accordingly to the target pitch mark set ; check his Phd   */
/*      thesis for details.                                               */
/*****************************************************************************/


void xps_PM2_psola(short *in, Marks *ref, Marks *tar, int shift, short *out)
```

```
{
   int i, size, start, left, rigth;
   short *st;

   start = (tar->prev == NULL) ? 0 : tar->prev->nsamp;
   left = (ref->prev == NULL) ? ref->nsamp : ref->prev->nsamp;
   rigth = (ref->next == NULL) ? ref->nsamp : ref->next->nsamp;
   size = rigth - left + 1;
   st = xps_ST_Signal(in, ref, 1);

   for (i=0; i<size; i++)

     out[start - shift + i] += st[i];

   xfree(st);

}



/************************************ Dumb Duration Modification Method ***/
/*   Psola-less duration modification :                                   */
/*      This performs a simple smoothing of the discontinuities that are  */
/*      likely to appear in either elimination or suppression, using a    */
/*      psola-less method.  This is performed  after the modification.    */
/*************************************************************************/


void xps_DM1_dumb(short *wave, Marks *ref, int type, short *buffer)

{
   int i, span1=15, span2=5, size;
   float x, mean=0;


   if (type == 1)                                          /* duplication */
     {
       size = ref->next->nsamp - ref->nsamp;

       for (i=ref->nsamp; i<ref->next->nsamp; i++)            /* copying */
         {
           buffer[i-ref->nsamp] = wave[i];
           buffer[i-ref->nsamp+size] = wave[i];
         }

       for (i=-span1; i<=span2; i++)
         {
           x = (i<0) ?                                  /* smooth averaging */
                     xps_Sygmoid((float)(i+span1)/(2*span1)) :
                     xps_Sygmoid((float)(i+span2)/(2*span2));

           buffer[size+i] = (i<0) ?
             (short)((1-x) * wave[ref->next->nsamp+i] + x*wave[ref->nsamp+i]) :
             (short)(x* wave[ref->nsamp+i] + (1-x)* wave[ref->next->nsamp+i]);

         }
     }


   if (type == -1)                                         /* elimination */
     {
       size = ref->nsamp - ref->prev->nsamp;

       for (i=ref->prev->nsamp; i<ref->nsamp; i++)            /* copying */

           buffer[i-ref->prev->nsamp] = wave[i];
```



```
       for (i=ref->next->nsamp; i<ref->next->next->nsamp; i++)

           buffer[i-ref->next->nsamp+size] = wave[i];


       for (i= -span1; i<span2; i++)

           mean += (i<0) ?  *(wave+ref->nsamp+i) : 3*  *(wave+ref->nsamp+i);

       mean = mean / (6*span2);                          /* weigthed average */

       for (i= -span1; i<span2; i++)
         {
           x = (i<0) ?
                        xps_Sygmoid((float)(span1+i)/(2*span1)) :
                        xps_Sygmoid((float)(i+span2)/(2*span2));

           buffer[size+i] = (i<0) ?
             (short)((1-x)* wave[ref->nsamp+i] + x* wave[ref->next->nsamp+i]) :
             (short)( x* wave[ref->next->nsamp+i] + (1-x)*wave[ref->nsamp+i]);

         }
     }


   if (type == 0)                                            /* plain copying */
     {
       size = ref->next->nsamp - ref->nsamp;

       for (i=0; i<size; i++)

         buffer[i] = wave[ref->nsamp + i];
     }


}



/********************************* 2nd Dumb Duration Modification Method ***/
/*   Psola-less duration modification :                                    */
/*     This tries to minimize the discontinuities created : junctions are  */
/*     made simply by selecting a suitable (regarding slope, sample value) */
/*     pair of points near  the concerned pitch marks. It returns the rel- */
/*     ative position of the cut point. Done before the modification.      */
/*************************************************************************/


int xps_DM2_select(short *wave, Marks *ref, int type, short *buffer)

{
   int i, j, k, size, toto;
   float diff, slope, d, s;


   j=(k=0);
   s=(d=5000);
   toto = (type == -1) ? MIN(50, ref->nsamp-ref->prev->nsamp) :
                         MIN(50, ref->next->nsamp-ref->nsamp);

   if (type != 0)
   {
   while ((j<toto)||(d--5000))   {

        diff = Abs(*(wave+ref->nsamp-j) - *(wave+(ref->next)->nsamp-j)) +1;
```

```
        slope = Abs(*(wave+ref->nsamp-j-1) - *(wave+ref->nsamp-j+1) -
          *(wave+(ref->next)->nsamp-j-1) + *(wave+(ref->next)->nsamp-j+1)) +1;

        if ( ((2*diff+slope)*(1+j/toto)*(1+j/toto)) <
                                    ((2*d+s)*(1+k/toto)*(1+k/toto)))

                                        /* a better pair of points is found */
          {
            k=j;
            d=diff;
            s=slope;
          }

        j++;                                    /* left of pitch mark only */

       }
  }

   if (type == 1)                                           /* duplication */
     {
       size = ref->next->nsamp - k - ref->nsamp;

       for (i=ref->nsamp; i<(ref->next->nsamp - k); i++)

         buffer[i-ref->nsamp] = wave[i];

       for (i=(ref->nsamp - k); i<ref->next->nsamp; i++)

         buffer[i-ref->nsamp+k+size] = wave[i];
     }

   if (type == -1)                                          /* elimination */
     {
       size = ref->nsamp - k - ref->prev->nsamp;

       for (i=ref->prev->nsamp; i<(ref->nsamp - k); i++)

         buffer[i-ref->prev->nsamp] = wave[i];

       for (i=(ref->next->nsamp - k); i<ref->next->next->nsamp; i++)

         buffer[i-ref->next->nsamp+k+size] = wave[i];
     }

   if (type == 0)                                           /* plain copying */
     {

       size = ref->next->nsamp - ref->nsamp;

       for (i=0; i<size; i++)

         buffer[i] = wave[ref->nsamp + i];

       k = 0;
     }


   return(k);

}


/********************************** Standard Duration Modification Method */
/*   Psola-based duration modification :                                  */
/*      This employs short term signals, which are eliminated or duplicated */
/*      as required. Again, Moulines' thesis has all the details.        */
```



```
/*****************************************************************************/


void xps_DM3_psola(short *wave, Marks *ref, int type, short *buffer)

{
   int i, size1, size2, size3, size4;
   short *st1, *st2;


   if (type == 1)                                           /* duplication */
     {
        size1 = ref->nsamp - ref->prev->nsamp;
        size2 = ref->next->nsamp - ref->nsamp;
        st1 = xps_ST_Signal(wave, ref, 1);

        for (i=0; i<=size2; i++)                              /* 1st frame */

           buffer[i] = *(st1 + size1 + i);

        for (i=size2; i>MAX(0,size2-size1); i--)              /* 2nd frame */

           buffer[i] += *(st1 + i +size1 - size2);

        for (i=size2; i<(ref->next->next->nsamp - size2); i++)/* 2nd period */

           buffer[i] = *(wave + ref->nsamp + i);

        xfree(st1);
      }

   if (type == -1)                                          /* elimination */
     {
        size1 = ref->nsamp - ref->prev->nsamp;
        size2 = ref->next->nsamp - ref->nsamp;
        size3 = ref->next->next->nsamp - ref->next->nsamp;
        size4 = ref->prev->nsamp - ref->prev->prev->nsamp;
        st1 = xps_ST_Signal(wave, ref->next, 1);
        st2 = xps_ST_Signal(wave, ref->prev, 1);

        for (i=0; i<=size1; i++)

            buffer[i] = *(st2 + size4 + i);                   /* 1st frame */

        for (i=size1; i>=MAX(0,size1-size2); i--)

            buffer[i] += *(st1 + i + size2 - size1);          /* 2nd frame */

        for (i=size1; i<=(size1 + size3); i++)                /* 2nd period */

           buffer[i] = *(wave + ref->prev->nsamp + i + size2);


        xfree(st1);
        xfree(st2);

     }

   if (type == 0)                                            /* copying */
     {

      size2 = (ref->next != NULL) ? (ref->next->nsamp - ref->nsamp) : 0;

      for (i=0; i<size2; i++)

        buffer[i] = wave[ref->nsamp + i];
```

```
    }

}


/*************************************************************** All PSOLA */
/* Pure PSOLA :                                                            */
/*    This applies the psola algorithm to a given sub-unit ; results should */
/*    be the same using PM2 and DM3, but this is about twice as fast, and   */
/*    therefore is used whenever possible.                                  */
/*************************************************************************/


void xps_all_PSOLA(short *wave, Marks *ref,Marks *tar, Map *map, short *buffer)

{
   int middle, i, j, k, npm, rnpm, left, rigth, size, offset;
   int index, begin, thingie;
   short *st;

   rnpm = (npm = 1);
   while ((ref[rnpm].next != NULL)&&(ref[rnpm].voice == 1)&&
         (ref[rnpm].boundary == 0))                              rnpm++;
   while ((tar[npm].next != NULL)&&(tar[npm].voice == 1)&&
         (tar[npm].boundary == 0))                               npm++;
   rnpm++;                                    /* we stop at boundaries, or */
   npm++;                                       /* one past voicing change */

   offset = tar[0].nsamp;
   size = tar[npm-1].nsamp - offset + 1;
   bzero(buffer, size*sizeof(short));

   for (i=0, j=0, index=0, begin=0; i<rnpm; i++)
    {
      left = (ref[i].prev == NULL) ? 0 : ref[i].nsamp - ref[i].prev->nsamp;
      rigth = (ref[i].next == NULL) ? 0 : ref[i].next->nsamp - ref[i].nsamp;
      st = xps_ST_Signal(wave, ref+i, 1);

      if (ref[i].rank==0)  xps_strcopy(buffer,st+left,rigth+1,1); /* edges */
      else if (i == rnpm-1)
              {
                thingie = (tar[npm-1].nsamp - tar[npm-2].nsamp) -
                          (ref[rnpm-1].nsamp - ref[rnpm-2].nsamp);
                for (k=0; k<left+1; k++) buffer[begin+thingie+k] += st[k];
              }
           else if (map[i].repeat == 0)                           /* copy */
                 {
                   if (i==0)
                       {
                         begin = (tar-1)->nsamp - tar[0].nsamp; j = -1;
                         for (k=0; k<rigth+1; k++) buffer[k] += st[left+k];
                       }
                   else if (begin >= 0)
                         {
                           thingie = (tar[j+1].nsamp - tar[j].nsamp) -
                                     (ref[i].nsamp - ref[i-1].nsamp);
                           for (k=MAX(0,-(begin+thingie)); k<left+rigth+1; k++)
                                buffer[begin+thingie+k] += st[k];
                         }
                      else for (k=0; k<rigth+1; k++) buffer[k] += st[left+k];
                   begin += (tar + ++j)->nsamp -  (tar+j-1)->nsamp;
                 }
                 else if (map[i].repeat == 1)                  /* duplicate */
                       {
                         if (i--0)
```



```
                          {
                            for (k=0; k<rigth+1; k++)
                                     buffer[k] += st[left+k];
                            thingie = (tar[1].nsamp - tar[0].nsamp) -
                                      (ref[0].nsamp - (ref-1)->nsamp);
                            for (k=MAX(0, -thingie); k<left+rigth+1; k++)
                                     buffer[thingie+k] += st[k];
                            begin = tar[1].nsamp - tar[0].nsamp;
                            j = 1;
                          }

                        else
                         {
                           thingie = (tar[j+1].nsamp - tar[j].nsamp) -
                                     (ref[i].nsamp - ref[i-1].nsamp);
                           for (k=MAX(0,-(begin+thingie));k<left+rigth+1; k++)
                                    buffer[begin+thingie+k] += st[k];
                           begin += (tar + ++j)->nsamp -  (tar+j-1)->nsamp;
                           thingie = (tar[j+1].nsamp - tar[j].nsamp) -
                                     (ref[i].nsamp - ref[i-1].nsamp);
                           for (k=0; k<left+rigth+1; k++)
                                    buffer[begin+thingie+k] += st[k];
                           begin += (tar + ++j)->nsamp -  (tar+j-1)->nsamp;
                         }
                       }
                     else if (i--0)                              /* eliminate */
                          { begin = (tar-1)->nsamp - tar[0].nsamp; j = -1;};

      xfree(st);

   }

}


/*********************************************************** Noisy Processing */
/*  Prosodic modifications for unvoiced phonemes :                         */
/*    This applies to the "noisy" phonemes as defined by the macro-function */
/*    NOISY. It makes prosodic modifications by cutting/duplicating samples */
/*    in the middle of the sub-unit. Special treatment for pauses (type 0). */
/*************************************************************************/


void xps_NP(short * wave, Marks *ref, Marks *tar, int type, short *buffer)

{
   int middle, i, npm, rnpm, size1, size2, size3, aux, aux2, shift;


   rnpm = (npm = 1);
   while ((ref[rnpm].next != NULL)&&(ref[rnpm].voice == 0)&&
         (ref[rnpm].boundary == 0))                              rnpm++;
   while ((tar[npm].next != NULL)&&(tar[npm].voice == 0)&&
         (tar[npm].boundary == 0))                               npm++;
   rnpm++;                                    /* we stop at boundaries, or */
   npm++;                                       /* one past voicing change */

   if (type == 0)
    {                                             /* silence, stuffed with 0 */
      middle = (rnpm+1)/2;
      size1 = (rnpm == 2) ? (ref[1].nsamp - ref[0].nsamp) / 2 :
                            ref[middle].nsamp - ref[0].nsamp;
      size2 = (rnpm == 2) ? (ref[1].nsamp - ref[0].nsamp +1) / 2 :
                            ref[rnpm-1].nsamp - ref[middle].nsamp;
      size3 = (npm == 2) ? tar[1].nsamp - tar[0].nsamp :
                           tar[npm-1].nsamp - tar[0].nsamp;
```

```
      xps_strcopy(buffer, wave+ref[0].nsamp, MAX(size1, size2)+1, 1);
      aux = size3-size2-size1;
      if (aux >= 0) {
        bzero(buffer+size1,  MIN(1,aux*sizeof(short)));
        shift = (rnpm == 2) ? ref[0].nsamp + size1 : ref[middle].nsamp;
        xps_strcopy(buffer+size1+aux, wave+shift, size2+1, 1);
      }  else
          xps_strcopy(buffer+size3/2, wave+ref[rnpm-1].nsamp -
                                  (size3-size3/2), size3 - size3/2+1, 1);

   }
   else                                  /* for consonants, we cut the middle */
    {                                              /* samples when reducing the */
      size1 = ref[rnpm-1].nsamp - ref[0].nsamp; /* sub-unit, and repeat til */
      size2 = tar[npm-1].nsamp - tar[0].nsamp;  /* necessary the middle 20% */
      size3 = size2 - size1;                   /* samples when expanding it */
      aux = size2/2;
      if (size3<(0.2*size1))                            /* one run is enough */
       {
         xps_strcopy(buffer, wave+ref[0].nsamp, aux, 1);
         xps_strcopy(buffer+aux, wave+ref[0].nsamp+size1-(size2 - aux),
                                                  (size2 - aux +1), 1);
       }
      else
       {                                      /* more than one run is needed */
         aux2 = size1 / 5;
         xps_strcopy(buffer, wave+ref[0].nsamp, 3*aux2, 1);
         shift = 2*aux2;
         while (size3>0)
          {
            aux = MIN(aux2, size3);
            xps_strcopy(buffer+shift, wave+ref[0].nsamp+2*aux2, aux, 1);
            size3 -= aux;
            shift += aux;
          }
         xps_strcopy(buffer+shift, wave+ref[rnpm-1].nsamp-2*aux2-
                              (size1-5*aux2), 2*aux2+(size1-5*aux2)+1, 1);

       }
    }
}



/******************************************************* Unit Concatenation **/
/*   Concatenate consecutive units :                                         */
/*      This will, according to the 'method' used, concatenates the unit of  */
/*      a given rank with the following, from 'S' and into 'pw'. If needed,   */
/*      field 'nsamp' (number of samples) in 'S' will be updated. Methods     */
/*      available are : dumb, point selection, psola, cepstral distance and   */
/*      frame extrapolation (not ready yet).                                  */
/*****************************************************************************/


int xps_CONCAT(Synthesis *S, int rank, P_Wave pw, int index, int method)

{
   short *st1, *st2, *current, *lw, *rw, buffer[4*BUF_SIZ], *p;
   int i, j, k, l, size1, size2, size3, new, max, min, prev;
   int x, aux1, aux2, span1, span2, isp1, isp2, isp3, shift1, shift2;
   Marks *left, *rigth;
   Joint joint;
   float scale1, scale2, power, alpha, average, rms1, rms2, rms3;
   float *fft1, *fft2, *fft3, **distance, mini, aux;
```



```
p = pw->wave + index;

if (rank == 0)
{
    for (i=0; i<S->nsamp[0]; i++)                              /* first unit*/

                p[i] = S->wl[0][i];

    new = S->nsamp[0];
    index = new;
    p = p + index;
}

if (S->ntgpm[rank+1] < 2) return index;

if (rank <(S->nunits - 1))
  {
new = index;
prev = rank;
while ((prev>0)&&(S->nsamp[prev] == 0)) prev--;

switch (method)
  {

    case 1 :  for (i=0; i<S->nsamp[rank+1]; i++)                  /* DUMB */
                {
                  p[i] = S->wl[rank+1][i];              /* plain copying */
                }

              pw->num_samp += 0;
              new = index + S->nsamp[rank+1];
              return new;
              break;

    case 2 :  left = (S->ntgpm[prev] != 0) ?
                                S->tar[prev]+S->ntgpm[prev]-1 : NULL;
              if (left == NULL)
              {
                for (i=0; i<S->nsamp[rank+1]; i++)

                   p[i] = S->wl[rank+1][i];              /* plain copying */

                pw->num_samp += 0;
                new = index + S->nsamp[rank+1];
                return new;
              }
              else
              {
                rigth = S->tar[rank+1];
                lw = S->wl[prev];
                rw = S->wl[rank+1];
                span1 = S->nsamp[prev]-1;
                shift1 = left->nsamp - left->prev->nsamp;
                shift2 = rigth->next->nsamp - rigth->nsamp;
                shift1 = MIN(shift1 / 2, S->srate / 200);        /* 5 msec */
                shift2 = MIN(shift2 / 2, S->srate / 200);
                distance = xalloc(shift1, float *);
                for (i=0; i<shift1; i++)
                  distance[i] = xalloc(shift2, float);

                for (i=0; i<shift1; i++)
                  for (j=0; j<shift2; j++)

                    distance[i][j] =(lw[span1-i]-rw[j])*(lw[span1-i]-rw[j]);
```

```
                mini = 100000000;
                aux1 = (aux2 = 0);
                for (i=1; i<shift1-1; i++)
                  for (j=1; j<shift2-1; j++)
                  {
                     aux = distance[i][j] + distance[i+1][j-1] +
                                                  distance[i-1][j+1];
                     if (aux < mini)
                     {
                       aux1 = i;
                       aux2 = j;
                       mini = aux;
                     }
                  }

                p = p - i;

                for (i=aux2; i<S->nsamp[rank+1]; i++)
                  {
                    p[i-aux2] = S->wl[rank+1][i];              /* copying */
                  }

                pw->num_samp -= (aux1 + aux2);
                new += S->nsamp[rank+1] - (aux1 + aux2);
                return new;
                }
                break;


        case 3 :  left = (S->ntgpm[prev] != 0) ?                    /* PSOLA */
                         S->tar[prev] + S->ntgpm[prev] - 2 : NULL;
                if (left == NULL)
                 {
                    for (i=0; i<S->nsamp[rank+1]; i++)
                     {
                       p[i] = S->wl[rank+1][i];           /* plain copying */
                     }

                    pw->num_samp += 0;
                    new = index + S->nsamp[rank+1];
                    return new;
                }
                else
                if (left->voice *S->tar[rank+1][0].voice == 0) /* unvoiced */
                {
                  for (i=0; i<S->nsamp[rank+1]; i++)              /* DUMB */

                     p[i] = S->wl[rank+1][i];             /* plain copying */

                  pw->num_samp += 0;
                  new = index + S->nsamp[rank+1];
                  return new;
                 }
                else
                {
                rigth = S->tar[rank+1] + 1;
                size1 = left->next->nsamp - left->nsamp;
                size2 = rigth->nsamp - rigth->prev->nsamp;
                size3 = (size1 + size2) / 2;
                aux1 = left->nsamp - left->prev->nsamp +1;
                lw = S->wl[prev];
                rw = S->wl[rank+1];
                st1 = xps_ST_Signal(lw, left, 1);
                st2 = xps_ST_Signal(rw, rigth, 1);
                bzero(&buffer[0], BUF_SIZ*sizeof(short));
```



```
                for (i=0; i<size1; i++)

                  buffer[i] = st1[aux1+i];                    /* 1st frame */

                for (i=size3-1; i>=MAX(0, (size3-size2)); i--)

                  buffer[i] += st2[i+size2-size3];          /* 2nd frame */

                p = p - size1;

                for (i=0; i<size3; i++)
                  {
                    p[i] = buffer[i];                              /* copying */
                                                      /* the intersection */
                  }


                for (i=size3; i<S->nsamp[rank+1]; i++)

                    p[i] = S->wl[rank+1][i];                       /* copying */
                                                   /* the rest of the unit */
p[size3] = 10000;

                new += size3 - size2 - size1 + S->nsamp[rank+1];

                pw->num_samp += size3 - size2 - size1;

                xfree(st1);
                xfree(st2);

                return new;
                }

                break;


        case 4 :                                        /* cepstral distances */

                joint = xps_Cut_Point( S->wl[prev], S->wl[rank+1],
                     S->tar[prev]+S->ntgpm[prev]-1, S->tar[rank+1]);

                left = S->tar[prev]+S->ntgpm[prev]-joint.nfra1-2;
                rigth = S->tar[rank+1]+joint.nfra2;
                shift1 = S->tar[prev][S->ntgpm[prev]-1].nsamp -
                                                    left->next->nsamp;
                shift2 = rigth->nsamp;
                st1 = xps_ST_Signal(S->wl[prev], left, 1);
                st2 = xps_ST_Signal(S->wl[rank+1],rigth, 1);

                size1 = left->nsamp - left->prev->nsamp;
                i = left->next->nsamp - left->nsamp;
                j = rigth->next->nsamp - rigth->nsamp;
                k = (i + j) / 2;

                bzero(&buffer[0], sizeof(short)*BUF_SIZ);

                for (l=0; l<MIN(i,k); l++)                          /* psola */
                   buffer[l] = st1[size1+l];
                for (l=k; l>MAX(0, k-j); l--)
                   buffer[l] += st2[l];

                i = (S->tar[prev]+S->ntgpm[prev]-1)->nsamp - left->nsamp;
                j = rigth->nsamp - (S->tar[rank+1])->nsamp;

                p = p - shift1;
```

```
                for (l=0; l<k; l++)                    /* junction frame */
                  p[l] = buffer[l];

                p = p + k;

                for (l=shift2; l<S->ntgpm[rank+1]; l++)       /* next unit */
                  p[l-shift2] = S->wl[rank+1][l];

                pw->num_samp -= shift1 + shift2 - k;

                new += S->nsamp[rank+1] -shift1 -shift2 +k;

                xfree(st1);
                xfree(st2);

                return new;
                break;


        case 5 : left = S->tar[prev]+S->ntgpm[prev]-2;    /* extrapolation */
                 rigth = S->tar[rank+1];
                 size1 = left->next->nsamp - left->nsamp;
                 size2 = rigth->next->nsamp - rigth->nsamp;
                 size3 = (size1+size2)/2;
                 rms1 = (rms2 = (rms3 = 0));
                 bzero(&buffer[0], 3*BUF_SIZ*sizeof(short));

                 for (i=0; i<size1; i++)                  /* left period */
                  {
                    buffer[BUF_SIZ+i] = S->wl[prev][left->nsamp+i];
                    rms1 += buffer[BUF_SIZ+i]*buffer[BUF_SIZ+i];
                  }
                 rms1 = sqrt(rms1/size1);                          /* rms */

                 for (i=0; i<size2; i++)                 /* rigth period */
                  {
                    buffer[2*BUF_SIZ+i] = S->wl[rank+1][i];
                    rms2 += buffer[2*BUF_SIZ+i]*buffer[2*BUF_SIZ+i];
                  }
                 rms2 = sqrt(rms2/size2);                          /* rms */

                 scale1 = (1.0*size1)/(1.0*size3);
                 scale2 = (1.0*size2)/(1.0*size3);


                 for (i=0; i<size3; i++)            /* temporal averaging */
                  {
                    buffer[i] =
              xps_Sygmoid((float)i/size3) * buffer[BUF_SIZ+(int)(i*scale1)] +
            (1-xps_Sygmoid((float)i/size3)) * buffer[2*BUF_SIZ+(int)(i*scale2)];

                    rms3 += buffer[i] * buffer[i];
                  }

                 rms3 = sqrt(rms3/size3);                  /* re-scaling */
                 power = (rms1 + rms2)/(2.0*rms3);

                 for (i=0; i<size3; i++)

                    p[i] = (short) (buffer[i] * power);


                 for (i=0; i<S->nsamp[rank+1]; i++)         /* rigth unit */
                   {
                     p[i+size3] = S->wl[rank+1][i];
```



```
                   }

                 span1 = 15;
                 span2 = 5;
                 average = 0;

                 for (i=-span1; i<=span2; i++)
                   average += (i<0) ? p[i] : 3*p[i];
                 average = average / (2*span1);

                 for (i=-span1; i<=span2; i++)         /* smooth averaging */
                   {
                      x = (i<0) ?  xps_Sygmoid((float)(i+span1)/(2*span1)) :
                                   xps_Sygmoid((float)(i+span2)/(2*span2));

                      p[i] = (i<0) ?  (short)(x*average + (1-x)*p[i]) :
                                      (short)((1-x)*p[i] + x*average);
                   }

                 p = p + size3;

                 average = 0;
                 for (i=-span1; i<=span2; i++)
                   average += (i<0) ?  p[i] : 3*p[i];
                   average = average / (2*span1);

                 for (i=-span1; i<=span2; i++)         /* smooth averaging */
                 {
                   x = (i<0) ?   xps_Sygmoid((float)(i+span1)/(2*span1)) :
                                 xps_Sygmoid((float)(i+span2)/(2*span2));

                   *(p-S->nsamp[rank+1]+i) = (i<0) ?
                                      (short)(x*average + (1-x)*p[i]) :
                                      (short)((1-x)*p[i] + x*average);
                  }

                 new += size3 + S->nsamp[rank+1];

                 pw->num_samp += size3;

                 return new;
                 break;


        case 6 : left = S->tar[prev]+S->ntgpm[prev]-2;    /* extrapolation */
                 rigth = S->tar[rank+1];
                 size1 = left->next->nsamp - left->nsamp;
                 size2 = rigth->next->nsamp - rigth->nsamp;
                 size3 = (size1+size2)/2;
                 rms1 = (rms2 = (rms3 = 0));
                 bzero(&buffer[0], 3*BUF_SIZ*sizeof(short));

                 for (i=0; i<size1; i++)                  /* left period */
                  {
                    buffer[BUF_SIZ+i] = S->wl[prev][left->nsamp+i];
                    rms1 += buffer[BUF_SIZ+i]*buffer[BUF_SIZ+i];
                  }
                 rms1 = sqrt(rms1/size1);                          /* rms */

                 for (i=0; i<size2; i++)                 /* rigth period */
                  {
                    buffer[2*BUF_SIZ+i] = S->wl[rank+1][i];
                    rms2 += buffer[2*BUF_SIZ+i]*buffer[2*BUF_SIZ+i];
                  }
                 rms2 = sqrt(rms2/size2);                          /* rms */
```

```
scale1 = (1.0*size1)/(1.0*size3);
scale2 = (1.0*size2)/(1.0*size3);

isp1 = xps_ispower2(size1);
isp2 = xps_ispower2(size2);
isp3 = xps_ispower2(size3);

fft1 = xalloc(isp1, float);                    /* frequency */
fft2 = xalloc(isp2, float);          /* domain */
fft3 = xalloc(isp3, float);
bzero(fft1, sizeof(float)*isp1);
bzero(fft2, sizeof(float)*isp2);
bzero(fft3, sizeof(float)*isp3);
xps_strcopy(fft1, &buffer[BUF_SIZ], size1, 2);
xps_strcopy(fft2, &buffer[2*BUF_SIZ], size2, 2);

FFT(fft1, isp1);
FFT(fft2, isp2);

for (i=0; i<size3; i++)             /* f-domain averaging */

   fft3[i] =
     (fft1[(int)(i*scale1)] + fft2[(int)(i*scale2)]) / 2;

iFFT(fft3, isp3);
for (i=0; i<size3; i++)
 {
   fft3[i] = fft3[i] / isp3;
   rms3 += fft3[i] * fft3[i];
 }

rms3 = sqrt(rms3/size3);                    /* re-scaling */
power = (rms1 + rms2)/(2.0*rms3);

for (i=0; i<size3; i++)

   p[i] = fft3[i] * power;


for (i=0; i<S->nsamp[rank+1]; i++)          /* rigth unit */
  {
    p[i+size3] = S->w1[rank+1][i];

  }

span1 = 15;
span2 = 5;
average = 0;

for (i=-span1; i<=span2; i++)
  average += (i<0) ? p[i] : 3*p[i];
average = average / (2*span1);

for (i=-span1; i<=span2; i++)         /* smooth averaging */
  {
     x = (i<0) ?  xps_Sygmoid((float)(i+span1)/(2*span1)) :
                  xps_Sygmoid((float)(i+span2)/(2*span2));



     p[i] = (i<0) ?  (short)(x*average + (1-x)*p[i]) :
                     (short)((1-x)*p[i] + x*average);

  }

p = p + size3;
```



```
average = 0;
for (i=-span1; i<=span2; i++)
  average += (i<0) ?  p[i] : 3*p[i];
  average = average / (2*span1);

for (i=-span1; i<=span2; i++)         /* smooth averaging */
{
  x = (i<0) ?    xps_Sygmoid((float)(i+span1)/(2*span1)) :
                 xps_Sygmoid((float)(i+span2)/(2*span2));


  *(p-S->nsamp[rank+1]+i) = (i<0) ?
                    (short)(x*average + (1-x)*p[i]) :
                    (short)((1-x)*p[i] + x*average);

 }

new += size3 + S->nsamp[rank+1];

pw->num_samp += size3;

xfree(fft1);
xfree(fft2);
xfree(fft3);

return new;
break;

 }

}

}
```

```
/*=========================================================================*/
/*           A T R Interpreting Telecommunications Labs                    */
/*                                                                         */
/*=========================================================================*/
/*                 CHATR Speech Synthesis System                           */
/*                      Christian Lelong                                   */
/*-------------------------------------------------------------------------*/
/*                                                                         */
/*                 PSOLA Related Definitions                               */
/*                                                                         */
/*                        Feb 1995                                         */
/*                  Copyrigth (C) 1994, 1995                               */
/*     ATR Interpreting Telecommunications Research Laboratories           */
/*                        All rights reserved.                             */
/*=========================================================================*/

#include <stdio.h>
#include <string.h>
#include <math.h>
#include "list.h"
#include "table.h"
#include "alloc.h"
#include "wave.h"
#include "udb.h"
#include "pmark.h"
#include "intonation.h"
#include "interface.h"
#include "phoneme.h"
#include "ph_unit.h"
#include "interface.h"

#define sq(x) ((x)*(x))
#define MIN(a,b) ((a) > (b) ? (b) : (a))
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define FFT(data, m) (xps_realft (data-1, m>>1, 1))
#define iFFT(data, m) (xps_realft (data-1, m>>1, -1))
#define fft(data, m) (xps_four1(data-1, m, 1))
#define ifft(data, m) (xps_four1(data-1, m, -1))
#define PI 3.141592653589793
#define Abs(a) ((a)>0 ? (a) : (-a))
#define swap(a,b,tempr) (tempr)=(a); (a)=(b); (b)=(tempr)
#define sign(a) ((a) > 0 ? (1) : (-1))

#define SAM_MAX  4096
#define MS_ASYN 10
#define MAX_LEN 3
#define MIN_LEN (udb_current->wave_sample_rate / 5)
#define BUF_SIZ (udb_current->wave_sample_rate / 5)


enum Unit_Type { vowel=1, liquid, nasal, affricative, fricative, closure, stop, silen
ce };

typedef struct complex
{
    float real;
    float imag;
} Complex;

typedef struct polar
{
    float scale;
    float phase;
} Polar;

typedef struct joint
```



```
{
   int nfral;                                   /* rank of left frame */
   int nfra2;                                  /* rank of rigth frame */
   int dur;                                     /* actual duration */
   int targ_dur;                                /* target duration */
   float distance;                             /* cepstral distance */
} Joint;

typedef struct scheme
{
   float D_thres;                /* threshold for duration modifications */
   float P_thres;                  /* threshold for pitch modifications */
   int D_method;                      /* duration modification method */
   int P_method;                       /* pitch modification method */
   float P_ceil;                 /* ceiling for pitch modifications */
   float D_ceil;               /* ceiling for duration modifications */
   int concat;                          /* concatenation processing */
   int power;                                   /* power processing */
   int contour;                                  /* pitch contour */
   int model;                                 /* signal modelling */
   int PERMAX;                          /* for larynx pitch marks */
   int test;                           /* to see what's happening */
   int stops;                                  /* prosody for stops */
   int tiny;                           /* ignore very short units */
   int voicing;                                /* voicing criteria */
}  Scheme;

typedef struct map
{
   int rank;                                    /* same as reference mark */
   int repeat;                         /* elimination / duplication */
   float D_modif;                          /* duration modification */
   float P_modif;                             /* pitch modification */
   float R_modif;                               /* rms modification */
} Map;

/* small_pm definition moved to udb.h */

typedef struct mark
{
   int nsamp;                        /* number of the sample it points to */
   int rank;                            /* number of current pitch mark */
   int phoneme;                                    /* type of phoneme */
   int boundary;                     /* signals beginning of new sub-unit */
   int forbid;                             /* for reference  marks only */
   float voice;                          /* 0 => unvoiced, 1 => voiced */
   struct mark *prev;                                 /* previous mark */
   struct mark *next;                                     /* next mark */
} Marks;

typedef struct  synthesis
{
   short **w1;                                  /* speech signal, per unit */
   short **w2;                        /* stochastic component, per unit */
   Marks **ref;                          /* corresponding pitch marks */
   Marks **tar;;                                /* target pitch marks */
   Map **map;                                    /* mapping, per unit */
   int *nsamp;                          /* number of samples, per unit */
   int *nrfpm;                           /* number of  marks, per unit */
   int *ntgpm;                       /* number of target  marks, per unit */
   int *nsubu;                          /* number of sub-units, per unit */
   int nunits;                                /* total number of units */
   int srate;                                        /* sampling rate */
}  Synthesis;
```

```
/*=================================================================*/
/*        A T R Interpreting Telecommunications Labs               */
/*                                                                 */
/*=================================================================*/
/*            CHATR Speech Synthesis System                        */
/*          Helene Valbret & Christian Lelong                      */
/*-----------------------------------------------------------------*/
/*                                                                 */
/*           Building complete wave from units                     */
/*                                                                 */
/*                                                                 */
/*                May 1994 - February 1995                         */
/*                                                                 */
/*             Copyright (C) 1993,1994,1995                        */
/* ATR Interpreting Telecommunications Research Laboratories       */
/*                   All rights reserved.                          */
/*=================================================================*/
/*                                                                 */
/*  Based on the Concat_Method choses a way to concatenate         */
/*  units.  Currently supports  :                                  */
/*                                                                 */
/*    - PSOLA / Cepstrum (nuutalk) / Dumb concatenation            */
/*                                                                 */
/*    - a range of options for power, pitch and duration           */
/*      modifications, for concatenation and also for the          */
/*      speech model used is available via the function            */
/*      psola_init_params(). Default is : PSOLA pitch &            */
/*      duration, no power modification, simple model,             */
/*      select-point concatenation.                                */
/*                                                                 */
/*  Note : PSOLA is programmed twice, and results differ.          */
/*  Of course there has to be a Unit stream for these to do        */
/*  anything                                                       */
/*                                                                 */
/*=================================================================*/

#include <time.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#include "xruc.h"

#include "alloc.h"
#include "list.h"
#include "table.h"
#include "chatr.h"
#include "interface.h"
#include "general.h"
#include "ph_unit.h"
#include "phoneme.h"
#include "wave.h"
#include "cep.h"
#include "grammar.h"
#include "pmark.h"
#include "udb.h"
#include "play.h"
#include "ruc.h"
#include "futils.h"

#include "../udb/nus.h"                    /* temporal, should be fixed */


#define FILENAME "rif.dat"                                 /* filters file */
```



```
FILE *fid;
extern int SPAN;
static Scheme sch;

static struct Wave *unit_dumbp_concat_module(Utterance utt);
static void get_pm_points(P_Marks pm,int *st_pm, int *ls_pm);
static void get_zcrossing(P_Wave w,int *fst, int *lst);
static void xpsola_init_params(void);

struct Wave *unit_concat_module(Utterance utt)
{
    /* Builds a wave forms from the unit stream information in a method */
    /* as defined by Concat_Method Parameter                            */
    /* Unit stream must be set up for this to work (and others things   */
    /* too depending on the type of concatenation                       */

    if (utt_stream("Unit",utt) == SNIL)
    {
        P_Error("Unit concat module called on utterance without unit stream");
        list_error(On_Error_Tag);
    }

    if (streq(ch_param.concat_method,"PSOLA"))
        return psola_concat_module(utt);
    else if (streq(ch_param.concat_method,"NUUCEP"))
        return nuucep_concat_module(utt);
    else if (streq(ch_param.concat_method,"DUMB"))
        return unit_dumb_concat_module(utt);
    else if (streq(ch_param.concat_method,"DUMB+"))
        return unit_dumbp_concat_module(utt);

    else if (streq(ch_param.concat_method,"XPSOLA"))
        return xpsola_concat_module(utt);

    else if (streq(ch_param.concat_method,"NULL"))
        return NULL;  /* sometimes useful for testing */
    else
    {
        P_Error("Unset or unknown concat method, using PSOLA by default");
        return psola_concat_module(utt);
    }
}

void utt_unit_concat_module(Utterance utt)
{
    /* Utt module function to do concatenation */
    struct Stream_cell *w_cell = new_stream_cell("Wave");
    free_pwave(SC(w_cell,Wave));  /* free the wave part */

    utt_stretch(utt);

    SC(w_cell,Wave) = unit_concat_module(utt);

    utt_set_stream("Wave",w_cell,utt);

    return;
}

struct Wave *psola_concat_module(Utterance utt)
{
    /* Build a whole wave form by simple concatenating the units */
    struct Wave *waveform;
    Ref_Wave *ref_wave;
    Ref_Marks *ref_pm;
```

```
    P_Marks targ_pm;
    Util_Ph *util_ph;
    Stream u;
    int nb_unit = 0;
    int i;
    int common_sr;
    int scheme = Simple;

    common_sr = udb_current->wave_sample_rate;

    for (nb_unit = 0, u = UNITSTREAM(utt); u != SNIL; u = SC_next(u))
        nb_unit++;

    /*********************************************/
    /** Reading Reference Wave and Pitch-Marking **/
    /*********************************************/
    ref_pm = Get_Ref_Pitch(utt, common_sr, nb_unit); /* modifies bounaries */
    ref_wave = Get_Ref_Wav(utt, common_sr, nb_unit);

    /*************************************************************/
    /** Creating Target Pitch Marks according to input prosody **/
    /*************************************************************/
    targ_pm =  Creat_Target_pm(utt, MS_ASYNC, common_sr);

    /*********************************/
    /** Reading Useful information **/
    /*********************************/
    util_ph = Extract_Util_Ph(utt, ref_pm, nb_unit, common_sr);

    /************************/
    /** Power Modification **/
    /** Use of Square Power**/
    /************************/
    Power_Modif(ref_wave, utt, PWR_SQR);

    /************************/
    /** TD-PSOLA Synthesis **/
    /************************/
    waveform = PMS(ref_wave, ref_pm, targ_pm, util_ph, scheme);

    /*********************/
    /** Memory Freeing **/
    /*********************/

    for (i = 0; l < ref_pm->nb_unit; i++)
        free_pm(ref_pm->unit_pm[i]);
    xfree(ref_pm->unit_pm);
    xfree(ref_pm->info_pm);

    free_pm(ref_pm->utt_pm);
    xfree(ref_pm);

    for (i = 0; i < ref_wave->nb_unit; i++)
         free_pwave(ref_wave->unit[i]);
    xfree(ref_wave->unit);
    xfree(ref_wave);

    free_pm(targ_pm);

    xfree(util_ph->sub);
    xfree(util_ph);

    return waveform;
}

struct Wave *unit_dumb_concat_module(Utterance utt)
```



```
{
    /* Build a whole wave form by simply concatenating the units */
    /* No PSOLA, just put them together                          */
    int i,j,k;
    Stream ucell;
    Ref_Wave *ref_wave;
    int num_units = 0;
    struct Wave *wholewave;
    short *waveform;
    int common_sr = 0;

    common_sr = udb_current->wave_sample_rate;
    for (ucell=UNITSTREAM(utt);ucell != SNIL; ucell = SC_next(ucell))
        num_units ++;

    ref_wave = Get_Ref_Wav(utt, common_sr, num_units);

    Power_Modif(ref_wave,utt,PWR_SQR);

    wholewave = make_wave();
    waveform = xalloc(ref_wave->num_samp,short);

    for (j=0,i=0; i< num_units; i++)
    {
        for (k=0; k<ref_wave->unit[i]->num_samp; k++)
            waveform[j++] = ref_wave->unit[i]->wave[k];
        free_pwave(ref_wave->unit[i]);
    }
    xfree(ref_wave->unit);
    xfree(ref_wave);

    wholewave->coding = CH_LIN16;   /* signed linear */
    wholewave->num_samp = ref_wave->num_samp;
    wholewave->samp_rate = ref_wave->samp_rate;
    wholewave->wave = waveform;

    return wholewave;
}

static struct Wave *unit_dumbp_concat_module(Utterance utt)
{
    /* Build a whole wave form by simply concatenating the units */
    /* Put the raw waves together at pitch mark boundaries       */
    int i,j,k;
    Stream u;
    Ref_Wave *ref_wave;
    int num_units, common_sr;
    int stp, edp;
    struct Wave *wholewave;
    Ref_Marks *ref_pm = NULL;
    short *waveform;

    common_sr = udb_current->wave_sample_rate;
    for (num_units=0,u=UNITSTREAM(utt); u != SNIL; u=SC_next(u))
        num_units ++;
    printf("Number of units %d\n",num_units);

    /** we read the pitch marks to cut in a nice place **/
    /** Note this *modifies* the start and end point to pitch marks **/
    if (list_str_eval("concat_dumb_pm",NULL))
        ref_pm = Get_Ref_Pitch(utt, common_sr, num_units);
    ref_wave = Get_Ref_Wav(utt, common_sr, num_units);
    Power_Modif(ref_wave,utt,PWR_SQR);

    wholewave = make_wave();
    waveform = xalloc(ref_wave->num_samp,short);
```

```
    for (j=0,i=0; i< num_units; i++)
    {
        stp = 0;
        edp = ref_wave->unit[i]->num_samp;
        if (list_str_eval("concat_dumb_zcs",NULL))
            get_zcrossing(ref_wave->unit[i],&stp,&edp);
        for (k=stp; k<edp; k++)
            waveform[j++] = ref_wave->unit[i]->wave[k];
        free_pwave(ref_wave->unit[i]);
    }
    wholewave->num_samp = j;

    wholewave->coding = CH_LIN16;   /* signed linear */
    wholewave->samp_rate = ref_wave->samp_rate;
    wholewave->wave = waveform;

    xfree(ref_wave->unit);
    xfree(ref_wave);

    if (ref_pm != NULL)
    {
        for (i = 0; i < ref_pm->nb_unit; i++)
            free_pm(ref_pm->unit_pm[i]);
        xfree(ref_pm->unit_pm);
        xfree(ref_pm->info_pm);
        free_pm(ref_pm->utt_pm);
        xfree(ref_pm);
    }

    return wholewave;
}

static void get_zcrossing(P_Wave w,int *fst, int *lst)
{   /* Find first and last zero crossing */
    int i;

    for (i=1; i < w->num_samp; i++)
        if (w->wave[0] < 0)
        {
            if (w->wave[i] > 0)
                break;
        }
        else if (w->wave[i] < 0)
            break;
    if (i < 64)
        *fst = i;
    else
        *fst = 0;
    for (i=w->num_samp-2; i>0 ; i--)
        if (w->wave[w->num_samp-1] < 0)
        {
            if (w->wave[i] > 0)
                break;
        }
        else if (w->wave[i] < 0)
            break;
    if ((w->num_samp - i) > 64)
        *lst = w->num_samp;
    else
        *lst = i;

    if ((*lst - *fst) < 100)
    {
        P_Warning("Unit is too short to move edges on");
        return;
```



```
    }

}

static void get_pm_points(P_Marks pm,int *st_pm, int *ls_pm)
{   /* Set first and last pitch mark as boundary for sub unit */
    if (pm->num_marks < 2)
    {
        P_Warning("Unit has less than 2 pitch marks, not moving boundaries");
        return;
    }
    else if (pm->num_marks < 5)
        P_Warning("Unit has less than 5 pitch marks, but moving boundaries");

    *st_pm = pm->mk[0].pos_samp;
    *ls_pm = pm->mk[pm->num_marks-1].pos_samp;
}

void unit_only_module(Utterance utt)
{
    /* Build a wave simply by concatenating the units in the unit stream */
    /* (No other streams are loaded                                      */

    struct Stream_cell *w_cell = new_stream_cell("Wave");

    SC(w_cell,Wave) = unit_dumb_concat_module(utt);

    utt_set_stream("Wave",w_cell,utt);

    return;

}

/* ----------------------------------------------------------------------------*/


/*************************************************************** Initialize */
/*  Initialize all XPSOLA parameters :                                       */
/*     This will readfrom variable "xpsola_params" the options that will be  */
/*     used during the following processing. If the variable has not been    */
/*     set, default values are asigned.                                      */
/*****************************************************************************/


static void xpsola_init_params(void)

{
   List psola_params, t;
   float threshold;
   char *str;
   int method;

   psola_params = list_str_eval("xpsola_params",NULL);

   threshold = param_get_float(psola_params, "P_thres", 0.1);
   sch.P_thres = MAX(0.01, threshold);                          /* default 1% */

   threshold = param_get_float(psola_params, "D_thres", 0.1);
   sch.D_thres = MAX(0.01, threshold);                          /* default 1% */

   threshold = param_get_float(psola_params, "P_ceil", 0.33);
   sch.P_ceil = MAX(0.01, threshold);                          /* default 33% */

   threshold = param_get_float(psola_params,"D_ceil", 0.66);
   sch.D_ceil = MAX(0.01, threshold);                          /* default 66% */
```

```
    str = param_get_str(psola_params,"P_method", "PSOLA");
    if (ci_streq(str, "intrapol"))  sch.P_method = 1;
    else sch.P_method = 2;                                  /* psola by default */

    str = param_get_str(psola_params,"D_method", "PSOLA");
    if (ci_streq(str, "smooth"))  sch.D_method = 1;
    else if (ci_streq(str, "select"))  sch.D_method = 2;
    else sch.D_method = 3;                                  /* psola by default */

    str = param_get_str(psola_params,"concat", "SELECT");
    if (ci_streq(str, "dumb"))  sch.concat = 1;
    else if (ci_streq(str, "select"))  sch.concat = 2;
    else if (ci_streq(str, "psola"))  sch.concat = 3;
    else if (ci_streq(str, "cepstral"))  sch.concat = 4;
    else if (ci_streq(str, "extrapol"))  sch.concat = 5;
    else if (ci_streq(str, "extrapol2"))  sch.concat = 6;
    else sch.concat = 2;                                    /* select by default */

    str = param_get_str(psola_params,"power", "NONE");
    if (ci_streq(str, "smooth"))  sch.power = 3;
    else if (ci_streq(str, "target"))  sch.power = 2;
    else sch.power = 1;                                     /* none by default */

    str = param_get_str(psola_params,"contour", "AVERAGED");
    if (ci_streq(str, "mixed"))  sch.contour = 3;
    else if (ci_streq(str, "target"))  sch.contour = 2;
    else sch.contour = 1;                                   /* averaged by default */

    str = param_get_str(psola_params,"model", "NONE");
    if (ci_streq(str, "hybrid"))  sch.model = 2;
    else if (ci_streq(str, "bands"))  sch.model = 3;
    else if (ci_streq(str, "hybrid+"))  sch.model = 4;
    else if (ci_streq(str, "bands+"))  sch.model = 5;
    else sch.model = 1;                                     /* none by default */

    method = param_get_num(psola_params, "PERMAX", 12);
    sch.PERMAX = MAX(0, MIN(30, method));                   /* 12 millisec by default */

    str = param_get_str(psola_params,"test", "NONE");
    if (ci_streq(str, "processing"))  sch.test = 1;
    else if (ci_streq(str, "model"))  sch.test = 2;
    else sch.test = 0;                                      /* none by default */

    str = param_get_str(psola_params,"stops", "NONE");
    if (ci_streq(str, "pitch"))  sch.stops = 1;
    else sch.stops = 0;                                     /* none by default */

    str = param_get_str(psola_params,"tiny", "DROP");
    if (ci_streq(str, "keep"))  sch.tiny = 1;
    else sch.tiny = 0;                                      /* drop'em by default */

    str = param_get_str(psola_params,"voicing", "PHONEME");
    if (ci_streq(str, "DATABASE"))  sch.voicing = 1;
    else sch.voicing = 0;                                   /* phoneme based, by default */

    threshold = param_get_float(psola_params, "SPAN", 18.0);
    threshold = MAX(1, threshold);
    SPAN = (int) threshold;                                 /* default 18 msec */

    return;

}


/************************************************** Prosodic Modifications */
/*  Main function for prosodic modifications :                             */
```



```
/*     This returns, according to the options set in "xpsola_params", the   */
/*     synthesized wave, given the input utterance.                         */
/*************************************************************************/


struct Wave *xpsola_concat_module(Utterance utt)

{
   Synthesis *S;
   Stream u;
   int i, size, offset, j, max;
   P_Wave pw, testwave;


   xpsola_init_params();                                  /* set the options */


   S = xps_INIT_ALL(utt);                                  /* initialization */

   for (u=UNITSTREAM(utt), i=0; u!=NULL; i++)
     {
       xps_READ_REFERENCE(S, i, u, sch);          /* reading reference pitch */
       xps_READ_WAVE(S, i, u);                        /* marks and waveforms */
       u = SC_next(u);
     }


/*************************************************************************/

if ((sch.test == 2)&&(sch.model != 1)) {      /* SEE HOW THE MODEL WORKS */

   max = MIN(8, S->nunits);                          /* only for so many units */

   for (i=0; i<max; i++)

     size += 4*S->nsamp[i];

   testwave = make_wave();
   testwave->wave = xalloc(size, short);                        /* allocation */
   testwave->num_samp = size;
   testwave->samp_rate = S->srate;
   bzero(testwave->wave, size*sizeof(short));

   for (i=0, offset = 0; i<max; i++) {                  /* original signal */

     if (S->nrfpm[i] > 1)
       size = S->ref[i][S->nrfpm[i]-1].nsamp;
     else size = 0;
     xps_strcopy(testwave->wave+offset, S->wl[i], size, 1);
     offset += size;  }

   offset += 2000;

   if ((sch.model == 2)||(sch.model == 4))

       xps_Hybrid(S, sch);                                    /* hybrid model */

   if ((sch.model == 3)||(sch.model == 5))                    /* bands model */

       xps_Bands(S);

   for (i=0; i<max; i++) {

     if (S->nrfpm[i] > 1)                                 /* voiced component */
        size = S->ref[i][S->nrfpm[i]-1].nsamp;
     else size = 0;
```

```
      xps_strcopy(testwave->wave+offset, S->wl[i], size, 1);
      offset += size;  }

   offset += 2000;

   for (i=0; i<max; i++) {

     if (S->nrfpm[i] > 1)
        size = S->ref[i][S->nrfpm[i]-1].nsamp;        /* unvoiced component */
     else size = 0;
     xps_strcopy(testwave->wave+offset, S->w2[i], size, 1);
     offset += size;  }

   return (testwave);
}

/*****************************************************************************/


   xps_POWER_PRO(S, utt, sch);                      /*  power smoothening */

   xps_MAPPING(S, sch, utt);                        /* target prosody */

   xps_MAKE_TARGET(S, sch);



/*****************************************************************************/

if (sch.test == 1) {                    /* SEE HOW THE PROCESSING WORKS */

   max = MIN(8, S->nunits);                      /* only for so many units */

   for (i=0; i<max; i++)

     size += 4*S->nsamp[i];

   testwave = make_wave();
   testwave->wave = xalloc(size, short);                  /* allocation */
   testwave->num_samp = size;
   testwave->samp_rate = S->srate;
   bzero(testwave->wave, size*sizeof(short));

   for (i=0, offset = 0; i<max; i++) {                    /* original units */

     if (S->nrfpm[i] > 1)
        size = S->ref[i][S->nrfpm[i]-1].nsamp;
     else size = 0;
     xps_strcopy(testwave->wave+offset, S->wl[i], size, 1);
     offset += size;  }

   offset += 1000;

   for (i=0; i<max; i++) {                              /* processed units */

     size = S->nsamp[i];
     xps_MAKE_UNIT(S, i, sch);
     xps_strcopy(testwave->wave+offset, S->wl[i], size, 1);
     offset += size;  }

   offset += 2000;

   for (i=0; i<max; i++) {                                /* test units */
     if (S->nrfpm[i] > 1)
       for (j=0; j<S->ref[i][S->nrfpm[i]-1].nsamp; j++)
```



```
        S->wl[i][j] = 500+2*j;
     size = S->nsamp[i];                                        /* sawtooth */
     xps_MAKE_UNIT(S, i, sch);
     xps_strcopy(testwave->wave+offset, S->wl[i], size-1, 1);
     offset += size-1;  }


   return(testwave);
}
/*****************************************************************************/


   if ((sch.model == 2)||(sch.model == 4))

                     xps_Hybrid(S, sch);                    /* hybrid model */

   if ((sch.model == 3)||(sch.model == 5))

                     xps_Bands(S);                           /* bands model */

   if (((sch.model != 2)&&(sch.model != 4)) ||
                         (sch.P_thres != 2) || (sch.D_thres != 2))

      for (i=0; i<S->nunits; i++)

        xps_MAKE_UNIT(S, i, sch);               /* processing is done HERE */

   if  (sch.model != 1)

      for (i=0; i<S->nunits; i++)

        xps_NOISE_PRO(S, i, sch.model);              /* special processing */

   pw = xps_FINAL_WAVE(S, sch);                  /* the wave is created HERE */

   xfree(S);

   return (pw);


}
```

```
/*=======================================================================*/
/*           A T R Interpreting Telecommunications Labs                  */
/*                                                                       */
/*=======================================================================*/
/*               CHATR Speech Synthesis System                           */
/*                    Christian Lelong                                   */
/*-----------------------------------------------------------------------*/
/*                                                                       */
/*        Unit Concatenation & Prosodic Modifications                    */
/*                                                                       */
/*                          Feb 1995                                     */
/*                    Copyrigth (C) 1994, 1995                           */
/*     ATR Interpreting Telecommunications Research Laboratories         */
/*                    All rights reserved.                               */
/*=======================================================================*/
#ifndef __XRUC_H__
#define __XRUC_H__
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "xtop.h"

#define NOISY(a) (((a) != 1)  && ((a) !=2) && ((a) !=3))
#define FIL_NUM 13
#define FIL_ORD 201
#define BANDS 13


int SPAN;

extern const float Filter_Bank[FIL_ORD][FIL_NUM];

/************************************************************ xmisc.c  */

void xps_Push_Marks(Marks *pm, int npm);
int xps_Is_Alone(Marks *pm, int span);
int xps_Find_Boundary(struct Unit *punit, int i, Marks *pmarks, int npm);
void xps_Filter_Samples(int k, short *data, int start, int length, short *out);
int xps_Test_Voicing(struct Unit *pu, Marks *pm, int npm,int time, int permax);
int xps_INDEX(Marks *pm, int type);
int xps_INDEX_bis(Marks *pm, int type);
float xps_MEAN_POWER(Utterance utt, Synthesis *S, int n);


/************************************************************ xmath.c  */

int xps_ispower2(int a);
float xps_Sygmoid(float x);
float *xps_Window(int dim,int scale, int middle, int type);
float xps_RMS(short *wave, Marks *pm, int n);
float xps_RMS_bis(short *wave, int start, int size);
float xps_LOG_POW(Synthesis *S, int n);
void xps_WIN_POW(float *s, Marks *pm1, Marks *pm2, float *w);
void xps_acf(float *V,int m);
void xps_Cepstrum (short *data, float *cep, int length);
void xps_Icepstrum (short *data, float *icep, int length);
float xps_Is_Voiced(short *data, int size, int unit, float k);
Joint xps_Cut_Point(short *w1, short *w2, Marks *pm1, Marks *pm2);


/************************************************************ xpros.c  */

void xps_MAKE_TARGET(Synthesis *S, Scheme sch);
void xps_MAKE_UNIT(Synthesis *S, int rank, Scheme sch);
void xps_NP(short * wave, Marks *ref, Marks *tar, int type, short *buffer);
int xps_CONCAT(Synthesis *S, int rank, P_Wave pw, int index, int method);
```



```
/************************************************************ xmod.c   */

void xps_NOISE_PRO(Synthesis *S, int rank, int type);
void xps_Hybrid(Synthesis *S, Scheme sch);
void xps_Bands(Synthesis *S);

/************************************************************ xio.c    */

int xps_PHONEM(struct Sub_Unit *su);
Synthesis *xps_INIT_ALL(Utterance utt);
void xps_READ_REFERENCE(Synthesis *S, int rank, Stream u, Scheme sch);
void xps_READ_REFERENCE_bis(Synthesis *S, int rank, Stream u, int PERMAX);
void xps_MAPPING(Synthesis *S, Scheme sch, Utterance utt);
void xps_PERIODS(int *f0, int total,int start,int end,int *n,float *periods);
void xps_READ_WAVE(Synthesis *S, int rank, Stream u);
void xps_POWER_PRO(Synthesis *S, Utterance utt, Scheme sch);
P_Wave xps_FINAL_WAVE(Synthesis *S, Scheme sch);

#endif
```

Appendix 1

XPSOLA unit - code

Following are the different files composing the *XPSOLA* module.

xio.c : input, output and initialisation

xmath.c : mathematical routines

xmisc.c : miscellaneous low-level functions

xmod.c : bands and hybrid models

xpros.c : main signal processing functions

ruc.c : top-level functions commanding the whole processing.

xruc.h : declarations of functions visible to other files.

xtop.h : data types, and most of the constants and macros.

Appendix 2

working in Japan

Considering all the books and articles that have been devoted to the issue, there must be something about it. As Japan has become richer and its technology better over the past years, its appeal to foreign managers, engineers and scientists has increased. Settling and working in Japan, however, is not like doing the same in a Western country.

Japan is an island with a rich and ancient culture that stayed isolated from the rest of Asia and the world until recently. True, it owes much to China, and Korea to a lesser extent, and Portuguese missionaries began to settle in the XVI century. But open contact with Western civilisation is only a century old, and big cultural differences exist, visible almost everywhere, and capable of disorienting foreigners for a long time. Social etiquette, sense of humour and much more differ enormously, which is a good thing : much of Japan`s charm lies there. But to a *gaijin*, who moreover comes with a not too flattering image of the country, this can lead to a bad start.

I beleive ATR is somehow speial in Japan, more international and open-minded than most places. During my stay, certain facts became evident, the first one concerning language. As most Japanese speak very poor English, learning Japanese is not only a polite or enjoyable thing to do, but almost a necessity. Otherwise, one is limited to dealing mostly with fellow foreigners, and this certainly won`t help at work nor anywhere else. This was the case at ATR, and I often wished I was more fluent. However, even when communication is not difficult, it is easy to notice the special treatment foreigners receive. They are not expected to comply with all the sometimes fastidious formalities Japanese colleagues have to endure. Also, the management section is full of attentions. This might lead to think that *gaijin* have a privileged situation, but that is only partly true, for it seems that foreigners are seldom fully accepted and integrated. A recent poll confirms this impression, shared by people I met.

Concerning the state of research, I was slightly disappointed. It is a fact that Japan is the world leader in consumer electronics ; it`s also a fact that Japan`s share of scientific Nobel Prizes and international papers, compared to Europe`s or America`s, is not very

impressive. My impression during the six months of my stay was that ATR as a whole is a big laboratory with many resources and much money at its disposal, and still it is inefficient. The few comparable places I have visited seem to accomplish more with less. I have only thought of two reasons to explain this : the still low percentage of foreigners, and what we shall call the "self-restraint" of the Japanese. Indeed, mixing people with different backgrounds and education can only increase the team's creativity, each member providing a certain approach, method and point of view. And it is to be hoped that researchers can communicate and discuss ideas freely, with bothering too much about loss of face, excessive modesty or extreme respect of other people's point of view.

As a whole, working in Japan, with a workforce predominantly Japanese but still quite international, was a splendid experience, that will certainly be of great help in the future. If one is to be as productive as one could, some effort is required before integration with the team is accomplished. This process can be tricky, and must be learned. And an obligatory step is, I think, adaptation to the ways of the host country. The stereotype of the narrow-minded, closed, no-nonsense Japanese working himself to death has, as most cliches, little truth in it. For many of my colleagues, the initial barriers eventually broke down, and they turned out to be great companions.

Too many foreigners, partly due to their poor level in Japanese, fail to make such a step, and it is most regrettable. And besides professional reasons, they are missing a fascinating country, in particular in the Kansai area of Osaka-Kyoto-Nara (featuring fabulous temples and lively people) where ATR is located.

Appendix 3

CHATR voices

Take 1

no prosodic modifications, dumb+ concatenation

@gsw Hello, here is a short demonstration of the voices available in CHATR. This voice is a British English RP male speaker called Gordon. It is based on a database of 200 phonetically balanced sentences. *@sally* This is a British English RP female speaker called Sally. This voice was made from a database of the same 200 phonetically balanced sentences as the previous male voice. *@sab600* When we increase the number of sentences to over 400, the quality improves. As now, thewre are many more examples to choose from. But, if we use the data from isolated words, rather than continuous specch, *@sab5* the quality becomes less natural, and overarticulated. Also, the durations are much longer. *@wnc600* Another British English male speaker is Nick. This voice is based on around 600 sentences. Again, due to the large number of examples to choose from, the quality increases. *@f2b* With a quick flight across the Atlantic we get a female American voice. This voice is built from 45 minutes of speech, from the speaker called f2b, from the Boston University FM Radio corpus. Of course, synthesizing from a news announcer corpus means the speech sounds like an evening news broadcast. Now, over to our correspondent in Tokyo.

(in Japanese)

@mhtbset This speech synthesizer is not limited to English, but can also produce Japanese speech. This male voice was built from a set of 503 sentences. *@fmp* A woman's voice also is available. Over to you, Gordon-san.

@gsw Thank you, FMP-san. We should not forget the older voices that existed in CHATR from the near beginning. *@gswdi* This voice is the CSTR diphone synthesizer developed at Edinburgh University. This British English male voice was recorded by the same person, Gordon, who is in the other larger British English male databases. *@isard* Another voice is this LPC diphone synthesizer, also developed at Edinburgh University. Although clear, perhaps it sounds more like a synthesizer should ?

@sab600 Finally, let me play the voice that CHATR first used. It is a formant synthesizer, copied from a free synthesizer available on the net. *@formant* But unfortunately, it is mostly incomprehensible.

@f2b And, that's the way it is. So from me, f2b, *@sab600* Sally, *@gsw* Gordon, *@wnc600* Nick, *@f2b* and all the others, thank you for listening. For WBCR, I'm f2b. Over to you, Jim

Take 2

no prosodic modifications, dumb+ concatenation

A 1980 state constitutional ammendment made Massachussets one of 23 states where citizens can enact laws by pleibiscite.

1. original natural waveform

2. minimize target distance

3. minimize continuity distance

4. equal weightings

5. minimize cepstrum distance

Take 3

pitch modification in the final word with PSOLA

I'm often perplexed by rapid advances in state of the art technology.

References

[1] J.L. Flanagan
Speech Analysis, Synthesis and Perception
Second Edition, Springer-Verlag, 1972

[2] Helene Valbret
Systeme de conversion de la voix pour la synthese de parole
PhD thesis, ENST, Paris 1994

[3] Eric Moulines
Algorithmes de codage et de modification des parametres prosodiques pour la synthèse de la parole a partir du texte.
PhD thesis, ENST, Paris 1990

[4] Olivier Boeffard & Fabio Violaro
Using a hybrid model in a Text-To-Speech system to enlarge prosodic modifications.
Proceedings ICSLP 94, Yokohama, Japon, Septembre '94

[5] J.P. Olive, A. Greenwood & J.S. Coleman
The dynamics of American English speech

[6] Alan W. Black & Paul Taylor
CHATR : a generic speech synthesis system
Proceedings COLING 94, Kyoto, Japon, Avril '94

[7] Pierre-Yves le Meur
Protection de segments sub-phonetiques en synthese PSOLA
XXemes Journees d'Etude sur la Parole, Tregastel, France, June 1994

[8] J. Allen, M. Sharon Hunnicutt & D. Klatt
From text to speech : the MITalk system
1987, Cambridge University Press

[9] Gael Richard
Modelisation de la composante stochastique de la parole
PhD thesis, Universite de Paris XI, April 1994

Reference guides

[10] Signal Processing Toolbox User's Guide, MATLAB
The Math Works Inc, November 1993

[11] Xwaves+ guide, version 5.0
Entropic Research Laboratory, Inc. 1993.

[12] Purify, version 3.01 User's Guide
Pure Software Inc., 1994.