Building Synthetic Voices

Alan W Black

Kevin A. Lenzo

Building Synthetic Voices
by Alan W Black and Kevin A. Lenzo

For FestVox 2.1 Edition

Table of Contents

Chapter 1. Overview of Speech Synthesis

History

* *AWB: probably way too biased as a history*

The idea that a machine could generate speech has been with us for some time, but the realization of such machines has only really been practical within the last 50 years. Even more recently, it's in the last 20 years or so that we've seen practical examples of text-to-speech systems that can say any text they're given -- though it might be "wrong."

The creation of synthetic speech covers a whole range of processes, and though often they are all lumped under the general term *text-to-speech*, a good deal of work has gone into generating speech from sequences of speech sounds; this would be a speech-sound (phoneme) to audio waveform synthesis, rather than going all the way from text to phonemes (speech sounds), and then to sound.

One of the first practical application of speech synthesis was in 1936 when the U.K. Telephone Company introduced a speaking clock. It used optical storage for the phrases, words, and part-words ("noun," "verb," and so on) which were appropriately concatenated to form complete sentences.

Also around that time, Homer Dudley developed a mechanical device at Bell Laboratories that operated through the movement of pedals, and mechanical keys, like an organ. With a trained operator, it could be made to create sounds that, if given a good set-up, almost sounded like speech. Called the *Voder*, it was demonstrated at the 1939 World's Fair in New York and San Francisco. A recording of this device exists, and can be heard as part of a collection of historical synthesis examples that were distributed on a record as part of [klatt87].

The realization that the speech signal could be decomposed as a source-and-filter model, with the glottis acting as a sound source and the oral tract being a filter, was used to build analog electronic devices that could be used to mimic human speech. The *vocoder*, also developed by Homer Dudley, is one such example. Much of the work in synthesis in the 40s and 50s was primarily concerned with constructing replicas of the signal itself rather than generating the phones from an abstract form like text.

Further decomposition of the speech signal allowed the development of *formant synthesis*, where collections of signals were composed to form recognization speech. The prediction of parameters that compactly represent the signal, without the loss of any information critical for reconstruction, has always been, and still is, difficult. Early versions of formant synthesis allowed these to be specified by hand, with automatic modeling as a goal. Today, formant synthesizers can produce high quality, recognizable speech if the parameters are properly adjusted, and these systems can work very well for some applications. It's still hard to get fully natural sounding speech from these when the process is fully automatic -- as it is from all synthesis methods.

With the rise of digital representations of speech, digital signal processing, and the proliferation of cheap, general-purpose computer hardware, more work was done in concatenation of natural recorded speech. *Diphones* appeared; that is, two adjacent half-phones (context-dependent phoneme realizations), cut in the middle, joined into one unit. The justification was that phone boundaries are much more dynamic than stable, interior parts of phones, and therefore mid-phone is a better place to concatenate units, as the stable points have, by definition, little rapid change, whereas there are rapid changes at the boundaries that depend upon the previous or next unit.

The rise of concatenative synthesis began in the 70s, and has largely become practical as large-scale electronic storage has become cheap and robust. When a megabyte of memory was a significant part of researchers salary, less resource-intensive techniques were worth their... weight in saved cycles in gold, to use an odd metaphor. Of course formant, synthesis can still require significant computational power, even if it

requires less storage; the 80s speech synthesis relied on specialized hardware to deal with the constraints of the time.

In 1972, the standard Unix manual (3rd edition) included commands to process text to speech, form text analysis, prosodic prediction, phoneme generation, and waveform synthesis through a specialized piece of hardware. Of course Unix had only about 16 installations at the time and most, perhaps even all, were located in Bell Labs at Murray Hill.

Techniques were developed to compress (code) speech in a way that it could be more easily used in applications. The Texas Instruments *Speak 'n Spell* toy, released in the late 70s, was one of the early examples of mass production of speech synthesis. The quality was poor, by modern standards, but for the time it was very impressive. Speech was basically encoded using LPC (linear Predictive Coding) and mostly used isolated words and letters though there were also a few phrases formed by concatenation. Simple text-to-speech (TTS) engines based on specialised chips became popular on home computers such as the BBC Micro in the UK and the Apple][.

Dennis Klatt's MITalk synthesizer [allen87] in many senses defined the perception of automatic speech synthesis to the world at large. Later developed into the product DECTalk, it produces somewhat robotic, but very understandable, speech. It is a formant synthesizer, reflecting the state of the art at the time.

Before 1980, research in speech synthesis was limited to the large laboratories that could afford to invest the time and money for hardware. By the mid-80s, more labs and universities started to join in as the cost of the hardware dropped. By the late eighties, purely software synthesizers became feasible; the speech quality was still decidedly inhuman (and largely still is), but it could be generated in near real-time.

Of course, with faster machines and large disk space, people began to look to improving synthesis by using larger, and more varied inventories for concatenative speech. Yoshinori Sagisaka at Advanced Telecommunications Research (ATR) in Japan developed nuu-talk [nuutalk92] in the late 80s and early 90s. It introduced a much larger inventory of concatenative units; thus, instead of one example of each diphone unit, there could be many, and an automatic, acoustically based distance function was used to find the best selection of sub-word units from a fairly broad database of general speech. This work was done in Japanese, which has a much simpler phonetic structure than English, making it possible to get high quality with a relatively small databases. Even up through 1994, the time needed to generate of the parameter files for a new voice in nuu-talk (503 senetences) was on the order of several days of CPU time, and synthesis was not generally possible in real time.

With the demonstration of general *unit selection synthesis* in English in Rob Donovan's PhD work [donovan95], and ATR's CHATR system ([campbell96] and [hunt96]), by the end of the 90's, unit selection had become a hot topic in speech synthesis research. However, despite examples of it working excellently, generalized unit selection is known for producing very bad quality synthesis from time to time. As the optimial search and selection agorithms used are not 100% reliable, both high and low quality synthesis is produced -- and many diffilculties still exists in turning general corpora into high-quality synthesizers as of this writing.

Of course, the development of speech synthesis is not isolated from other developments in speech technology. Speech recognition, which has also benefited from the reduction in cost of computational power and increased availability of general computing into the populace, informs a the work on speech synthesis, and vice versa. There are now many more people who have the computational resouces and interest in running speech applications, and this ability to run such applications puts the demand on the technology to deliver both working recognition and acceptable quality speech synthesis.

The availability of free and semi-free synthesis systems, such as the Festival Speech Synthesis System and the MBROLA project, makes the cost of entering the field of speech synthesis much lower, and many more groups have now joined in the development.

However, although we are now at the stage were talking computers are with us, there is still a great deal of work to be done. We can now build synthesizers of (probably) any language that can produce reconizable speech, with a sufficient amount of work; but if we are to use speech to receive information as easily when we're talking with computers as we do in everyday conversation, synthesized speech must be natural, controllable and efficient (both in rendering and in the building of new voices).

Uses of Speech Synthesis

While speech and language were already important parts of daily life before the invention of the computer, the equipment and technology that has developed over the last several years has made it possible to have machines that speak, read, or even carry out dialogs. A number of vendors provide both recognition and speech technology, and there are several telephone-based systems that do interesting things.

...

General Anatomy of a Synthesizer

[diagram: text going in and moving around, coming out audio]

Within Festival we can identify three basic parts of the TTS process

Text analysis:

From raw text to identified words and basic utterances.

Linguistic analysis:

Finding pronunciations of the words and assigning prosodic structure to them: phrasing, intonation and durations.

Waveform generation:

From a fully specified form (pronunciation and prosody) generate a waveform.

These partitions are not absolute, but they are a good way of chunking the problem. Of course, different waveform generation techniques may need different types of information. *Pronunciation* may not always use standard phones, and *intonation* need not necessarily mean an F0 contour. For the main part, at along least the path which is likely to generate a working voice, rather than the more research oriented techniques described, the above three sections will be fairly cleanly adhered to.

There is another part to TTS which is normally not mentioned, we will mention it here as it is the most important aspect of Festival that makes building of new voices possible -- the system *architecture*. Festival provides a basic utterance structure, a language to manipulate it, and methods for construction and deletion; it also interacts with your audio system in an efficient way, spooling audio files while the rest of the synthesis process can continue. With the Edinburgh Speech Tools, it offers basic analysis tools (pitch trackers, classification and regression tree builders, waveform I/O etc) and a simple but powerful scripting language. All of these functions make it so that you may get on with the task of building a voice, rather than worrying about the underlying software too much.

Text

We try to model the voice independently of the meaning, with machine learning techniques and statistical methods. This is an important abstraction, as it moves us from the realm of "all human thought" to "all possible sequences." Rather than asking "when and why should this be said," we ask "how is this performed, as a series of speech sounds?" In general, we'll discuss this under the heading of text analysis -- going from written text, possibly with some mark-up, to a set of words and their relationships in an internal representation, called an utterance structure.

Text analysis is the task of identifying the *words* in the text. By *words*, we mean tokens for which there is a well defined method of finding their pronunciation, i.e. from a lexicon, or using letter-to-sound rules. The first task in text analysis is to make chunks out of the input text -- *tokenizing* it. In Festival, at this stage, we also chunk the text into more reasonably sized utterances. An utterance structure is used to hold the information for what might most simply be described as a *sentence*. We use the term loosely, as it need not be anything syntactic in the traditional linguistic sense, though it most often has prosodic boundaries or edge effects. Separating a text into utterances is important, as it allows synthesis to work bit by bit, allowing the waveform of the first utterance to be available more quickly than if the whole files was processed as one. Otherwise, one would simply play an entire recorded utterance -- which is not nearly as flexible, and in some domains is even impossible.

Utterance chunking is an externally specifiable part of Festival, as it may vary from language to language. For many languages, tokens are white-space separated and utterances can, to a first approximation, be separated after full stops (periods), question marks, or exclamation points. Further complications, such as abbreviations, other-end punctuation (as the upside-down question mark in Spanish), blank lines and so on, make the definition harder. For languages such as Japanese and Chinese, where white space is not normally used to separate what we would term words, a different strategy must be used, though both these languages still use punctuation that can be used to identify utterance boundaries, and word segmentation can be a second process.

Apart from chunking, text analysis also does text *normalization*. There are many tokens which appear in text that do not have a direct relationship to their pronunciation. Numbers are perhaps the most obvious example. Consider the following sentence

> On May 5 1996, the university bought 1996 computers.

In English, tokens consisting of solely digits have a number of different forms of pronunciation. The "*5*" above is pronounced "*fifth*", an ordinal, because it is the day in a month, The first "*1996*" is pronounced as "*nineteen ninety six*" because it is a year, and the second "*1996*" is pronounced as "*one thousand nine hundred and ninety size*" (British English) as it is a quantity.

Two problems that turn up here: non-trivial relationship of tokens to words, and *homographs*, where the same token may have alternate pronunciations in different contexts. In Festival, homograph disambiguation is considered as part of text analysis. In addition to numbers, there are many other symbols which have internal structure that require special processing -- such as money, times, addresses, etc. All of these can be dealt with in Festival by what is termed *token-to-word rules*. These are language specific (and sometimes text mode specific). Detailed examples will be given in the text analysis chapter below.

Lexicons

After we have a set of words to be spoken, we have to decide what the sounds should be -- what phonemes, or basic speech sounds, are spoken. Each language and dialect has a phoneme set associated with it, and the choice of this inventory is still not

agreed upon; different theories posit different feature geometries. Given a set of units, we can, once again, train models from them, but it is up to linguistics (and practice) to help us find good levels of structure and the units at each.

Prosody

Prosody, or the way things are spoken, is an extremely important part of the speech message. Changing the placement of emphasis in a sentence can change the meaning of a word, and this emphasis might be revealed as a change in pitch, volume, voice quality, or timing.

We'll present two approaches to taming the prosodic beast: limiting the domain to be spoken, and intonation modeling. By limiting the domain, we can collect enough data to cover the whole output. For some things, like weather or stock quotes, very high quality can be produced, since these are rather contained. For general synthesis, however, we need to be able to turn any text, or perhaps concept, into a spoken form, and we can never collect all the sentences anyone could ever say. To handle this, we break the prosody into a set of features, which we predict using statistically trained models.

- phrasing - duration - intonation - energy - voice quality

Waveform generation

For the case of concatenative synthesis, we actually collect recordings of voice talent, and this captures the voice quality to some degree. This way, we avoid detailed physical simulation of the oral tract, and perform synthesis by integrating pieces that we have in our inventory; as we don't have to produce the precisely controlled articulatory motion, we can model the speech using the units available in the sound alone -- though these are the surface realization of an underlying, physically generated signal, and knowledge of that system informs what we do. During waveform generation, the system assembles the units into an audio file or stream, and that can be finally "spoken." There can be some distortion as these units are joined together, but the results can also be quite good.

We systematically collect the units, in all variations, so as to be able to reproduce them later as needed. To do this, we design a set of utterances that contain all of the variation that produces meaningful or apparent contrast in the language, and record it. Of course, this requires a theory of how to break speech into relevant parts and their associated features; various linguistic theories predict these for us, though none are undisputed. There are several different possible unit inventories, and each has tradeoffs, in terms of size, speed, and quality; we will discuss these in some detail.

Chapter 2. Speech Science

speech, articulators, formats, phones, syllables, utterances, intonation etc.
generating speech, formant, concatenative,

Chapter 3. A Practical Speech Synthesis System

The Festival Speech Synthesis Systems[1] was developed at the Centre for Speech Technology Reseach[2] at the University of Edinburgh[3] in the late 90's. It offers a free, portable, language independent, run-time speech synthesis engine for verious platforms under various APIs. This book is not about the Festival system itself, Festival is just the engine that we will use in the process of building voices, both as a run-time engine for the voices we build and as a tool in the building process itself. This chapter gives a background on the philosophy of the system, its basic use, and some lower level details on its internals that will make the understanding of the whole synthesis task easier.

The Festival Speech Synthesis System was designed to target three particular classes of speech synthesis user.

1. Speech synthesis researchers: where they may use Festival as a vehicle for developmeent and testing of new research in synthesis technology.
2. Speech application developers: where synthesis is not the primary interest, but Festival will be a substantial sub-component which may require significant integration and hence the system must be open and easily configurable.
3. End user: where the system simple takes text and generates speech, requiring no or very little configuration from the user.

In the design of Festival it was important that all three classes of user were served as there needs to be a clear route from research work to practial usable systems as this not only encourages research to be focussed but also, as has been shown by the large uptake of the system, ensures there is a large user community interested in seeing improvements to the system.

The Festival Speech Synthesis System was built based on the experience of previous synthesis engines. Design of a key architecture is important as what may seem general to begin with can quickly become a limiting factor, as new and more ambitious techniques are attempted within it. The basic architecture of Festival benefited mainly from previous synthesis engines developed at Edinburgh University, specifically Osprey [taylor91]. ATR's CHATR system, [black94] was also a major influence on Festival, CHATR's original core architecture was also developed by the same authors as Festival. In designing Festival, the intention was to avoid the previous limitations in the utterance representation and module specification, specifically in avoiding constraints on the types of modules and dependencies between them. However even with this intent, Festival went through a number of core changes before it settled.

The Festival system consists of a set of C++ objects and core methods suitable for doing synthesis tasks. These objects include synthesis specific objects like, waveforms, tracks and utterances as well as more general objects like feature sets, n-grams, and decision trees.

In order to give parameters and specify flow of control Festival offers a scripting language based on the Scheme programming language [Scheme96]. Having a scripting language is one of the key factors that makes Festival a useful system. Most of the techniques in this book for building new voices within Festival can be done without any changes to the core C++ objects. This makes development of new voices not only more accessible to a larger population or users, as C++ knowledge nor a C++ compiler is necessary, it also makes the distribution of voices built by these techniques easy as users do not require any recompilation to use newly created voices.

Scheme offers a very simple syntax but powerful language for specifying parameters and simple functions. Scheme was chosen as its implementation is small and would not increase the size of the Festival system unnecessarily. Also, using an embedded Scheme component does not increase the requirements for installation as would the use of say Java, Perl or Python as the scripting language. Scheme frightens some

people as Lisp based languages have an unfair reputation for being slow. Festival's use of Scheme is (in general) limited to simple functions and very little time is spent in the Scheme interpreter itself. Automatic garbage collection also has a reputation for slowing systems down. In Festival, garbage collection happens after each utterance is synthesized and again takes up only a small amount of time but allows the programmer nor to have to worry about explicitly freeing memory.

For the most part the third type of user, defined above, will never need to change any part of the systems (though they usually find something they want to change, like adding new entries to the lexicon). The second level of user typically does most of their customizing in Scheme, though this is usually just modifying existing pieces of Scheme in the way that people may add simple lines of Lisp to their `.emacs` file. It is primarily only the synthesis research community that has to deal with the C++ end of the system, though C/C++ interfaces to the systems as a library are also provided (see Chapter 24 for more discussions on APIs).

This chapter covers the basic use of the system and is followed by more details of the internal structures, particularly the utterance sturcture, accessing methods and modules. These later sections are probabaly more detail than one needs for building standard voices described in the book, but the is information is necessary when more ambituous voice building tasks are attempted.

Basic Use

The examples here are given based on a standard installation on a Unix system as described in Chapter 29, however the examples are likely to work under any platform Festival supports.

The most simple way to use Festival to speak a file from the command line, is by the command

```
festival --tts example.txt
```

This will speak the text in `example.txt` using the default voice.

Festival can also read text from `stdin` using a command like

```
echo "Hello world" | festival --tts
```

Festival actually offers two modes, a *text mode* and a *command mode*. In text mode everything given to Festival is treated as text to be spoken. In comamnd mode everything is treated as Scheme commands and interpreted.

When festival is started with no arguments if goes into interactive command mode. There you may type Scheme command and have Festival interpret them. For example

```
$ festival
....
festival>
```

One simple command is `SayText` takes a single string argument and says its contents.

```
festival> (SayText "Hello world.")
#<Utterance 0x402a9ce8>
festival>
```

You may select other voices for synthesis by calling the appropriate function. For example

```
festival> (voice_cmu_sls_diphone)
cmu_us_sls_diphone
festival> (SayText "Hello world.")
#<Utterance 0x402f0648>
festival>
```

Will use a female US English voice (if installed).

The command line interface offers comand line history though the up and down arrows (ctrl-P and ctrl-N) and editing through standard **emacs**-like commands. Importantly the interface does function and filename completion too, using the **TAB** key.

Any Scheme command may be typed at the command line for example

```
festival> (Parameter.set 'Duration_Stretch 1.5)
1.5
festival>
```

Will make all durations longer for the current voice (making the voice speak slower.

```
festival> (SayText "a very slow example.")
#<Utterance 0x402f564376>
festival>
```

Calling any specific voice will reset this value (or you may do it by hand).

```
festival> (voice_cmu_us_kal_diphone)
cmu_us_kal_diphone
festival> (SayText "a normal example.")
#<Utterance 0x402e3348>
festival>
```

The `SayText` is just a simple function that takes the given string, constructs an utterance object from is, synthesizes it and sends the resulting waveform to the audio device. This isn't really suitable for synthesizing anythign but very short utterances. The TTS process involves the more complex task of splitting text streams into utterance synthesizing them and sendthem to the audio device to they may play as the same time working on the next utterance to that the audio output is continuous. Festival does this through the `tts` function (which is what is actually called when Festival is given the `--tts` argument on the command line. In Scheme the `tts` funciton takes two arguments, a filename and a mode. Modes are described in more detail in the Section called *TTS modes* in Chapter 6, and can be used to allow special processing of text, such as respecting markup or particular styles of text like email etc. In simple case the mode will be `nil` which denotes the basic raw or fundamental mode.

```
festival> (tts "WarandPeace.txt" nil)
t
festival>
```

Commands can also be stored in files, which is normal when a number of function definitions and parameter settings are required. These scheme files can be loaded by the function `SayText` as in

```
festival> (load "commands.scm")
t
festival>
```

Arguments to Festival at startup time will normally be treated as command files and loaded.

```
$ festival commands.scm
...
festival>
```

However if the argument starts with a left parenthesis (the argument is interpreted directly as a Scheme command.

```
$ festival '(SayText "a short example.")'
...
festival>
```

If the `-b` (batch) option is specified Festival does not go into interactive mode and exits after processing all of the given arguments.

```
$ festival -b mynewvoicedefs.scm '(SayText "a short example.")'
```

Thus we can use Festival interactively or simple as a batch scripting language. The batch format will be used often in the voice building process though the intereactive mode is useful for testing new voices.

Utterance structure

The basic building block for Festival is the *utterance*. The structure consists of a set of *relations* over a set of *items*. Each item represents a object such as a word, segment, syllable, etc. while relations relate these items together. An item may appear in multiple relations, such as a segment will be in a `Segment` relation and also in the `SylStructure` relation. Relations define an ordered structure over the items within them, in general these may be arbitrary graphs but in practice so far we have only used *lists* and *trees* Items may contain a number of features.

There are no built-in relations in Festival and the names and use of them is controlled by the particular modules used to do synthesis. Language, voice and module specific relations can easy be created and manipulated. However within our basic voices we have followed a number of conventions that should be followed if you wish to use some of the existing modules.

The relation names used will depend on the particular structure chosen for your voice. So far most of our released voices have the same basic structure though some of our research voices contain quite a different set of relations. For our basic English voices the relations used are as follows

`Text`

Contains a single item which contains a feature with the input character string that is being synthesized

`Token`

A list of trees where each root of each tree is the white space separated tokenized object from the input character string. Punctuation and whitespace has been stripped and placed on features on these token items. The daughters of each of these roots are the list of words that the token is associated with. In many cases this is a one to one relationship, but in general it is one to zero or more. For example tokens comprising of digits will typically be associated with a number of words.

`Word`

The words in the utterance. By *word* we typically mean something that can be given a pronunciation from a lexicon (or letter-to-sound rules). However in most of our voices we distinguish pronunciation by the words and a part of speech feature. Words with also be leaves of the `Token` relation, leaves of the `Phrase` relation and roots of the `SylStructure` relation.

`Phrase`

A simple list of trees representing the prosodic phrasing on the utterance. In our voices we only have one level of prosodic phrase below the utterance (though you can easily add a deeper hierarchy if your models require it). The tree roots are labeled with the phrase type and the leaves of these trees are in the `Word` relation.

`Syllable`

A simple list of syllable items. These syllable items are intermediate nodes in the `SylStructure` relation allowing access to the words these syllables are in and the segments that are in these syllables. In this format no further onset/coda distinction is made explicit but can be derived from this information.

`Segment`

A simple list of segment (phone) items. These form the leaves of the `SylStructure` relation through which we can find where each segment is placed within its syllable and word. By convention silence phones do not appear in any syllable (or word) but will exist in the segment relation.

`SylStructure`

A list of tree structures over the items in the `Word`, `Syllable` and `Segment` items.

`IntEvent`

A simple list of intonation events (accents and boundaries). These are related to syllables through the `Intonation` relation.

`Intonation`

A list of trees whose roots are items in the `Syllable` relation, and daughters are in the `IntEvent` relation. It is assumed that a syllable may have a number of intonation events associated with it (at least accents and boundaries), but an intonation event may only by associated with one syllable.

`Wave`

A relation consisting of a single item that has a feature with the synthesized waveform.

`Target`

A list of trees whose roots are segments and daughters are F0 target points. This is only used by some intonation modules.

`Unit, SourceSegments, Frames, SourceCoef TargetCoef`

A number of relations used the the `UniSyn` module.

Modules

The basic synthesis process in Festival is viewed as applying a set of *modules* to an utterance. Each module will access various relations and items and potentially generate new features, items and relations. Thus as the modules are applied the utterance structure is filled in with more and more relations until ultimately the waveform is generated.

Modules may be written in C++ or Scheme. Which modules are executed are defined in terms of the utterance `type`, a simple feature on the utterance itself. For most text-to-speech cases this is defined to be of type `Tokens`. The function `utt.synth` simply looks up an utterance's type and then looks up the definition of the defined synthesis process for that type and applies the named modules. Synthesis types maybe defined using the function `defUttType`. For example definition for utterances of type `Tokens` is

```
(defUttType Tokens
  (Token_POS utt)
  (Token utt)
  (POS utt)
  (Phrasify utt)
  (Word utt)
  (Pauses utt)
  (Intonation utt)
  (PostLex utt)
  (Duration utt)
  (Int_Targets utt)
  (Wave_Synth utt)
  )
```

While a simpler case is when the input is phone names and we don't wish to do all that text analysis and prosody prediction. Then we use the type `Phones` which simply loads the phones, applies fixed prosody and the synthesizes the waveform

```
(defUttType Phones
  (Initialize utt)
  (Fixed_Prosody utt)
  (Wave_Synth utt)
  )
```

In general the modules named in the type definitions are general and actually allow further selection of more specific modules within them. For example the `Duration` module respects the global parameter `Duration_Method` and will call then desired duration module depending on this value.

When building a new voice you will probably not need to change any of these definitions, though you may wish to add a new module and we will show how to do that without requiring any change to the synthesis definitions in a later chapter.

There are many modules in the system, some simply wraparounds to choose between other modules. However the basic modules used for text-to-speech have the basic following function

`Token_POS`

: basic token identification, used for homograph disambiguation

`Token`

: Apply the token to word rules building the `Word` relation.

`POS`

: A standard part of speech tagger (if desired)

`Phrasify`

Build the `Phrase` relation using the specified method. Various are offered, from statistically trained models to simple CART trees.

`Word`

Lexical look up building the `Syllable` and `Segment` relations and the `SylStructure` related these together.

`Pauses`

Prediction of pauses, inserting silence into the `Segment` relation, again through a choice of different prediction mechanisms.

`Intonation`

Prediction of accents and boundaries, building the `IntEvent` relation and the `Intonation` relation that links IntEvents to syllables. This can easily be parameterized for most practical intonation theories.

`PostLex`

Post lexicon rules that can modify segments based on their context. This is used for things like vowel reduction, contractions, etc.

`Duration`

Prediction of durations of segments.

`Int_Targets`

The second part of intonation. This creates the `Target` relation representing the desired F0 contour.

`Wave_Synth`

A rather general function that in turn calls the appropriate method to actually generate the waveform.

Utterance access

A set of simple access methods exist for utterances, relations, items and features, both in Scheme and C++. As much as possible these access methods are as similar as possible.

As the users of this document will primarily be accessing utterance via Scheme we will describe the basic Scheme functions available for access and give some examples of idioms to achieve various standard functions.

In general the required arguments to a lisp function are reflected in the first parts of the name of the function. Thus `item.relation.next` requires an item, and relation name and will return the next item in that named relation from the given one.

A listing a short description of the major utterance access and manipulation functions is given in the Festival manual.

An important notion to be aware of is that an item is always viewed through so particular relation. For example, assuming a typically utterance called `utt1`.

```
(set! seg1 (utt.relation.first utt1 'Segment))
```

`seg1` is an item viewed from the `Segment` relation. Calling `item.next` on this will return the next item in the `Segment` relation. A `Segment` item may also be in the

`SylStructure` item. If we traverse it using next in that relation we will hit the end when we come to the end of the segments in that syllable.

You may *view* a given item from a specified relation by requesting a view from that. In Scheme `nil` will be returned if the item is not in the relation. The function `item.relation` takes an item and relation name and returns the item as view from that relation.

Here is a short example to help illustrate the basic structure.

```
(set! utt1 (utt.synth (Utterance Text "A short example.")))
```

The first segment in `utt!` will be silence.

```
(set! seg1 (utt.relation.first utt1 'Segment))
```

This item will be a silence as can shown by

```
(item.name seg1)
```

If we find the next item we will get the schwa representing the indefinite article.

```
(set! seg2 (item.next seg1))
(item.name seg2)
```

Let us move onto the "sh" to illustrate the different between traversing the `Segment` relation as opposed to the `SylStructure`

```
(set! seg3 (item.next seg2))
```

Let use define a function which will take an item, print its name name call next on it *in the same relation* and continue until it reaches the end.

```
(define (toend item)
  (if item
      (begin
       (print (item.name item))
       (toend (item.next item)))))
```

If we call this function on `seg3` which is in the `Segment` relation we will get a list of all segments until the end of the utterance

```
festival> (toend seg3)
"sh"
"oo"
"t"
"i"
"g"
"z"
"aa"
"m"
"p"
"@"
"l"
"#"
nil
festival>
```

However if we first changed the view of seg3 to the `SylStructure` relation we will be traversing the leaf nodes of the syllable structure tree which will terminate at the end of that syllable.

```
festival> (toend (item.relation seg3 'SylStructure)
"sh"
"oo"
"t"
nil
festival>
```

Note that `item.next` returns the item immediately to the next in that relation. Thus it return `nil` when the end of a sub-tree is found. `item.next` is most often used for traversing simple lists through it is defined for any of the structure supported by relations. The function `item.next_item` allows traversal of any relation returning a next item until it has visiting them all. In the simple list case this this equivalent to `item.next` but in the tree case it will traverse the tree in *pre-order* that is it will visit roots before their daughters, and before their next siblings.

Scheme is particularly adept at using functions as first class objects. A typical traversal idiom is to apply so function to each item in a a relation. For example support we have a function *PredictDuration* which takes a single item and assigns a duration. We can apply this to each item in the `Segment` relation

```
(mapcar
 PredictDuration
 (utt.relation.items utt1 'Segment))
```

The function `utt.relation.items` returns all items in the relation as a simple lisp list.

Another method to traverse the items in a relation is use the `while` looping paradigm which many people are more familiar with.

```
(let ((f (utt.relation.first utt1 'Segment)))
  (while f
   (PredictDuration f)
   (set! f (item.next_item f))))
```

If you wish to traverse only the leaves of a tree you may call `utt.relation.leafs` instead of `utt.relation.items`. A leaf is defined to be an item with no daughters. Or in the `while` case, there isn't standardly defined a `item.next_leaf` but code easily be defined as

```
(define (item.next_leaf i)
  (let ((n (item.next_item i)))
   (cond
    ((null n) nil)
    ((item.daughters n) (item.next_leaf n))
    (t n))))
```

Features as pathnames

Rather than explicitly calling a set of functions to find your way round an utterance we also allow access through a linear flat *pathname* mechanism. This mechanism is read-only but can succinctly access not just features on a given item but features on related items too.

For example rather than calling an explicit next function to find the name of the following item thus

```
(item.name (item.next i))
```

You can access it via the pathname

```
(item.feat i "n.name")
```

Festival will interpret the feature name as a pathname. In addition to traversing the current relation you can switch between relations via the element `R:`*relationname*. Thus to find the stress value of an segment item `seg` we need to switch to the `SylStructure` relation, find its parent and check the `stress` feature value.

```
(item.feat seg "R:SylStructure.parent.stress")
```

Feature pathnames make the definition of various prediction models much easier. CART trees for example simply specify a pathname as a feature, dumping features for training is also a simple task. Full function access is still useful when manipulation of the data is required but as most access is simply to find values pathnames are the most efficient way to access information in an utterance.

Access idioms

For example suppose you wish to traverse each segment in an utterance replace all vowels in unstressed syllables with a schwa (a rather over-aggressive reduction strategy but it servers for this illustrative example.

```
(define (reduce_vowels utt)
 (mapcar
  (lambda (segment)
   (if (and (string-equal "+" (item.feat segment "ph_vc"))
            (string-equal
             "1" (item.feat segment "R:SylStructure.parent.stress")))
       (item.set_name segment "@")))
  (utt.relation.items 'Segment)))
```

Utterance building

As well as using Utterance structures in the actual runtime process of converting text-to-speech we also use them in database representation. Basically we wish to build utterance structures for each utterance in a speech database. Once they are in that structure, as if they had been (correctly) synthesized, we can use these structures for training various models. For example given the actually durations for the segments in a speech database and utterance structures for these we can dump the actual durations and features (phonetic, prosodic context etc.) which we feel influence the durations and train models on that data.

Obviously real speech isn't as clean as synthesized speech so its not always easy to build (reasonably) accurate utterances for the real utterances. However here we will itemize a number of functions that will make the building of utterance from real speech easier. Building utterance structures is probably worth the effort considering how easy it is to build various models from them. Thus we recommend this even though at first the work may not immediately seem worthwhile.

In order to build an utterance of the type used for our English voices (and which is suitable for most of the other languages we have done), you will need label files for the following relations. Below we will discuss how to get these labels, automatically, by hand or derived from other label files in this list and the relative merits of such derivations.

The basic label types required are

`Segment`

segment labels with (near) correct boundaries, in the phone set of your language.

`Syllable`

Syllables, with stress marking (if appropriate) whose boundaries are closely aligned with the segment boundaries.

`Word`

Words with boundaries aligned (close) to the syllables and segments. By *words* we mean the things which can be looked up in a lexicon thus "*1986*" would not be considered a word and should be rendered as three words "*nineteen eighty six*".

`IntEvent`

Intonation labels aligned to a syllable (either within the syllable boundary or explicitly naming the syllable they should align to. If using ToBI (or some derivative) these would be standard ToBI labels, while in something like Tilt these would be "*a*" and "*b*" marking accents and labels.

`Phrase`

A name and marking for the end of each prosodic phrase.

`Target`

The mean F0 value in Hertz at the mid-point of each segment in the utterance.

Segment labels are probably the hardest to generate. Knowing what phones are there can only really be done by actually listening to the examples and labeling them. Any automatic method will have to make low level phonetic classifications which machines are not particularly good at (nor are humans for that matter). Some discussion of autoaligning phones is given in the diphone chapter where an aligner distributed with this document is described. This may help but as much depends on the segmental accuracy getting it right ultimately hand correction at least is required. We have used that aligner on a speech database though we already knew from another (not so accurate) aligner what the phone sequences probably were. Our aligner improved the quality of exist labels and the synthesizer (phonebox) that used it, but there are external conditions that made this a reasonably thing to do.

Word labeling can most easily be done by hand, it is much easier than to do than segment labeling. In the continuing process of trying to build automatic labelers for databases we currently reckon that word labeling could be the last to be done automatically. Basically because with word labeling, segment, syllable and intonation labeling becomes a much more constrained task. However it is important that word labels properly align with segment labels even when spectrally there may not be any real boundary between words in continuous speech.

Syllable labeling can probably best be done automatically given segment (and word) labeling. The actual algorithm for syllabification may change but whatever is chosen (or defined from a lexicon) it is important that that syllabification is consistently used throughout the rest of the system (e.g. in duration modeling). Note that automatic techniques in aligning lexical specifications of syllabification are in their nature inexact. There are multiple acceptable ways to say words and it is relatively important to ensure that the labeling reflects what is actually there. That is simply looking up a word in a lexicon and aligning those phones to the signal is not necessarily correct. Ultimately this is what we would like to do but so far we have discovered our unit selection algorithms are nowhere near robust enough to do this.

The Target labeling required here is a single average F0 value for each segment. This currently is done fully automatically from the signal. This is naive and a better representation of F0 could be more appropriate, it is used only in some of the model building described below. Ultimately it would be good if the F0 need not be explicitly used at all but just use the factors that determine the F0 value, but this is still a research topic.

Phrases could potentially be determined by a combination of F0 power and silence detection but the relationship is not obvious. In general we hand label phrases as part of the intonation labeling process. Realistically only two levels of phrasing can reliably be labeled, even though there are probably more. That is, roughly, sentence internal and sentence final, what ToBI would label as (2 or 3) and 4. More exact labelings would be useful.

For intonation events we have more recently been using Tilt accent labeling. This is simpler than ToBI and we feel more reliable. The hand labeling part marks `a` (for accent) and `b` for boundary. We have also split boundaries into `rb` (rising boundary) and `fb` (falling boundary). We have been experimenting with autolabeling these and have had some success but that's still a research issue. Because there is a well defined and fully automatic method of going from a/b labeled waveforms to a parameterization of the F0 contour we've found Tilt the most useful Intonation labeling. Tilt is described in [taylor00a].

ToBI accent/tone labeling [silverman92] is useful too but time consuming to label. If it exists for the database then its usually worth using.

In the standard Festival distribution there is a festival script `festival/examples/make_utts` which will build utterance structures from the labels for the six basic relations.

This function can most easily be used given the following directory/file structure in the database directory. `festival/relations/` should contain a directory for each set of labels named for the utterance relation it is to be part of (e.g. `Segment/`, `Word/`, etc.

The constructed utterances will be saved in `festival/utts/`.

Extracting features from utterances

Many of the training techniques that are described in the following chapters extract basic features (via pathnames) from a set of utterances. This can most easily be done by the `festival/examples/dumpfeats` Festival script. It takes a list of feature/pathnames, as a list or from a file and saves the values for a given set of items in a single feature file (or one for each utterance). Call `festival/examples/dumpfeats` with the argument `-h` for more details.

For example suppose for all utterances we want the segment duration, its name, the name of the segment preceding it and the segment following it.

```
dumpfeats -feats '(segment_duration name p.name n.name)' \
   -relation Segment -output dur.feats festival/utts/*.utt
```

If you wish to save the features in separate files one for each utterance, if the output filename contains a "%s" it will be filled in with the utterance fileid. Thus to dump all features named in the file `duration.featnames` we would call

```
dumpfeats -feats duration.featnames -relation Segment \
      -output feats/%s.dur festival/utts/*.utt
```

The file `duration.featnames` should contain the features/pathnames one per line (without the opening and closing parenthesis.

Other features and other specific code (e.g. selecting a voice that uses an appropriate phone set), can be included in this process by naming a scheme file with the `-eval` option.

The dumped feature files consist of a line for each item in the named relation containing the requested feature values white space separated. For example

```
0.399028 pau 0 sh
0.08243 sh pau iy
0.07458 iy sh hh
0.048084 hh iy ae
0.062803 ae hh d
0.020608 d ae y
0.082979 y d ax
0.08208 ax y r
0.036936 r ax d
0.036935 d r aa
0.081057 aa d r
...
```

Notes

1. http://www.cstr.ed.ac.uk/projects/festival/
2. http://www.cstr.ed.ac.uk
3. http://www.ed.ac.uk/

Chapter 4. Basic Requirements

This section identifies the basic requirements for building a voice in a new language, and adding a new voice in a language already supported by Festival.

Hardware/software requirements

Because we are most familiar with a Unix environment the scripts, tools etc. assume such a basic environment. This is not to say you couldn't run these scripts on other platforms as many of these tools are supported on platforms like WIN32, its just that in our normal work environment, Unix is ubiquitous and we like working in it. Festival also runs on Win32 platforms.

Much of the testing was done under Linux; wherever possible, we are using freely available tools. We are happy to say that no non-free tools are required to build voices, and we have included citations and/or links to everything needed in this document.

We assume Festival 1.4.3 and the Edinburgh Speech Tools 1.2.3.

Note that we make an extensive use of the Speech Tools programs, and you will need the full distribution of them as well as Festival, rather than the run-time (binary) only versions which are available for some Linux platforms. If you find the task of compiling Festival and the speech tools daunting, you will probably find the rest of the tasks specified in this document more so. However, it is not necessary for you to have any knowledge of C++ to make voices, though familiarity with text processing techniques (e.g. awk, sed, perl) will make understanding the examples given much easier.

We also assume a basic knowledge of Festival, and of speech processing in general. We expect the reader to be familiar with basic terms such as *F0*, *phoneme*, and *cepstrum*, but not in any real detail. References to general texts are given (when we know them to exist). A basic knowledge of programming in Scheme (and/or Lisp) will also make things easier. A basic capability in programming in general will make defining rules, etc., much easier.

If you are going to record your own database, you will need recording equipment: the higher quality, the better. A proper recording studio is ideal, though may not be available for everyone. A cheap microphone stuck on the back of standard PC is not ideal, though we know most of you will end up doing that. A high quality sound board, close-talking, high quality microphone and a nearly soundproof recording environment will often be the compromise between these two extremes.

Many of the techniques described in here require a fair amount of processing time to achieve, though machines are indeed getting faster and this is becoming less of an issue. If you use the provided aligner for labeling diphones you will need a processor of reasonable speed, likewise for the various training techniques for intonation, duration modeling and letter-to-sound rules. Nothing presented here takes weeks though a number of processes may be over-night jobs, depending on the speed of your machine, and size of your database.

Also we think that you will need a little patience. The process of building a voice is not necessarily going to work first time. It may even fail completely, so if you don't expect anything special, you wont be disappointed.

Voice in a new language

The following list is a basic check list of the core areas you will need to provide pieces for. You may, in some cases, get away with very simple solutions (e.g. fixed phone durations), or be able to borrow from other voices/languages, but whatever you end up doing, you will need to provide something for each part.

You will need to define

- Phone set
- Token processing rules (numbers etc)
- Prosodic phrasing method
- Word pronunciation (lexicon and/or letter-to-sound rules)
- Intonation (accents and F0 contour)
- Durations
- Waveform synthesizer

Voice in an existing language

The most common case is when someone wants to make their own voice into a synthesizer. Note that the issues in voice modeling of a particular speaker are still open research problems. Much of the quality of a particular voice comes mostly from the waveform generation method, but other aspects of a speaker such as intonation and duration, and pronunciation are all part of what makes that person's voice sound like them. All of the general-purpose voices we have heard in Festival sound like the speaker they were record from (at least as far as we know all the speakers), but they also don't have all the qualities of that person's voice, though they can be quite convincing for limited-domain synthesizers.

As a practical recommendation to make a new speaker in an existing supported language, you will need to consider

- Waveform synthesis
- Speaker specific intonation
- Speaker specific duration

Chapter 20 deals with specifically building a new US or UK English voice. This is a relatively easy place to start, though of course we encourage reading this entire document.

Another possible solution to getting a new or particular voice is to do voice conversion, as is done at the Oregon Graduate Institute (OGI) [kain98] and elsewhere. OGI have already released new voices based on this conversion and may release the conversion code itself, though the license terms are not the same as those of Festival or this document.

Another aspect of a new voice in an existing language is a voice in a new dialect. The requirements are similar to those of creating a voice in a new language. The lexicon and intonation probably need to change as well as the waveform generation method (a new diphone database). Although much of the text analysis came probably be borrowed, be aware that simple things like number pronunciation can often change between dialects (cf. US and UK English).

We also do work on limited domain synthesis in the same framework. For limited domain synthesis, a reasonably small corpus is collected, and used to synthesize a much larger range of utterances in the same basic style. We give an example of recording a talking clock, which, although built from only 24 recordings, generates over a thousand unique utterances; these capture a lot of the latent speaker characteristics from the data.

Selecting a speaker

We have found that choosing the right speaker to record, is actually as important as all the the other processes we describe. Some people just have better voices that are better for synthesis than others. In general people with, clearer, more consistent voices are better than others but unfortunately its not as clear as that. Professional speakers are in general better for synthesis that non-professional. Though not all professional voices work, and many non-professional speakers give good results.

In general you are looking for clear speakers, who don't mumble and don't have any speech impediments. It helps if they are aware of speech technology, i.e. have some vague idea of what a phoneme is. A consistent deliver is important. As different parts of speech from different parts of the recorded database are going to be extracted and put together you what the speech to be as consistent as possible. This is usually the quality that professional speakers have (or any one used to public speaking). Also note most people can't actually talk for long periods without practice. Lectures/Teachers are actually much more used to this than students, though this ability can be learned quite easily.

Note choosing the right speaker, if its important to you, can be a big project. For example, an experiment done at AT&T to select good speakers for synthesis involved recording fair sized databases for a number of professional speakers (20 or so) and building simple synthesis example from their voice and submitting these to a large number of human listeners to get them to evaluate quality [syrdal??]. Of course most of us don't have the resources to do searches like that but it is worth taking a little time to think of the best speaker before investing the considerable time in takes in building a speaker.

Note that trying to capture a particular voice is still somewhat difficult. You will always capture some of that persons voice but its unlikely a synthesizer built from recordings of a person will always sound just like that person. However you should note that voice you think are distinctive may be so because of lots variation. For example Homer Simpson's voice is distinctive but it would be difficult to built a synthesizer from. The Comic Book Guy (also from the Simpsons) also has a very distinctive voice but is much less varied prosodically than Homer's and hence it iss likely to be easier to build a synthesizer from his voice. Likewise, Patrick Stewart's voice should be easier than Jim Carey's.

However as it is usually the case that you just have to take any speaker you have willing to do it (often yourself), there are still things you should do that will help the quality. It is best is recording is done in the same session, as it is difficult to set up the same recording environment (even when you are very careful). We recommend recording some time in the morning (not immediately you get up), and if you must re-record do so at the same time of day. Avoid recording when the speaker has a cold, or a hangover as it can be difficult to recreate that state if multiple sessions are required.

Who owns a voice

It is very important that your speaker and you understand the legal status of the recorded database. It is very wise that the speaker signs a statement before you start recording or at least talk to them ensuring they understand what you want to do with the data and what restrictions if any they require. Remember in recording their voice you are potentially allowing anyone (who gets access to the database) to fake that person's voice. The whole issue of building a synthetic voice from recordings is still actually an uninvestigated part of copyright but there are clear ways to ensure you wont be caught out by a law suit, or a disgruntled subject later.

Explain what you going to do with the database. Get the speaker to agree to the level use you may make of the recordings (and any use of them). This will roughly be:

- free for any use
- free to distribute to anyone but cannot be used for commercial purposes without further contract.
- research use only (does this allow public demos?)
- fully proprietary

You must find out what the speaker agrees to before you start spending your time recording. There is nothing worse than spending weeks on building a good voice only to discover that you don't have rights to do anything with it.

Also, don't lie to the speaker make it clear, what it means if their voice is to be released for free. If you release the voice on the net (as we do with our voices), *anyone* may use it. It could be used anywhere, from reading porn stories to emergency broadcast systems. Also note that effectively building a voice from a synthesizer means that the person will no longer be able to use voice id systems as a password protection (actually that depends on the type of voice id system). However also reassure them that these extremes are very unlikely and actually they will be contributing to world of speech science and people will use their voice because they like it.

We (KAL and AWB) have already given up the idea that our voices are in anyway ours and have recorded databases and made them public (even though AWB has a funny accent). When recording others we ensure they understand the consequences and get them to explicitly sign a license that gives us (and/or our institution) the rights to do anything they wish, but the intention is the voice will be released for free without restriction. From our point of view, having no restrictions is by far the easiest. We also give (non-exclusive) commercial rights to the voice to the speaker themselves. This actually costs us nothing, and given most of our recorded voices are for free the speaker could re-release the free version and use it commercially (as can anyone else) but its nice that the original license allows the speaker direct commercial rights (none that I know of have actually done anything with those rights).

There may be other factors though. Someone else may be paying for the database so they need to be accommodated on any such license. Also a database may already be recorded under some license and you wish to use it to build a synthetic voice, make sure you have the rights to do this. Its amazing how mainly people record speech databases and don't take into account the fact that someone else may build a general TTS systems from their voice. Its better that you check that have to deal with problems later.

An example of the license we use at CMU is given in the festvox distribution `festvox/src/vox_files/speaker.licence`.

Also note that there are legal aspects to other parts of a synthetic voice the builder must also ensure they have rights to. Lexicons may have various restrictions. The Oxford Advanced Learners' Dictionary that we currently use for UK English voices is free for non-commercial use only, thus effectively imposing the same restriction on the complete voice even though the prosodic models and diphone databases are free. Also be careful you check the rights when building models from existing data. Some databases are free for research only and even data derived from them (e.g. duration models) may not be further distributed. Check at the start, question all pieces of the system to make sure you know who owns what and what restrictions they impose. This process is worth doing at the start of a project so things are always clear.

Recording under Unix

Although the best recording conditions can't probably be achieved recording directly to a computer (under Unix or some other operating systems). We accept that in many cases recording directly to a computer has many conveniences outweighing its disadvantages.

The disadvantages are primarily in quality, the electromagnetic noise generated by a machine is large and most computers have particularly poor shielding of there audio hardware such that noise always gets added to the recorded signal. But there are ways to try to minimize this. Getting better quality sound cards helps, but they can be very expensive. "Professional" sound cards can go for as much as a thousand dollars.

The advantage of using a computer directly is that you have much more control over the recording session and first line processing can down at record-time. In recent years we found the task of transferring the recorded data from DAT tapes to a computer, and splitting them into individual files, even before phonetic, labeled a significantly laborious task, often larger and resource intensive than the rest of the voice building process. So recently we've accept that direct recording to disk using a machine worthwhile, except for voices that require the highest quality (and when we have the money to take more time). This section describes the issues in recording under Unix, though they mostly apply under Windows too if you go that route.

The first thing you'll find out about recording on a computer is that no one knows how to do it, and probably no-one has actually used the microphone on the machine at all before (or even knows if there is a microphone). Although we believe we are living in a multi-media computer age, setting up audio is still a tricky thing and even when it works its often still flakey.

First you want to ensure that audio works on the machine at all. Find out if anyone has actually heard any audio coming from it. Even though there may be an audio board in there, it may not have any drivers installed or the kernel doesn't know about it. In general, audio rarely, "just works" under Linux in spite of people claiming Linux is ready for the desktop. But before you start claiming Windows is better, we've found that audio rarely "just works" there too. Under Windows when it works its often fine, but when it doesn't the general Windows user is much less likely to have any knowledge about how to fix it, while at least in the Linux world, users have more experience in getting recalcitrant devices to come to life.

Its difficult to name products here as the turn over in PC hardware is frantic. Generally newer audio cards wont work and older card do. For audio recording, we *only* require 16bit PCM, and none of the fancy, FM synthesizers and wavetable devices, those are irrelevant and often make card difficult or very hard to use. Laptops are particular good for recording, as they generally add less noise to the signal (especially if run on the battery) and they are portable enough to take into a quite place that doesn't have desktop cooling fan running in the background. However sound on Laptops under Unix (Linux, FreeBSD, Solaris etc) is unfortunately even less likely to work, due to leading edge technology and proprietary audio chips. Linux is improving in this area but although we are becoming relatively good at getting audio to work on new machines, its still quite a skill.

In general search the net for answers here. Linux offers both ALSA[1] and the Open Sound drivers[2] drivers which go a long way to help. Note though even when these work there may be other problems (e.g. on one laptop you can have either sound working or suspend working but not both at once).

To test audio you'll need something to modify the basic gain on the audio drivers (e.g. `xmixer` under Linux/FreeBSD, or `gaintool` under Solaris). And you can test audio out with the command (assuming you've set `ESTDIR}`

```
$ESTDIR/bin/na_play $ESTDIR/lib/example_data/kdt_001.wav
```

which should play a US male voice saying "She had your dark suit in greasy washwater all year."

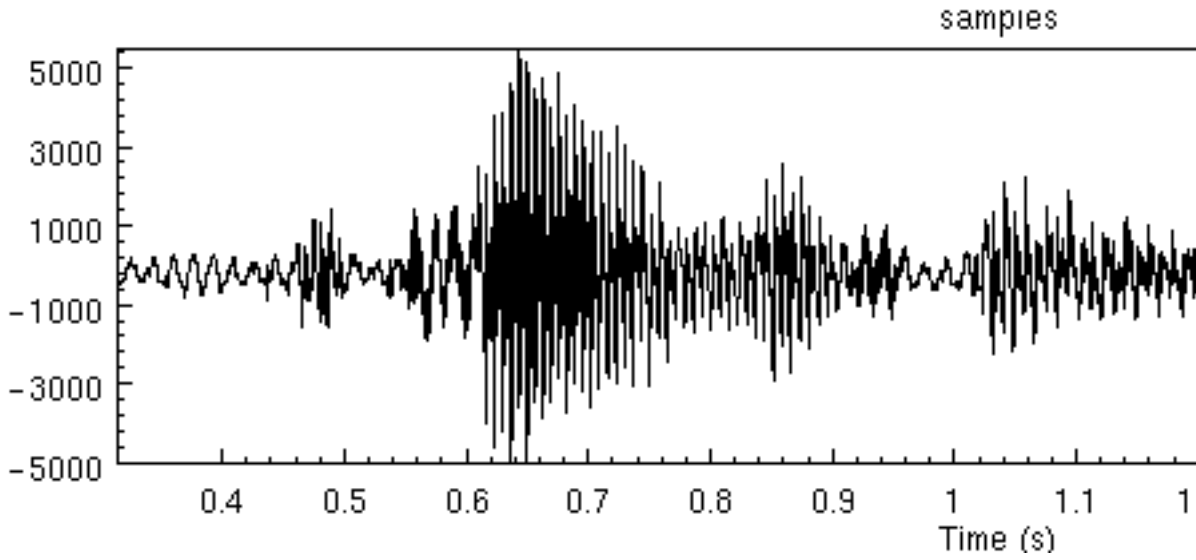
To test audio in you can use the command

```
$ESTDIR/bin/na_record -o file.wav -time 3
```

where the time given is the number of seconds to record. Note you may need to change the microphone levels and/or input gain to make this work.

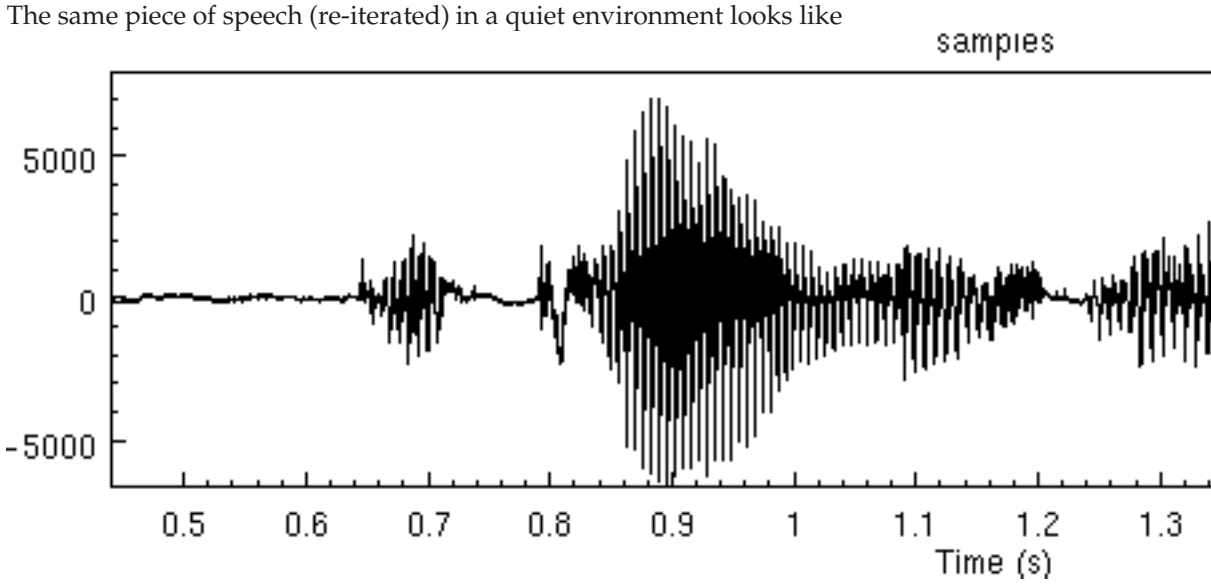
You should *look* at the audio signal as well as listen to it. Its often quite easy to see noise in a signal than hear it. The human ear has developed so that it can mask out noise, but unfortunately not developed enough to mask all noise in synthesis. But in synthesis when we are going to concatenated different parts of the signal the human ear isn't as forgiving.

The following is a recording made with background noise, (probably a computer).



Example waveform recorded with lots of background noise

The same piece of speech (re-iterated) in a quiet environment looks like



Example waveform recorded in clean environment

As you can see the quiet parts of the speech are much quieter in the clean case than the noise case.

Under Linux there is a *Audio-Quality-HOWTO* document that helps get audio up and running. AT time of writting it can be found http://audio.netpedia.net/aqht.html

Extracting pitchmarks from waveforms

Although never as good as extracting pitchmarks from an EGG signal, we have had a fair amount of success in extracting pitchmarks from the raw waveform. This area is somewhat a research area but in this section we'll give some general pointers about how to get pitchmarks form waveforms, or if not at least be able to tell if you are getting reasonable pitchmarks from waveforms or not.

The basic program which we use for the extraction is `pitchmark` which is part of the Speech Tools distribution. We include the script `bin/make_pm_wave` (which is copied by ldom and diphone setup process). The key line in the script is

```
$ESTDIR/bin/pitchmark tmp$$.wav -o pm/$fname.pm -otype est \
  -min 0.005 -max 0.012 -fill -def 0.01 -wave_end \
  -lx_lf 200 -lx_lo 51 -lx_hf 80 -lx_ho 51 -med_o 0
```

This program filters in incoming waveform (with a low and a high band filter, then uses autocorellation to find the pitch mark peaks with the min and max specified. Finally it fills in the unvoiced section with the default pitchmarks.

For debugging purposes you should remove the `-fill` option so you can see where it is finding pitchmarks. Next you should modify the min and max values to fit the range of your speaker. The defaults here (0.005 and 0.012) are for a male speaker in about the range 200 to 80 Hz. For a female you probably want values about 0.0033 and 0.7 (300Mhz to 140Hz).

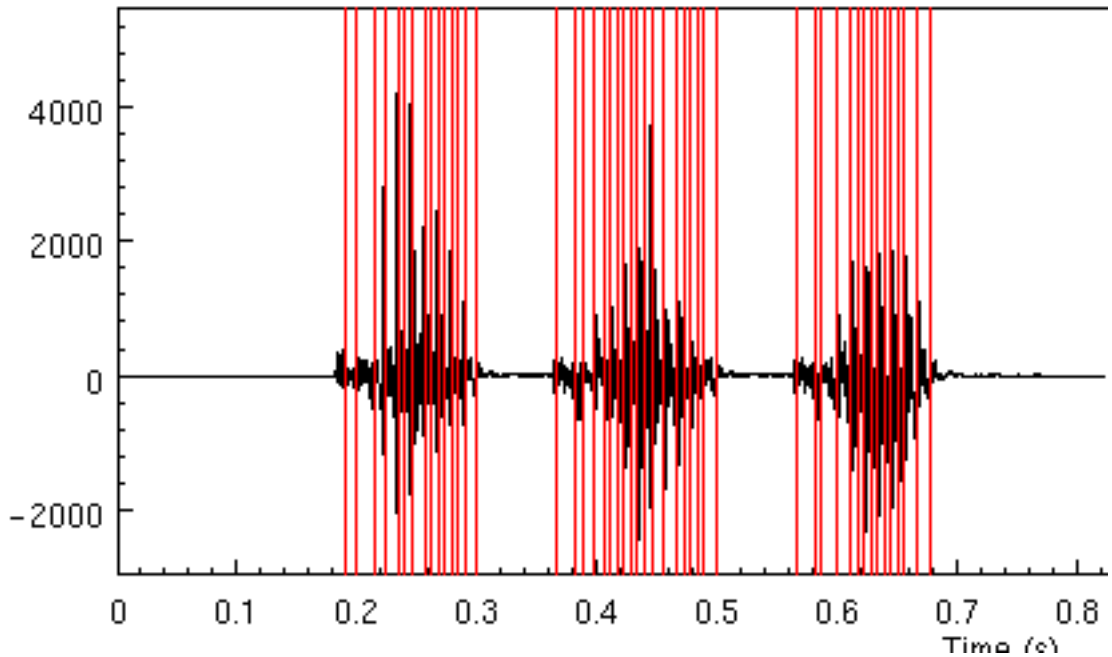
Modify the script to your approximate needs, and run it on a single file, then run the script that translates the pitchmark file into a labeled file suitable for emulabel

```
bin/make_pm_wave wav/awb_0001.wav
bin/make_pm_pmlab pm/awb_0001.pm
```

You can the display the pitchmark with

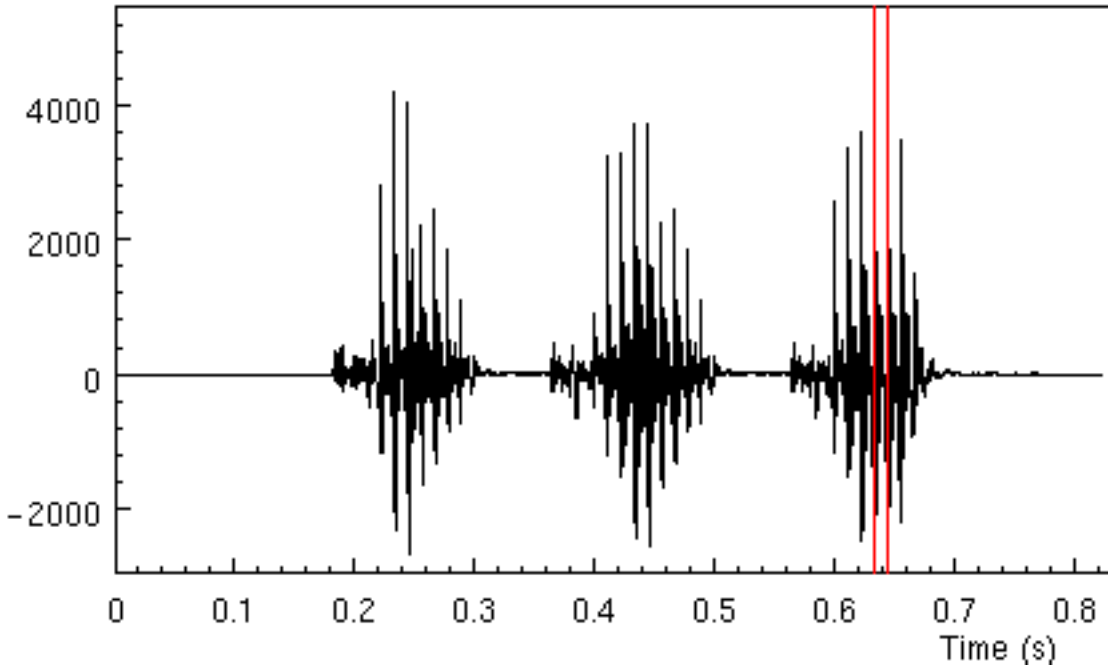
```
emulabel etc/emu_pm awb_0001
```

This should should a number of pitchmarks over the voiced sections of speech. If there are none, or very few it definitely means the parameters are wrong. For example the above parameters on this file `taataataa` properly find pitchmarks in the three vowel sections



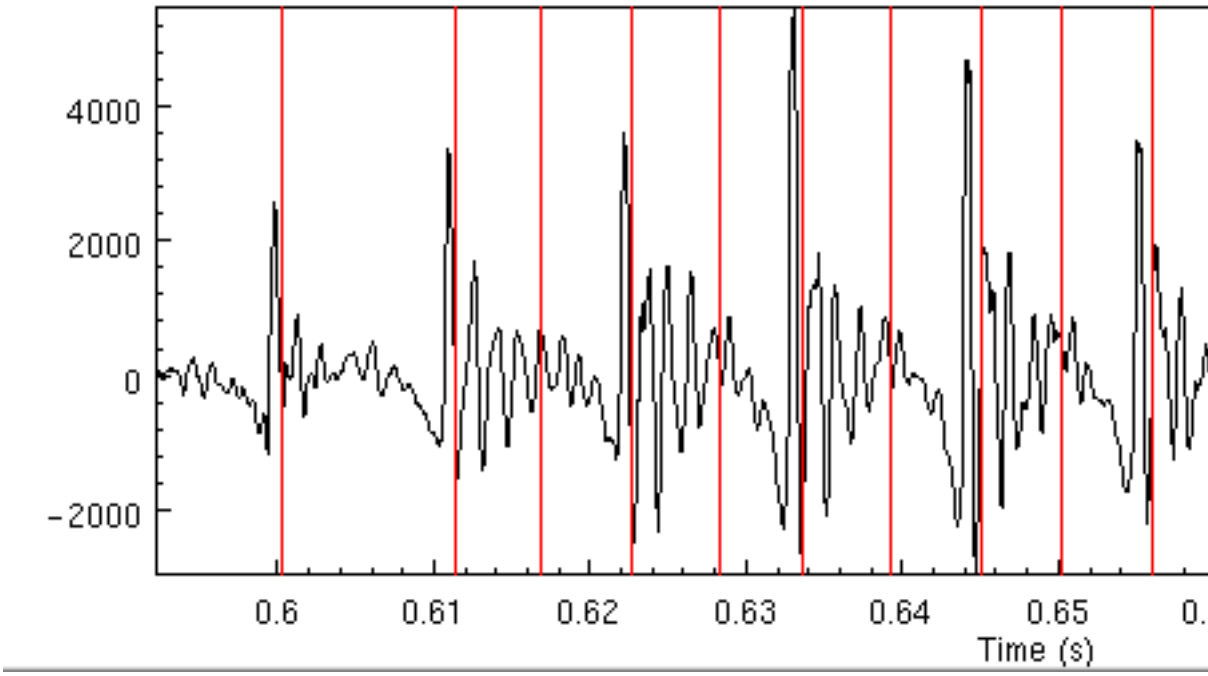
Pitchmarks in waveform signal

It the high and low pass filter values `-lx_lf 200 -lx_hf 80` are in appropriate for the speakers pitch range you may get either too many, or two few pitch marks. For example if we change the 200 to 60, we find only two pitch marks in the third vowel.



Bad pitchmarks in waveform signal

If we zoom in our first example we get the following

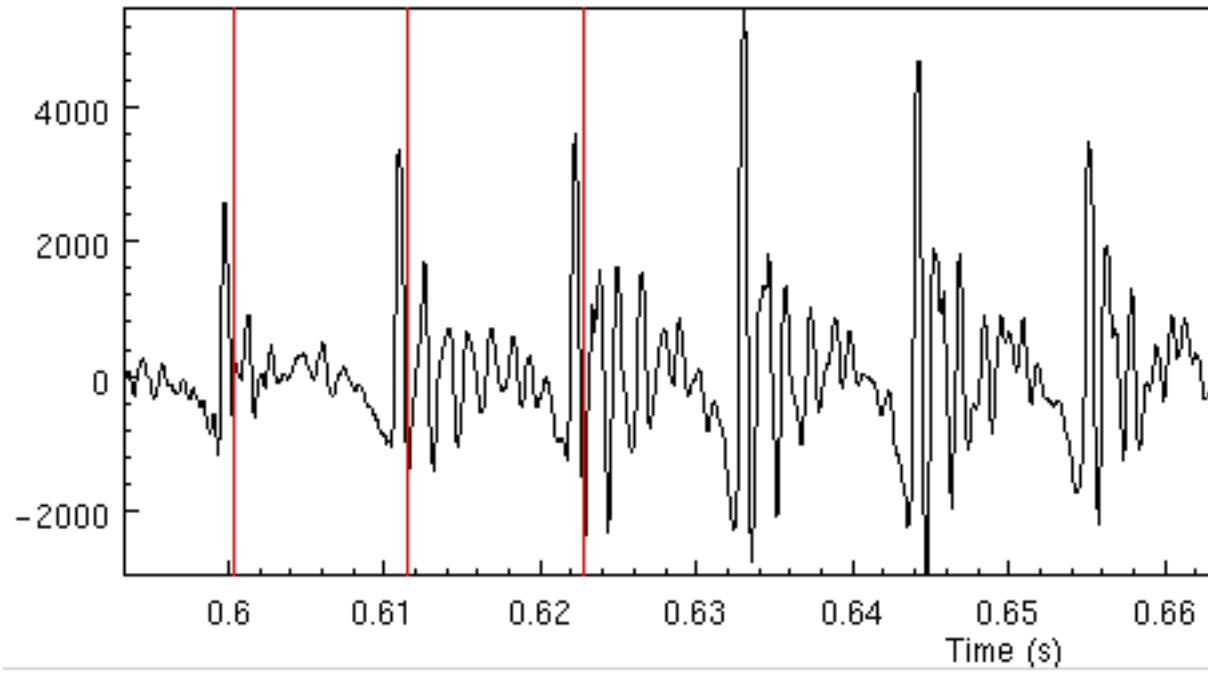


Close-up of pitchmarks in waveform signal

The pitch marks should be aligned to the largest (above zero) peak in each pitch period. Here we can see there are too many pitchmarks (effectively twice as many). The pitchmarks at 0.617, 0.628, 0.639 and 0.650 are extraneous. This means our pitch range is too wide. If we rerun changing the min size, and the low frequency filter

```
$ESTDIR/bin/pitchmark tmp$$.wav -o pm/$fname.pm -otype est \
  -min 0.007 -max 0.012 -fill -def 0.01 -wave_end \
  -lx_lf 150 -lx_lo 51 -lx_hf 80 -lx_ho 51 -med_o 0
```

We get the following

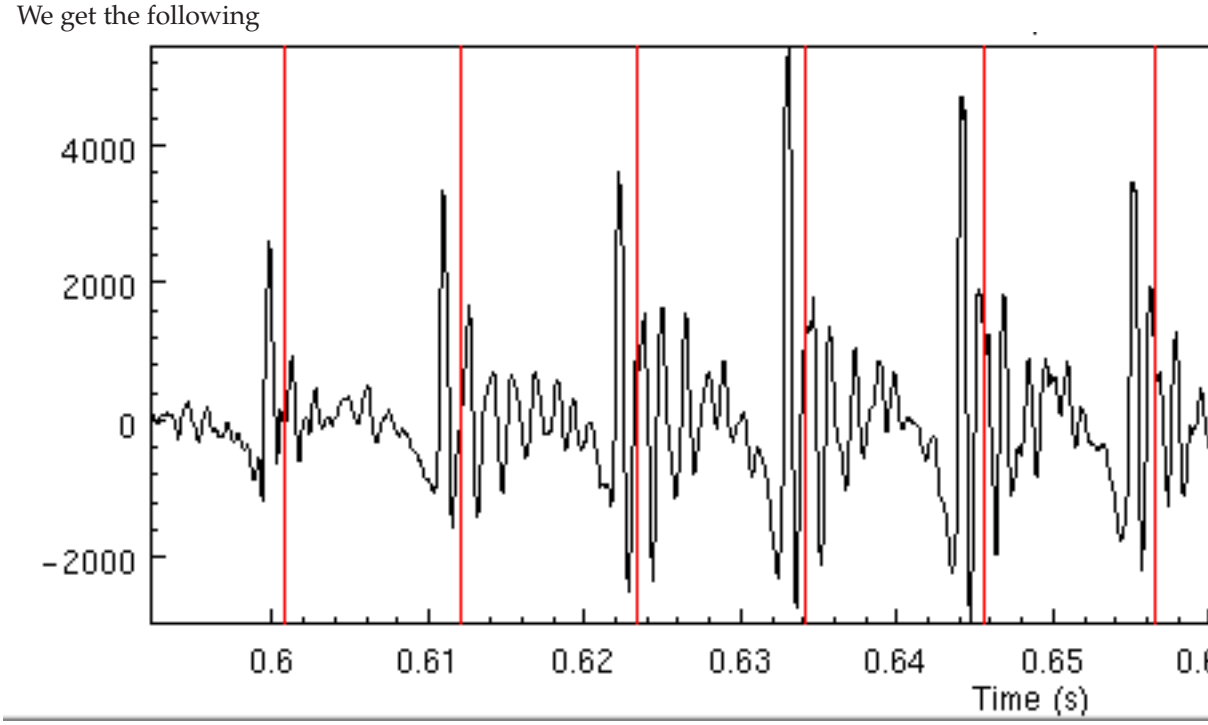


Close-up of pitchmarks in waveform signal (2)

Which is better but its now missing pitchmarks towards the end of the vowel, at 0.634, 0.644 and 0.656. Giving more range for the min (0.005) gives slight better results, but still we get bad pitchmarks. The double pitch mark problem can be lessened by not only changing the range but also the amount order of the high and low pass filters (effectively allowing more smoothing). Thus when secondary pitchmarks appear increasing the `-lx_lo` parameter often helps

```
$ESTDIR/bin/pitchmark tmp$$.wav -o pm/$fname.pm -otype est \
  -min 0.005 -max 0.012 -fill -def 0.01 -wave_end \
  -lx_lf 150 -lx_lo 91 -lx_hf 80 -lx_ho 51 -med_o 0
```

We get the following



Close-up of pitchmarks in waveform signal (3)

This is satisfactory this file and probably for the whole databases of that speaker. Though it is worth checking a few other files to get he best results. Note the by increasing the order of the filer the pitchmark creep forward (which is bad).

If you feel brave (or are desperate) you can actually edit the pitchmarks yourself with emulabel. We have done this occasionally especially when we find persistent synthesis errors (spikes etc). You can convert a pm_lab file back into its pitchmark format with

```
bin/make_pm_pmlab pm_lab/*.lab
```

An post-processing step is provided that moves the predicted pitchmarks to the nearest waveform peak. We find this useful for both EGG extracted pitchmarks and waveform extracted ones. A simple script is provided for this

```
bin/make_pm_fix pm/*.pm
```

If you pitchmarks are aligning to the largest troughs rather than peaks your signal is upside down (or you are erroneously using `-inv`. If you are using `-inv`, don't, if you are not, then invert the signal itself with

```
for i in wav/*.wav
do
  ch_wave -scale -1.0 $i -o $i
done
```

Note the above are quick heuristic hacks we have used when trying to get pitchmarks out of wave signals. These require more work to offer a more reliable solution, which we know exists. Extracting (fixed frame) LPC coefficients and extracting a residual, then extracting pitchmarks could give a more reliable solution but although all these tools are available we have not experimented with that yet.

Notes

1. http://www.alsa-project.org/
2. http://www.opensound.com/
3. http://audio.netpedia.net/aqht.html

Chapter 5. Limited domain synthesis

This chapter discusses and gives examples of building synthesis systems for limited domains. By limited domain, we mean applications where the speech output is constrained. Such domains may still be infinite but they may be target to specific vocabulary and phrases. In fact with today's current speech system such limited domain applications are in fact the most common. Some typical examples are telling the time, reading telephone numbers. However from experience we can see that this technique can be extended to include more general information giving systems and dialog systems, such as reading the weather, or even the DARPA Communicator domain (flight information dialog system).

Limited domains are discussed here as it is felt that it should be easier to build unit selection type synthesizers for domains where there are a much smaller and controlled number of units. The second reason is that general TTS systems (e.g. diphone systems) still sound synthetic. General unit selection when its good, offers near human quality, but when its bad it is usually much worse than a diphone synthesizer. Hybrid systems look interesting but as we cannot yet automatically detect when general unit selection systems go bad, its not clear when a diphone system should be swapped in. But as unit selection offers so much promise, it is hoped that in a limited domain we can get the unit selection good quality, and avoid the bad quality. Finally, although full TTS systems may be our ultimate goal actually for many existing systems a limited domain synthesizer is adequate.

There is a stage beyond limited domain, but falling short of general one synthesis, where the most common phrases are the best synthesized and the quality gracefully degrades as the phrases become less common. Some hybrid recorded prompts/unit selection/diphone systems have been proposed and should be able to deliver and answer but we will not deal directly with those here.

However one point you quickly find is that although most speech dialog systems are very constrained in their vocabulary many require the hardest class of words: proper names.

In continuing in mode of tutorial this chapter first gives a complete walkthrough of a talking clock. This is a small example which will probably work. Following through this example will give you a good idea of what is involved in building a limited domain synthesizer. Also in the following section problems and modifications can be better discussed with respect to this complete example.

designing the prompts

To get a good limited domain synthesizer, it is important to understand what is going on so you can properly tailor your application to take proper advantage of what can be good, and to avoid the limitations these methods impose.

Note that you may wish to change your application to take better advantage of this by making its output forms more regular, or at least use a smaller vocabulary. As a first basic approximation the techniques here require that the training contain all words which are to be actually synthesized. Therefore if a word does not appear in the training data it *cannot* be synthesized. We do provide fall back positions, using a diphone voice, but that will always be worse than the more natural unit selection synthesis. Often this isn't much of a restriction, or you can tailor your application to avoid having a large vocabulary. For example if you are going to build a system for reading weather reports you can make the weather reports not actually name the city/town they refer to and just use phrases like "This city ..." and depend of context for the user to know which actually city is being talked about.

Of course many speech applications have limited vocabularies in all but a very few places. Proper names such as places, people, movie names, etc are in general complete open classes. Building a speech application around those aspects isn't easy and may make a limit domain synthesizer just impractical. But it should be noted that

those open classes are also the classes that more general synthesizers will often fail on too. Some hybrid system may better solve that, which we will not really deal with here.

For almost closed class, recording and modify the data may be a solution but we have not yet got enough experience to comment on this yet but we feel that may be a reasonable compromise.

The most difficult part of building a limited domain synthesizer is designed the database to record that best covers what you wish the synthesizer to say. Sometimes this is fairly easy in that you wish the synthesizer to simple read utterances in a very standard form where slot will be filled with varying values (such as, dates numbers etc.) Such as

The area code you require is NUMBER NUMBER NUMBER.

The prompts can be devised to fill in values for each of the NUMBER variables.

More complex utterance can still be viewed in this way

The weather at TIME on DATE: outlook OUTLOOK, NUMBER degrees.

But once we move into more general dialog its appears initially harder to properly find the utterances that cover the domain.

The first important observation to make is that in such systems where limited domain synthesis is practical the phrases to be spoken are almost certainly generated by a computer. That is there exists an explicit function which the language generated by the applications. In some case this will take the form of an explicit grammar. In this case we can use that grammar to generate phrase language and then select from them utterance that adequately cover the domain. However even when there is an explicit grammar it usually will not allow explicitly encode the frequency of each generated utterance. As we wish to ensure that the most common phrases are synthesized best we ideally need to know which utterances are to be synthesized most often to properly select which utterance to record.

Where a system is already running with a standard synthesizer it is possible to record what is currently being said and how often. We can then use such logs of current system usage to select which utterance should be in our set of prompts to record.

In practice you will be design the system's output at the same time as the limited domain synthesizer so some combination, and guessing of the frequency and cover will be necessary.

In general you should design your databases to have at least 2 (and probably 5) examples of each word in you vocabulary. Secondly you should select utterances that maximise bi-gram coverage. That is try to ensure as many different word-word pairings over your corpus. We have used techniques based on these recommendations to greedily select utterances from larger corpora to record.

customizing the synthesizer front end

Once you decided on a set of utterances that appropriately cover the domain you also need to consider how those particular text strings are synthesized. For example if the data contains flight numbers, dates, times etc, you must ensure that festival properly renders those. As we are discussing a limited domain the distribution of token types will be different from standard text but also more constrained so simple changes to the lexicon, token to word rules, etc. will allow properly synthesis of these utterances.

One particular area of customization we have noted is worthwhile is that of phrasing. It seems important to explicitly mark phrasing in the prompts, and have the speaker follow such phrasing as it allows for much better joins in unit selection, as

well as consist prosody from the speaker. Thus in the default code provided below the normal phrasing module in festival is replaced with one that treat punctuation as phrasal markers.

autolabeling issues

We currently use autolabel the recorded prompts using the alignment technique based on [malfrere97] that we discussed above for diphone labeling. Although this technique works its is not as robust for general speech as it is for carefully articulated nonsense words. Specifically this technique does not allow for alternative pronunciations which are common.

Ideally we should use a more generally speech recognition systems. In fact for labeling of unit selection database the bets results would be to train a recognition using Baum-Welch on the database until convergence. This would give the most consistent labeling. We have started initial experiments with using the open source CMU Sphinx recognition system are are likely to provide scripts to do this in later releases.

In the mean time the greatest problem in predict phone list must be the same (or very similar) to what was actually spoken. This can be achieved by a knowledge speaker, and by customizing the front end of the synthesizer so it produces more natural segments.

Two observations are worth mentioning here. First if the synthesizer makes a mistake in pronunciation and the human speaker does not we have found the things may work out anyway. For example we found the word "hwy" appeared in the prompts, which the synthesizer pronounced as "H W Y" while the speaker said "highway". The forced alignment however cause the phones in the pronunciation of the letters "H W Y" to be aligned with the phones in "highway" thus on selection appropriate, though misnamed, phones were selected. However you should *depend* on such compound errors.

The second observation is that festival includes prediction of vowel reduction. We are beginning to feel that such prediction is unnecessary in limited domain synthesizer or in unit selection in general. This vowel variation can itself be achieve but the clustering technique themselves and hence allows a reasonable back-off selection strategy than making firm type decisions.

unit size and type

The basic cluster unit selection code available in festival uses segments as the size of unit. However the acoustic distance measure used in cluster uses significant portions of the previous segment. Thus the cluster unit selection effectively selects diphones from the database.

The type of units the cluster selection code uses is based on the segment name, by default. In the case of limited domain synthesis we have found that constraining this further gives both better, and faster synthesis. Thus we allow for the unit type to be defined by an arbitrary feature. In the default limited domain set up we use

```
SEGMENT_WORD
```

That is, the segment plus the word the segment comes from. Note this doesn't mean we are doing *word* concatenation in our synthesizer. We are still selecting phone units but that the these phone are differentiated depending on the word they come from thus a /t/ from the word "unit" cannot be used to synthesis a /t/ in "table". The primary reason for us doing this was to cut down the search, though it notable improves synthesis quality to. As we have constructed the database to have good coverage this is a practical thing to do.

The feature function `clunit_name` constructs the unit type for a particular segment item. We have provided the above default (segment name plus (downcased) word name), but it is easy to extend this.

In one domain we have worked in we wish to differentiate between words in different prosody contexts. Particularly we wished to mark words us "questionable" so we can ask users for confirmation. To do this we marked the "questionable" words in the prompts with a question mark prefix. We then recorded them with appropriate intonation and then defined our `clunit_name` function in include "C_" is the word was prefixed by a question mark. For example the following two prompts will be read in a different manner

theater is Squirrel Hill Theater
theater is ?Squirrel ?Hill ?Theater

Likewise in unit selection the units in the word "Squirrel" will not be used to synthesize the word "?Squirrel" and vice versa. Although crude, this does give simple control over prosody variation though this technique can require the vocabulary of the units to increase to where this technique ceases to be practical.

It would be good if this technique had a back-off strategy where if no unit can be found for a particular word it would allow other words to contribute candidates. This is ultimately what general unit selection is. We do consider this our goal in unit type but in the interest of building quick and reliable limited domain synthesizers we do not yet do this but consider it an area we will experiment with. One specific area that only partially cross this line is in the synthesis of numbers. It seem very reasonable to allow selection of units from simple numbers (e.g. "seven" and "seventy") but we have not experimented on that yet.

One further important point should be highlights about this method for defining unit types. Although including the word name in the unit name does greatly encourage whole words to be selected it does not mean that joins in the synthesize utterances only occur at word boundaries. It is common that contiguous units are selection from different occurrences of the same word. Mid-word (e.g. within vowels, or at stops) joins at stable places are common. The optimal coupling technique selects the best place within a word for the cross over between two different parts of the database.

using limited domain synthesizers

The goal of building such limited domain synthesizer is not just to show off good synthesis. We followed this route as we see this as a very practical method for building speech output systems.

For practical reasons, the default configures includes the possibility of a back-up voice that will be called to do synthesis if the limited domain synthesizer fails, which for this default setup means the phrase includes a out of vocabulary word. It would perhaps be more useful if the fall position just required synthesis of the out of vocabulary word itself rather than the whole phrase, but that isn't as trivial as it might be. The limit domain synthesis does not prosody modification of the selections, except for pitch smooth at joins, thus slotting in a diphone one word would sound very bad. At present each limited domain synthesizer has an explicitly defined `closest_voice`. This voice is used when the limited domain synthesis fails and also when generating the prompts, which can be looked upon as absolute minimal case when the synthesizer has no data to synthesize from.

There are also issues in speed here, which we are still trying to improve. This technique should in fact be fast but it is still slower than our diphone synthesizer. One significant reason is the cost if finding the optimal join put in selected units. Also this synthesizer technique require more memory that diphones as the cepstrum parameters for the whole database are required at run time, in addition to the full waveforms. These issues we feel can and should be addressed as these techniques are not funda-

mentally computationally expensive so we intend to work on these aspect in later releases.

Telling the time

Festival includes a very simple little script that speaks the current time (@file{festival/examples/saytime}). This section explains how to replace the synthesizer used from this script with one that talks with your own voice. This is an extreme example of a limited domain synthesizer but it is a good example as it allows us to give a walkthrough of the stages involved in building a limited domain synthesizer. This example is also small enough that it can be done in well under an hour.

Following through this example will give a reasonable understanding of the relative importance of many important steps in the voice building process.

The following tasks are required:

- Designing the prompts
- Customized the synthesizer front end
- Recording the prompts
- Autolabeling the prompts
- Building utterance structures for recorded utterances
- Extracting pitchmark and building LPC coefficients
- Building a clunit based synthesizer from the utterances
- Testing and tuning

Before starting set the environment variables `FESTVOXDIR` and `ESTDIR` to the directories which contain the festvox distribution and the Edinburgh Speech Tools respectively. Under bash and other good shells this may be done by commands like

```
export FESTVOXDIR=/home/awb/projects/festvox
export ESTDIR=/home/awb/projects/1.4.3/speech_tools
```

In earlier releases we only offered a command line based method for building voices and limited domain synthesizers. In order to make the process easier and less prone to error we have introduced and graphical front end to these scripts. This front end is called `pointyclicky` (as it offers a pointy-clicky interface). It is particularly useful in the actual prompting and recording. Although `pointyclicky` is the recommend route in the section we go through the process step by step to give a better understanding of what is required and where problems may lie that require attention.

A simple script is provided setting up the basic directory structure and copying in some default parameter files. The festvox distribution includes all the setup for the time domain. When building for your domain, you will need to provide the file `etc/DOMAIN.data` contains your prompts (as described below).

```
mkdir ~/data/time
cd ~/data/time
$FESTVOXDIR/src/ldom/setup_ldom cmu time awb
```

As in the definition of diphone databases we require three identifiers for the voice. These are (loosely) institution, domain and speaker. Use `net` if you feel there isn't an appropriate institution for you, though we have also use the project name that the voice is being build for here. The domain name seems well defined. For speaker name we have also used style as opposed to speaker name. The primary reason for

these to so that people do not all build limited domain synthesizer with the same thus making it not possible to load them into the same instance of festival.

This setup script makes the directories and copies basic scheme files into the `festvox/` directory. You may need to edit these files later.

Designing the prompts

In this `saytime` example the basic format of the utterance is

The time is now, EXACTNESS MINUTE INFO, in the DAYPART.

For example

The time is now, a little after five to ten, in the morning.

In all there are 1152 (4x12x12x2) utterances (although there are three possible day info parts (morning, afternoon and evening) they only get 12 hours, 6 hours and 6 hours respectively). Although it would technically be possible to record all of these we wish to reduce the amount of recording to a minimum. Thus what we actually do is ensure there is at least one example of each value in each slot.

Here is a list of 24 utterances that should cover the main variations.

The time is now, exactly five past one, in the morning
The time is now, just after ten past two, in the morning
The time is now, a little after quarter past three, in the morning
The time is now, almost twenty past four, in the morning
The time is now, exactly twenty-five past five, in the morning
The time is now, just after half past six, in the morning
The time is now, a little after twenty-five to seven, in the morning
The time is now, almost twenty to eight, in the morning
The time is now, exactly quarter to nine, in the morning
The time is now, just after ten to ten, in the morning
The time is now, a little after five to eleven, in the morning
The time is now, almost twelve.
The time is now, just after five to one, in the afternoon
The time is now, a little after ten to two, in the afternoon
The time is now, exactly quarter to three, in the afternoon
The time is now, almost twenty to four, in the afternoon
The time is now, just after twenty-five to five, in the afternoon
The time is now, a little after half past six, in the evening
The time is now, exactly twenty-five past seven, in the evening
The time is now, almost twenty past eight, in the evening
The time is now, just after quarter past nine, in the evening
The time is now, almost ten past ten, in the evening
The time is now, exactly five past eleven, in the evening
The time is now, a little after quarter to midnight.

These examples are first put in the prompt file with an utterance number and the prompt in double quotes like this.

```
(time0001 "The time is now ...")
(time0002 "The time is now ...")
(time0003 "The time is now ...")
...
```

These prompt should be put into `etc/DOMAIN.data`. This file is used by many of the following sub-processes.

Recording the prompts

The best way to record the prompts is to use a professional speaker in a professional recording studio (anechoic chamber) using dual channel (one for audio and the other for the electroglottograph signal) direct to digital media using a high quality head mounted microphone.

However most of us don't have such equipment (or voice talent) so readily available so whatever you do will probably have to be a compromise. The head mounted mike requirement is the cheapest to meet and it is pretty important so you should at least meet that requirement. Anechoic chambers are expensive, and even professional recording studios aren't easy to access (though most Universities will have some such facilities). It is possible to do away with the EGG reading if a little care is taken to ensure pitchmarks are properly extracted from the waveform signal alone.

We have been successful in recording with a standard PC using a standard soundblaster type 16bit audio card though results do vary from machine to machine. Before attempting this you should record a few examples on the PC to see how much noise is being picked up by the mike. For example try the following

```
$ESTDIR/bin/na_record -f 16000 -time 5 -o test.wav -otype riff
```

This will record 5 seconds from the microphone in the machine you run the command on. You should also do this to test that the microphone is plugged in (and switched on). Play back the recorded wave with `na_play` and perhaps play with the mixer levels until you get the least background noise with the strongest spoken signal. Now you should display the waveform to see (as well as hear) how much noise is there.

```
$FESTVOXDIR/src/general/display_sg test.wav
```

This will display the waveform and its spectrogram. Noise will show up in the silence (and other) parts.

There a few ways to reduce noise. Ensure the microphone cable isn't wrapped around other cables (especially power cables). Turning the computer 90 degrees may help and repositioning things can help too. Moving the sound board to some other slot in the machine can also help as well as getting a different microphone (even the same make).

There is a large advantage in recording straight to disk as it allows the recording to go directly into right files. Doing off-line recording (onto DAT) is better in reducing noise but transferring it to disk and segmenting it is a long and tedious process.

Once you have checked your recording environment you can proceed with the build process.

First generate the prompts with the command

```
festival -b festvox/build_ldom.scm '(build_prompts "etc/time.data")'
```

and prompt and record them with the command

```
bin/prompt_them etc/time.data
```

You may or may not find listening to the prompts before speaking useful. Simply displaying them may be adequate for you (if so comment out the `na_play` line in `bin/prompt_them}`.

Autolabeling the prompts

The recorded prompt can be labeled by aligning them against the synthesize prompts. This is done by the command

```
bin/make_labs prompt-wav/*.wav
```

If the utterances are long (> 10 seconds of speech) you may require lots of swap space to do this stage (this could be fixed).

Once labeled you should check that they are labeled reasonable. The labeler typically gets it pretty much correct, or very wrong, so a quick check can often save time later. You can check the database using the command

```
emulabel etc/emu_lab
```

Once you are happy with the labeling you can construct the whole utterance structure for the spoken utterances. This is done by combining the basic structure from the synthesized prompts and the actual times from the automatically labeled ones. This can be done with the command

```
festival -b festvox/build_ldom.scm '(build_utts "etc/time.data")'
```

Extracting pitchmarks and building LPC coefficients

Getting good pitchmarks is important to the quality of the synthesis, see the Section called *Extracting pitchmarks from waveforms* in Chapter 4 for more detailed discussion on extrating pitchmarks from waveforms. For the limited domain synthesizers the pitch extract is a little less crucial that for diphone collection. Though spending a little time on this does help.

If you have recorded EGG signals the you can use `bin/make_pm` from the `.lar` files. Note that you may need to add (or remove) the option `-inv` depending on the updownness of your EGG signal. However so far only the CSTR larygnograph seems to produce inverted signals so the default should be adequate. Also note the parameters that specify the pitch period range, `-min` and `max` the default setting are suitable for a male speaker, for a female you should modify these to something like

```
-min 0.0033 -max 0.0875 -def 0.005
```

The changing from a range of (male) 200Hz-80Hz with a default of 100Hz, to a female range of 300Hz-120Hz and default of 200Hz.

If you don't have an EGG signal you must extract the pitch from the waveform itself. This works though may require a little modification of parameters, and it is computationally more expensive (and wont be as exact as from an EGG signal). There are two methods, one using Entropic's `epoch` program which work pretty well without tuning parameters. The second is to use the free Speech Tools program `pitchmark`. The first is very computationally expensive, and as Entropic is no longer in existence, the program is no longer available (though rumours circulate that it may appear again for free). To use `epoch` use the program

```
bin/make_pm_epoch wav/*.wav
```

To use `pitchmark` use the command

```
bin/make_pm_wave wav/*.wav
```

As with the EGG extraction `pitchmark` uses parameters to specify the range of the pitch periods, you should modify the parameters to best match your speakers range. The other filter parameters also can make a different to the success. Rather than try to explain what changing the figures mean (I admit I don't fully know), the best solution is to explain what you need to obtain as a result.

Irrespective of how you extract the pitchmarks we have found that a post-processing stage that moves the pitchmarks to the nearest peak is worthwhile. You can achieve this by

```
bin/make_pm_fix pm/*.pm
```

At this point you may find that your waveform file is upside down. Normally this wouldn't matter but due to the basic signal processing techniques we used to find the pitch periods upside down signals confuse things. People tell me that it shouldn't happen but some recording devices return an inverted signal. From the cases we've seen the same device always returns the same form so if one of your recordings is upside down all of them probably are (though there are some published speech databases e.g. BU Radio data, where a random half are upside down).

In general the higher peaks should be positive rather than negative. If not you can invert the signals with the command

```
for i in wav/*.wav
do
  ch_wave -scale -1.0 $i -o $i
done
```

If they are upside, invert them and re-run the pitch marking. (If you do invert them it is not necessary to re-run the segment labeling.)

Power normalization can help too. This can be done globally by the function

```
bin/simple_powernormalize wav/*.wav
```

This should be sufficient for full sentence examples. In the diphone collection we take greater care in power normalization but that vowel based technique will be too confused by the longer more varied examples.

Once you have pitchmarks, next you need to generate the pitch synchronous MEL-CEP parameterization of the speech used in building the cluster synthesizer.

```
bin/make_mcep wav/*.wav
```

Building a clunit based synthesizer from the utterances

Building a full clunit synthesizer is probably a little bit of over kill but the technique basically works. See Chapter 12 for a more detailed discussion of unit selection technique. The basic parameter file `festvox/time_build.scm`, is reasonable as a start.

```
festival -b festvox/build_ldom.scm '(build_clunits "etc/time.data")'
```

If all goes well this should create a file `festival/clunits/cmu_time_awb.catalogue` and set of index trees in `festival/trees/cmu_time_awb_time.tree`.

Testing and tuning

To test the new voice start Festival as

```
festival festvox/cmu_time_awb_ldom.scm '(voice_cmu_time_awb_ldom)'
```

The function `(saytime)` can now be called and it should say the current time, or `(saythistime "11:23")`.

Note this synthesizer can *only* say the phrases that it has phones for which basically means it can only say the time in the format given at the start of this chapter. Thus although you can use `SayText` it will only synthesis words that are in the domain. That's what limited domain synthesis is.

A full directory structure of this example with the recordings and parameters files is available at http://festvox.org/examples/cmu_time_awb_ldom/. And an on-line demo of this voice in that directory is available at http://festvox.org/examples/cmu_time_awb_ldom/[2].

Making it better

The above walkthough is to give you a basic idea of the stages involved in building a limited domain synthesizer. The quality of a limited domain synthesizer will most likely be excellent in parts and very bad in others which is typical of techniques like this. Each stage is, of course, more complex than this and there are a number of things that can be done to improve it.

For limited domain synthesize it should be possible to correct the errors such that it is excellent always. To do so though requires being able to diagnose where the problems are. The most likely problems are listed here

- Mis-labeling Due to lipsmacks, and other reasons the labeling may not be correct. The result may the wrong, extra or missing segments in the synthesized utterance. Using `emulabel` you can check and hand correct the labels.
- Mis-spoken data The speaker may have made a mistake in the content. This can often happen even when the speaker is careful. Mistakes can be actual content (it is easy to read a list of number wrongly), but also hesitations and false starts can make the recording bad. Also note that inconsistent prosodic variation can also affect the synthesis quality. Re-recording can be considered for bad examples, or you can delete them from the `etc/LDOM.data` list, assuming there is enough variation in the rest of the examples to ensure proper coverage of the domain.
- Bad pitchmarking Automatic pitchmarking is not really automatic. It is very worthwhile checking to see if it is correct and re-running the pitchmarking with better parameters until it is better. (We need better documentation here on how to know what "correct" is.)
- Looking at the data There is never a substitute for actually looking at the data. Use `emulabel` to actually look at the recorded utterances and see what the labeling is. Ensure these match and files haven't got out of order. Look at a random selection not just the first example.
- Improving the unit clustering The clustering techniques and the features used here are pretty generic and by no means optimal. Even for the simple example given here it is not very good. See Chapter 12 on unit selection for more discussion on this. Adding new features for use in cluster may help a lot.

The line between limited domain synthesis and unit selection is fuzzy. The more complex and varied the phrases you synthesize are, the more difficult it is to produce reliable synthesis.

Notes

1. http://festvox.org/examples/cmu_time_awb_ldom/
2. http://festvox.org/ldom/index.html

Chapter 6. Text analysis

This chapter discusses some of the basic problems in analyzing text when trying to convert it to speech. Although it is oftain considered a trival problem, not worthy of specnding time one, to anyone who has to actually listen to general text-to-speech systems quickly realises it is not as easy to pronounce text as it first appears. Numbers, symbols, acronyms, abbreviations apear to various degrees in different types of text, even the most general types, like news stoires and novels still have tokens which do not have a simple pronunciaiton that can be found merely by looking up the token in a lexicon, or using letter to sound rules.

In any new language, or any new domain that you wish to tranfer text to speech building an apporpriate text analysis module is necessary. As an attempt to define what we mean by text analysis more specifically we will consider this module as taking in strings of characters and producing strings of *words* where we defined words to be items for which a lexicon can provide pronucniations either by direct lookup or by some form of letter to sound rules.

The degree of difficult of this convertion task depends on the text type and language. For example in lanmguages like Chinese, Japanese etc., there is, conventionally, no use of whitespace characaters between words as found in most Western language, thus even identfying the token boundaries is an interesting task. Even in English text the proportion of simple pronouncable words to what we will term *non-standard words* can vary greatly. We define non-standard words (NSWs) to be those tokens which do not apear directly in the lexicon (at least as a first simplication). Thus tokens contains digits, abbreviations, and out of vocabulary words are all considered to be NSWs that require some form of identification before their pronunciation can be specified. Sometimes NSWs are ambiguous and some (often shallow) level of analysis is necessary to identfiy them. For example in English the string of digits `1996` can have several different pronunciations depending on its use. If it is used as a year it is pronunciation as `nineteen ninety-six`, if it is a quantity it is more likely pronuounced as `one thousand nine hundred (and) ninety-six` while if it is used as a telephone extention it can be pronounced simpelas a string of digits `one nine nine six` . Deterimining the appropriate type of expansion is the job of the text analysis module.

Much of this chapter is based on a project that was carried out at a summer workshop at Johns Hopkins University in 1999 [JHU-NSW-99] and later published in [Sproat00], the tools and techniques developed at that workshop were further developed and documented and now distributed as part of the FestVox project. After a discussion of the problem in more detail, concentrating on English examples, a full presentation of NSW text analysis technique will be given with a simple example. After that we will address different appropaches that can be taken in Festival to build general and customized text analysis models. Then we will address a number of specifc problems that appear in text analysis in various languages including homogra[h disambiguation, number pronunciation in Slavic languages, and segmentation in Chinese.

Non-standard words analysis

In an attempt to avoid relying solely on a bunch of "hacky" rules, we can better define the task of analyzing text using a number of statistical trained models using either labeled or unlabeled text from the desired domain. At first approximation it may seem to be a trival problem, but the number of non-standard words is enough even in what is considered clean text such as press wire news articales to make their synthesis sound bad without it.

Full NSW model description and justification to be added, doan play the following (older) parts.

Token to word rules

The basic model in Festival is that each *token* will be mapped a list of *words* by a call to a `token_to_word` function. This function will be called on each token and it should return a list of words. It may check the tokens to context (within the current utterance) too if necessary. The default action should (for most languages) simply be returning the token itself as a list of own word (itself). For example your basic function should look something like.

```
(define (MYLANG_token_to_words token name)
  "(MYLANG_token_to_words TOKEN NAME)
Returns a list of words for the NAME from TOKEN.  This primarily
allows the treatment of numbers, money etc."
  (cond
   (t
    (list name))))
```

This function should be set in your voice selection function as the function for token analysis

```
(set! token_to_words MYLANG_token_to_words)
```

This function should be added to to deal with all tokens that are not in your lexicon, cannot be treated by your letter-to-sound rules, or are ambiguous in some way and require context to resolve.

For example suppose we wish to simply treat all tokens consisting of strings of digits to be pronounced as a string of digits (rather than numbers). We would add something like the following

```
(set! MYLANG_digit_names
  '((0 "zero")
    (1 "one")
    (2 "two")
    (3 "three")
    (4 "four")
    (5 "five")
    (6 "six")
    (7 "seven")
    (8 "eight")
    (9 "nine")))

(define (MYLANG_token_to_words token name)
  "(MYLANG_token_to_words TOKEN NAME)
Returns a list of words for the NAME from TOKEN.  This primarily
allows the treatment of numbers, money etc."
  (cond
   ((string-matches name "[0-9]+") ;; any string of digits
    (mapcar
     (lambda (d)
      (car (cdr (assoc_string d MTLANG_digit_names))))
     (symbolexplode name)))
   (t
    (list name))))
```

But more elaborate rules are also necessary. Some tokens require context to disambiguate and sometimes multiple tokens are really one object e.g "*$12 billion*" must be rendered as "*twelve billion dollars*", where the money name crosses over the second word. Such multi-token rules must be split into multiple conditions, one for each part of the combined token. Thus we need to identify the "*$DIGITS*" is in a context followed by "*?illion*". The code below renders the full phrase for the dollar amount. The

second condition ensures nothing is returned for the "*?illion*" word as it has already been dealt with by the previous token.

```
((and (string-matches name "\\$[123456789]+")
      (string-matches (item.feat token "n.name") ".*illion.?"))
 (append
  (digits_to_cardinal (string-after name "$")) ;; amount
  (list (item.feat token "n.name"))           ;; magnitude
  (list "dollars")))                          ;; currency name
((and (string-matches name ".*illion.?")
      (string-matches (item.feat token "p.name") "\\$[123456789]+"))
 ;; dealt with in previous token
 nil)
```

Note this still is not enough as there may be other types of currency pounds, yen, francs etc, some of which may be mass nouns and require no plural (e.g. "*yen}*" and some of which make be count nouns require plurals. Also this only deals with whole numbers of .*illions, "*$1.25 million*" is common too. See the full example (for English) in `festival/lib/token.scm`.

A large list of rules are typically required. They should be looked upon as breaking down the problem into smaller parts, potentially recursive. For example hyphenated tokens can be split into two words. It is probably wise to explicitly deal with all tokens than are not purely alphabetic. Maybe having a catch-all that spells out all tokens that are not explicitly dealt with (e.g. the numbers). For example you could add the following as the penumtilmate condition in your `token_to_words` function

```
((not (string-matches name "[A-Za-z]"))
 (symbolexplode name))
```

Note this isn't necessary correct when certain letters may be homograpths. For example the token "*a*" may be a determiner or a letter of the alhpabet. When its a derterminer it may (often) be reduced) while as a letter it probably ins't (i.e pronunciation in "@" or "*ei}*". Other languages also example this problem (e.g. Spanish "*y*". Therefore when we call symbol explode we don't want just the the letter but to also specify that it is the letter pronunciation we want and not the any other form. To ensure the lexicon system gets the right pronunciation we there wish to specify the part fo speech with the letter. Actually rather than just a string of atomic words being returned by the `token_to_words` function the words may be descriptions including features. Thus for example we dont just want to return

```
(a b c)
```

We want to be more specific and return

```
(((name a) (pos nn))
 ((name b) (pos nn))
 ((name c) (pos nn)))
```

This can be done by the code

```
((not (string-matches name "[A-Za-z]"))
 (mapcar
  (lambda (l)
   ((list 'name l) (list 'pos 'nn)))
  (symbolexplode name)))
```

The above assumes that all single characters symbols (letters, digits, punctuation and other "funny" characters have an entry in your lexicon with a part of speech field `nn`, with a pronunctiation of the character in isolation.

The list of tokens that you may wish to write/train rules for is of couse language dependent and to a certain extent domain dependent. For example there are many more numbers in email text that in narative novels. The number of abbreviations is also much higher in email and news stories than in more normal text. It may be worth having a look at some typical data to find out the distribution and find out what is worth working on. For a rough guide the folowing is a list if the symbol types we currentl deal with in English, many of which will require some treatment in other languages.

Money

Money amounts often have different treatment than simple numbers and conventions about the sub-currency part (i.e. cents, pfennings etc). Remember that you its not just numbers in the local currency you have to deal with currency values from different countries are common in lots of different texts (e.g dollars, yen, DMs and euro).

Numbers

strings of digits will of course need mapping even if there is only one mapping for a language (rare). Consider at least telphone numbers verses amounts, most languages make a distinction here. In English we need to distinguish further, see below for the more detailed discussion.

number/number

This can be used as a date, fraction, alternate, context will help, though techniques of dropping back to saying the the string of characters often preserve the ambiguity which can be better that forcing a decision.

acronyms

List of upper case letters (with or without vowels). The decision to pronounce as a word or as letters is difficult in general but good guesses go far. If its short (< 4 chatacters) not in your lexicon not surround by other words in upper case, its probably an acronym, further analyss of vowels, consonant clusters etc will help.

number-number

Could be a range, of score (football), dates etc.

word-word

Usually a simple split on each part is sufficient---but not as when used as a dash.

word/word

As an alternative, or a Unix pathname

's or TOKENs

An appended "*s*" to a non alphabetic token is probabaly some form of pluralization, removing it and recursing on the analysis is a reasonable thing to try.

times and dates

These exist is variaous stnadardized forms many of which are easy to recognize and break down.

telephone numbers

This various from country to country (and by various conventions) but there may be standard forms that can be recognized.

romain numerals

Sometimes these are pronounced as numbers "*chapter II*", or as cardinals "*James II*".

ascii art

If you are dealing with on line text there are often extra characters in a document that should be ignored, or at least not pronounced literally, e.g. lines of hyphens used as separators.

email addresses, URLs, file names

Depending on your context this may be worth spending time on.

tokens containing any other non-alphanumeric character

Spliting the token around the non-alphanumeric and recursing on each part before and after it may be reasonable.

Remember the first purpose of text analysis is ensure you can deal with *anything*, even if it is just saying the word "*unknown*" (in the appropriate language). Also its probabaly not worth spending time on rare token forms, though remember it not easy to judge what are rare and what are not.

Number pronunciation

Almost every one will expect a synthesizer to be able to speech numbers. As it is not feasible to list all possible digit strings in your lexicon. You will need to provide a function that returns a string of words for a given string of digits.

In its simplest form you should provide a function that decodes the string of digits. The example `spanish_number` (and `spanish_number_from_digits}` in the released Spanish voice (`festvox_ellpc11k.tar.gz` is a good general example.

Multi-token numbers

A number of languages uses spaces within numbers where English might use commas. For example German, Polish and others text may contain

64 000

to denote sixty four thousand. As this will be multiple tokens in Festival's basic analysis it is necessary to write multiple conditions in your `token_to_words` function.

Declensions

In many languages, the pronunciation of a number depends on the thing that is being counted. For example the digit '1' in Spanish has multiple pronunciations depending on whether it is refering to a masculine or feminine object. In some languages this becomes much more complex where there are a number of possible declensions. In our Polish synthesizer we solved this by adding an extra argument to number generation function which then selected the actual number word (typically the final word in a number) based in the desired declension.

%%%%%%%%%%%%%%%%%%%%
Example to be added
%%%%%%%%%%%%%%%%%%%%

Homograph disambiguation

%%%%%%%%%%%%%%%%%%%%
Discussion to be added
%%%%%%%%%%%%%%%%%%%%

TTS modes

%%%%%%%%%%%%%%%%%%%%
Discussion to be added
%%%%%%%%%%%%%%%%%%%%

Mark-up modes

In some situtation it ispossible for the user of a text-to-speech system to provide more information for the synthesizer that just the text, or the type of text. It is near impossible for TTS engines to get everything right all of the time, so in such situation it is useful to offer the developer a method to help guide the synthesizer in its syntehsis process.

Most speech synthesizer offer some speech method or embedded commands but these are specific to one interface or one API. For example the Microsoft SAPI interface allows various commands to be embedded in a text string

* *some examples*

.

However there has been a move more recently to offer a general mark up method that is more general. A number of people saw the potential use of XML as a general method for marking up text for speech synthesis. The earliest method we know was in a Masters thesis at Edinburgh in 1995 [isard]. This was later published under the name SSML. A number of other groups were alos looking at this and a large consortium formed to define this further under various names STML, and eventually Sable.

Around the same time, more serious definitions of such a mark-up were being developed. The first to reach a well-define stage was JSML, (Java Speech Mark-up Language), which covered aspects of speech recognition and grammars as well as speech synthesis mark-up. Unlike any of the other XML based markup languages, JSML, as it was embedded within Java, could define exceptions in a reasonable way. One of the problems iwth a simpel XML markup is that it is one way. You can request a voice or a language or some functionality, but there is no mechanism for feedback to know if such a feature is actually available.

XML markup for speech have been further advances with VoiceXML, which defines a mark-up language for basic dialog systems. The speech synthesi part of the VoiceXML is closely follows the functionality of JSML and its predecessors.

A new standard for markup for speech synthesis is currently being defined by W3C under the name SSML, confusingly the same name as the earliest example, but not designed to be compatible with the original, but take into account the functionaly

and desires of users of TTS. SSML markup is also defined as the method for speech synthesis markup in Microsoft's SALT tags.

%%%%%%%%%%%%%%%%%%%%
Discussion to be added
%%%%%%%%%%%%%%%%%%%%

Chapter 7. Lexicons

This chapter covers method for finding the pronunciation of a word. This is either by a lexicon (a large list of words and their pronunciations) or by some method of letter to sound rules.

Word pronunciations

A pronunciation in Festival requires not just a list of phones but also a syllabic structure. In some languages the syllabic structure is very simple and well defined and can be unambiguously derived from a phone string. In English however this may not always be the case (compound nouns being the difficult case).

The lexicon structure that is basically available in Festival takes both a word and a part of speech (and arbitrary token) to find the given pronunciation. For English this is probably the optimal form, although there exist homographs in the language, the word itself and a fairly broad part of speech tag will mostly identify the proper pronunciation.

An example entry is

```
("photography"
 n
 (((f @ ) 0) ((t o g) 1) ((r @ f) 0) ((ii) 0)))
```

Not that in addition to explicit marking of syllables a stress value is also given (0 or 1). In some languages lexical is fully predictable, in others highly irregular. In some this field may be more appropriately used for an other purpose, e.g. tone type in Chinese.

There may be other languages which require a more complex (less complex) format and the decision to use some other format rather than this one is up to you.

Currently there is only residual support for morphological analysis in Festival. A finite state transducer based analyzer for English based on the work in [ritchie92] is included in `festival/lib/engmorph.scm` and `festival/lib/engmorphsyn.scm`. But this should be considered experimental at best. Give the lack of such an analyzer our lexicons need to list not only based forms of words but also all their morphological variants. This is (more or less) acceptable in languages such as English or French but which languages with richer morphology such as German it may seem an unnecessary requirement. Agglutenative languages such as Finnish and Turkish this appears to be even more a restriction. This is probably true but this current restriction not necessary hopeless. We have successfully build very good letter-to-sound rules for German, a language with a rich morphology which allows the system to properly predict pronunciations of morphological variants of root words it has not seen before. We have not yet done any experiments with Finnish or Turkish but see this technique would work, (though of course developing a properly morphological analyzer would be better).

Lexicons and addenda

The basic assumption in Festival is that you will have a large lexicon, tens of thousands of entries, that is a used as a standard part of an implementation of a voice. Letter-to-sound rules are used as back up when a word is not explicitly listed. This view is based on how English is best dealt with. However this is a very flexible view, An explicit lexicon isn't necessary in Festival and it may be possible to do much of the work in letter-to-sound rules. This is how we have implemented Spanish. However even when there is strong relationship between the letters in a word and their pronunciation we still find the a lexicon useful. For Spanish we still use the lexicon for symbols such as "$", "%", individual letters, as well as irregular pronunciations.

In addition to a large lexicon Festival also supports a smaller list called an *addenda* this is primarily provided to allow specific applications and users to add entries that aren't in the existing lexicon.

Out of vocabulary words

Because its impossible to list all words in a natural language for general text-to-speech you will need to provide something to pronounce out of vocabulary words. In some languages this is easy but in other's it is very hard. No matter what you do you *must* provide something even if it is simply replacing the unknown word with the word "*unknown*" (or its local language equivalent). By default a lexicon in Festival will throw an error if a requested word isn't found. To change this you can set the `lts_method`. Most usefully you can reset this to the name of function, which takes a word and a part of speech specification and returns a word pronunciation as described above.

For example is we are always going to return the word `unknown` but print a warning the the word is being ignored a suitable function is

```
(define (mylex_lts_function word feats)
"Deal with out of vocabulary word."
  (format t "unknown word: %s\n" word)
  '("unknown" n (((uh n) 1) ((n ou n) 1))))
```

Note the pronunciation of "*unknown*" must be in the appropriate phone set. Also the syllabic structure is required. You need to specify this function for your lexicon as follows

```
(lex.set.lts.method 'mylex_lts_function)
```

At one level above merely identifying out of vocabulary words, they can be spelled, this of course isn't ideal but it will allow the basic information to be passed over to the listener. This can be done with the out of vocabulary function, as follows.

```
(define (mylex_lts_function word feats)
"Deal with out of vocabulary words by spelling out the letters in the
word."
 (if (equal? 1 (length word))
    (begin
     (format t "the character %s is missing from the lexicon\" word)
     '("unknown" n (((uh n) 1) ((n ou n) 1))))
    (cons
     word
     'n
     (apply
      append
      (mapcar
       (lambda (letter)
        (car (cdr (cdr (lex.lookup letter 'n)))))
       (symbolexplode word))))))
```

A few point are worth noting in this function. This recursively calls the lexical lookup function on the characters in a word. Each letter should appear in the lexicon with its pronunciation (in isolation). But a check is made to ensure we don't recurse for ever. The `symbolexplode` function assumes that that letters are single bytes, which may not be true for some languages and that function would need to be replaced for that language. Note that we append the syllables of each of the letters in the word. For long words this might be too naive as there could be internal prosodic structure in such a spelling that this method would not allow for. In that case you

would want letters to be words thus the symbol explosion to happen at the token to word level. Also the above function assumes that the part of speech for letters is `n`. This is only really important where letters are homographs in languages so this can be used to distinguish which pronunciation you require (cf. "*a*" in English or "*y*" in French).

Building letter-to-sound rules by hand

For many languages there is a systematic relationship between the written form of a word and its pronunciation. For some languages this can be fairly easy to write down, by hand. In Festival there is a letter to sound rule system that allows rules to be written, but we also provided a method for building rule sets automatically which will often be more useful. The choice of using hand-written or automatically trained rules depends on the language you are dealing with and the relationship it has between its orthography and its phone set.

For well defined languages like Spanish and Croatian writting rules by hand can be more simple than training. Training requires an existing set of lexical entries to train from and that may be your decision criteria. Hand written letter to sound rules are context dependent re-write rules which are applied in sequence mapping strings letters to string of phones (though the system does not explicitly care what the types of the strings actually will be used for.

The basic form of the rules is

```
( LC [ alpha ] RC => beta )
```

Which is interpreter as `alpha`, a string of one or more symbols on the input tape is written to `beta`, a string of zero or more symbols on the output tape, when in the presence of `LC`, a left context of zero or more input symbols, and `RC` a right context on zero or more input symbols. Note the input tape and the output tape are different, allthough the input and output alphabets need not be distinct the left hand side of a rule only can refer to the input tape and never to anything that has been produce by a right hand side. Thus rules within a ruleset cannot "feed" or "bleed" themselves. It is possible to cascade multiple rule sets, but we will discuss that below.

For example to desl with the pronunciation of the letters "ch" word initially in English we may right two rules like this

```
( # [ c h ] r => k )
( # [ c h ] => ch )
```

To deal with words like "christmas", and "chair". Note the # symbol is special and used to denote a word boundary. LTS rule may refer to word boundary but cannot refer to prevous or following words, you would need to do this with some form of post-lexical rule (See the Section called *Post-lexical rules*) where the word is within some context. In the above rules we are mapping *two* letters *c* and *h* to a single phone *k* or *ch*. Also note the order of these rules. The first rule is more specific than the second. This is should appear first to deal with the specific case. In the order were reversed `k` could never apply as the `ch` would cover that case too.

Thus LTS rules should be written with the most specific cases first and typically end in a default case. Their should be a default case for all individual letters in the language's alphabet without and context restrictions mapping to some default phone. Therefore following the above rules there would be other `c` rules with various contexts but the final one should probably be

```
( [ c ] => k )
```

As it is a common error in writting these rules, it is worth repeating. If a rule set is to be universally applicable *all* letters in the input alphabet must have at a rule mapping them to some phone.

The section to be mapped (within square brackets) and the section it is mapped into (after the "=>") must be items in the input and output alphabets and may not include sets or regular expression operators. This does mean more rules need to be explicitly written than you might like, but that will also help you not forget some rules that are required.

For some languages it is conveninet to write a number of rules sets. For example, one to map the input in lower case, and maybe deal with alternate treatments of accent characters e.g. re-write the ASCII "e'" as

* *AWB: e acute*

. Also we have used rule tests to post process the generated phone string to add stress and syllable breaks.

Finally some people have stressed that writing good letter to sound rules is hard. We would disagree with this, from our experience writing good letter to sound rules by hand is *very* hard and *very* skilled and *very* laborious. For anything but the simplest of languages writting rules by hand requires much more time that people typically have, and will still contain errors (even with an exception list). However hand rules sets may be ideal in some circumstances.

Building letter-to-sound rules automatically

For some languages the writing of a rule system is too difficult. Although there have been many valiant attempts to do so for languages like English life is basically too short to do this. Therefore we also include a method for automatically building LTS rules sets for a lexicon of pronunciations. This technique has successfully been used from English (British and American), French and German. The difficulty and appropriateness of using letter-to-sound rules is very language dependent,

The following outlines the processes involved in building a letter to sound model for a language given a large lexicon of pronunciations. This technique is likely to work for most European languages (including Russian) but doesn't seem particularly suitable for very language alphabet languages like Japanese and Chinese. The process described here is not (yet) fully automatic but the hand intervention required is small and may easily be done even by people with only a very little knowledge of the language being dealt with.

The process involves the following steps

- Pre-processing lexicon into suitable training set
- Defining the set of allowable pairing of letters to phones. (We intend to do this fully automatically in future versions).
- Constructing the probabilities of each letter/phone pair.
- Aligning letters to an equal set of phones/_epsilons_.
- Extracting the data by letter suitable for training.
- Building CART models for predicting phone from letters (and context).
- Building additional lexical stress assignment model (if necessary).

All except the first two stages of this are fully automatic.

Before building a model its wise to think a little about what you want it to do. Ideally the model is an auxiliary to the lexicon so only words not found in the lexicon will require use of the letter-to-sound rules. Thus only unusual forms are likely to require the rules. More precisely the most common words, often having the most

non-standard pronunciations, should probably be explicitly listed always. It is possible to reduce the size of the lexicon (sometimes drastically) by removing all entries that the training LTS model correctly predicts.

Before starting it is wise to consider removing some entries from the lexicon before training, I typically will remove words under 4 letters and if part of speech information is available I remove all function words, ideally only training from nouns verbs and adjectives as these are the most likely forms to be unknown in text. It is useful to have morphologically inflected and derived forms in the training set as it is often such variant forms that not found in the lexicon even though their root morpheme is. Note that in many forms of text, proper names are the most common form of unknown word and even the technique presented here may not adequately cater for that form of unknown words (especially if they unknown words are non-native names). This is all stating that this may or may not be appropriate for your task but the rules generated by this learning process have in the examples we've done been much better than what we could produce by hand writing rules of the form described in the previous section.

First preprocess the lexicon into a file of lexical entries to be used for training, removing functions words and changing the head words to all lower case (may be language dependent). The entries should be of the form used for input for Festival's lexicon compilation. Specifically the pronunciations should be simple lists of phones (no syllabification). Depending on the language, you may wish to remove the stressing---for examples here we have though later tests suggest that we should keep it in even for English. Thus the training set should look something like

```
("table" nil (t ei b l))
("suspicious" nil (s @ s p i sh @ s))
```

It is best to split the data into a training set and a test set if you wish to know how well your training has worked. In our tests we remove every tenth entry and put it in a test set. Note this will mean our test results are probably better than if we removed say the last ten in every hundred.

The second stage is to define the set of allowable letter to phone mappings irrespective of context. This can sometimes be initially done by hand then checked against the training set. Initially construct a file of the form

```
(require 'lts_build)
(set! allowables
      '((a _epsilon_)
        (b _epsilon_)
        (c _epsilon_)
        ...
        (y _epsilon_)
        (z _epsilon_)
        (# #)))
```

All letters that appear in the alphabet should (at least) map to `_epsilon_`, including any accented characters that appear in that language. Note the last two hashes. These are used by to denote beginning and end of word and are automatically added during training, they must appear in the list and should only map to themselves.

To incrementally add to this allowable list run festival as

```
festival allowables.scm
```

and at the prompt type

```
festival> (cummulate-pairs "oald.train")
```

with your train file. This will print out each lexical entry that couldn't be aligned with the current set of allowables. At the start this will be every entry. Looking at these entries add to the allowables to make alignment work. For example if the following word fails

```
("abate" nil (ah b ey t))
```

Add `ah` to the allowables for letter `a`, `b` to `b`, `ey` to `a` and `t` to letter `t`. After doing that restart festival and call `cummulate-pairs` again. Incrementally add to the allowable pairs until the number of failures becomes acceptable. Often there are entries for which there is no real relationship between the letters and the pronunciation such as in abbreviations and foreign words (e.g. "aaa" as "t r ih p ax l ey"). For the lexicons I've used the technique on less than 10 per thousand fail in this way.

It is worth while being consistent on defining your set of allowables. (At least) two mappings are possible for the letter sequence `ch}---havin` letter `c` go to phone `ch` and letter `h` go to `_epsilon_` and also letter `c` go to phone `_epsilon_` and letter `h` goes to `ch`. However only one should be allowed, we preferred `c` to `ch`.

It may also be the case that some letters give rise to more than one phone. For example the letter `x` in English is often pronounced as the phone combination `k` and `s`. To allow this, use the multiphone `k-s`. Thus the multiphone `k-s` will be predicted for `x` in some context and the model will separate it into two phones while it also ignoring any predicted `_epsilons_`. Note that multiphone units are relatively rare but do occur. In English, letter `x` give rise to a few, `k-s` in `taxi`, `g-s` in `example`, and sometimes `g-zh` and `k-sh` in `luxury`. Others are `w-ah` in `one`, `t-s` in `pizza`, `y-uw` in `new` (British), `ah-m` in `-ism` etc. Three phone multiphone are much rarer but may exist, they are not supported by this code as is, but such entries should probably be ignored. Note the `-` sign in the multiphone examples is significant and is used to identify multiphones.

The allowables for OALD end up being

```
(set! allowables
'
((a _epsilon_ ei aa a e@ @ oo au o i ou ai uh e)
 (b _epsilon_ b )
 (c _epsilon_ k s ch sh @-k s t-s)
 (d _epsilon_ d dh t jh)
 (e _epsilon_ @ ii e e@ i @@ i@ uu y-uu ou ei aa oi y y-u@ o)
 (f _epsilon_ f v )
 (g _epsilon_ g jh zh th f ng k t)
 (h _epsilon_ h @ )
 (i _epsilon_ i@ i @ ii ai @@ y ai-@ aa a)
 (j _epsilon_ h zh jh i y )
 (k _epsilon_ k ch )
 (l _epsilon_ l @-l l-l)
 (m _epsilon_ m @-m n)
 (n _epsilon_ n ng n-y )
 (o _epsilon_ @ ou o oo uu u au oi i @@ e uh w u@ w-uh y-@)
 (p _epsilon_ f p v )
 (q _epsilon_ k )
 (r _epsilon_ r @@ @-r)
 (s _epsilon_ z s sh zh )
 (t _epsilon_ t th sh dh ch d )
 (u _epsilon_ uu @ w @@ u uh y-uu u@ y-u@ y-u i y-uh y-@ e)
 (v _epsilon_ v f )
 (w _epsilon_ w uu v f u)
 (x _epsilon_ k-s g-z sh z k-sh z g-zh )
 (y _epsilon_ i ii i@ ai uh y @ ai-@)
 (z _epsilon_ z t-s s zh )
 (# #)
 ))
```

Note this is an exhaustive list and (deliberately) says nothing about the contexts or frequency that these letter to phone pairs appear. That information will be generated automatically from the training set.

Once the number of failed matches is significantly low enough let `cummulate-pairs` run to completion. This counts the number of times each letter/phone pair occurs in allowable alignments.

Next call

```
festival> (save-table "oald-")
```

with the name of your lexicon. This changes the cumulation table into probabilities and saves it.

Restart festival loading this new table

```
festival allowables.scm oald-pl-table.scm
```

Now each word can be aligned to an equally-lengthed string of phones, epsilon and multiphones.

```
festival> (aligndata "oald.train" "oald.train.align")
```

Do this also for you test set.

This will produce entries like

```
aaronson _epsilon_ aa r ah n s ah n
abandon ah b ae n d ah n
abate ah b ey t _epsilon_
abbe ae b _epsilon_ iy
```

The next stage is to build features suitable for `wagon` to build models. This is done by

```
festival> (build-feat-file "oald.train.align" "oald.train.feats")
```

Again the same for the test set.

Now you need to construct a description file for `wagon` for the given data. The can be done using the script `make_wgn_desc` provided with the speech tools

Here is an example script for building the models, you will need to modify it for your particular database but it shows the basic processes

```
for i in a b c d e f g h i j k l m n o p q r s t u v w x y z
do
  # Stop value for wagon
  STOP=2
  echo letter $i STOP $STOP
  # Find training set for letter $i
  cat oald.train.feats |
   awk '{if ($6 == "'$i'") print $0}' >ltsdataTRAIN.$i.feats
  # split training set to get heldout data for stepwise testing
  traintest ltsdataTRAIN.$i.feats
  # Extract test data for letter $i
  cat oald.test.feats |
   awk '{if ($6 == "'$i'") print $0}' >ltsdataTEST.$i.feats
  # run wagon to predict model
  wagon -data ltsdataTRAIN.$i.feats.train -test ltsdataTRAIN.$i.feats.test \
        -stepwise -desc ltsOALD.desc -stop $STOP -output lts.$i.tree
  # Test the resulting tree against
```

```
    wagon_test -heap 2000000 -data ltsdataTEST.$i.feats -desc ltsOALD.desc \
            -tree lts.$i.tree
done
```

The script `traintest` splits the given file `X` into `X.train` and `X.test` with every tenth line in `X.test` and the rest in `X.train`.

This script can take a significant amount of time to run, about 6 hours on a Sun Ultra 140.

Once the models are created the must be collected together into a single list structure. The trees generated by `wagon` contain fully probability distributions at each leaf, at this time this information can be removed as only the most probable will actually be predicted. This substantially reduces the size of the tress.

```
(merge_models 'oald_lts_rules "oald_lts_rules.scm" allowables)
```

(`merge_models` is defined within `lts_build.scm` The given file will contain a `set!` for the given variable name to an assoc list of letter to trained tree. Note the above function naively assumes that the letters in the alphabet are the 26 lower case letters of the English alphabet, you will need to edit this adding accented letters if required. Note that adding "'" (single quote) as a letter is a little tricky in scheme but can be done---the command `(intern "'")` will give you the symbol for single quote.

To test a set of lts models load the saved model and call the following function with the test align file

```
festival oald-table.scm oald_lts_rules.scm
festival> (lts_testset "oald.test.align" oald_lts_rules)
```

The result (after showing all the failed ones), will be a table showing the results for each letter, for all letters and for complete words. The failed entries may give some notion of how good or bad the result is, sometimes it will be simple vowel differences, long versus short, schwa versus full vowel, other times it may be who consonants missing. Remember the ultimate quality of the letter sound rules is how adequate they are at providing *acceptable* pronunciations rather than how good the numeric score is.

For some languages (e.g. English) it is necessary to also find a stress pattern for unknown words. Ultimately for this to work well you need to know the morphological decomposition of the word. At present we provide a CART trained system to predict stress patterns for English. If does get 94.6% correct for an unseen test set but that isn't really very good. Later tests suggest that predicting stressed and unstressed phones directly is actually better for getting whole words correct even though the models do slightly worse on a per phone basis [black98b].

As the lexicon may be a large part of the system we have also experimented with removing entries from the lexicon if the letter to sound rules system (and stress assignment system) can correct predict them. For OALD this allows us to half the size of the lexicon, it could possibly allow more if a certain amount of fuzzy acceptance was allowed (e.g. with schwa). For other languages the gain here can be very significant, for German and French we can reduce the lexicon by over 90%. The function `reduce_lexicon` in `festival/lib/lts_build.scm` was used to do this. A discussion of using the above technique as a dictionary compression method is discussed in [pagel98]. A morphological decomposition algorithm, like that described in [black91], may even help more.

The technique described in this section and its relative merits with respect to a number of languages/lexicons and tasks is discussed more fully in [black98b].

Post-lexical rules

In fluent speech word boundaries are often degraded in a way that causes co-articulation across boundaries. A lexical entry should normally provide pronunciations as if the word is being spoken in isolation. It is only once the word has been inserted into the the context in which it is going to spoken can co-articulary effects be applied.

Post lexical rules are a general set of rules which can modify the segment relation (or any other part of the utterance for that matter), after the basic pronunciations have been found. In Festival post-lexical rules are defined as functions which will be applied to the utterance after intonational accents have been assigned.

For example in British English word final /r/ is only produced when the following word starts with a vowel. Thus all other word final /r/s need to be deleted. A Scheme function that implements this is as follows

```
(define (plr_rp_final_r utt)
  (mapcar
   (lambda (s)
    (if (and (string-equal "r" (item.name s))  ;; this is an r
             ;; it is syllable final
             (string-equal "1" (item.feat s "syl_final"))
             ;; the syllable is word final
             (not (string-equal "0"
                  (item.feat s "R:SylStructure.parent.syl_break")))
             ;; The next segment is not a vowel
             (string-equal "-" (item.feat s "n.ph_vc")))
        (item.delete s)))
   (utt.relation.items utt 'Segment)))
```

In English we also use post-lexical rules for phenomena such as vowel reduction and schwa deletion in the possessive "*'s*".

Building lexicons for new languages

Traditionally building a new lexicon for a language was a significant piece of work taking several expert phonologists perhaps several years to construct a lexicon with reasonable coverage. However we include a method here that can cut this time significantly using the basic technology provided with this documentation.

The basic idea is add the most common words to a lexicon, expclitly giving their pronunciation by hand, then automatically build letter to sound rules from the initial data. Then finding the most common words submit them to the system and check their correctness. If wrong they are corrected and added to the lexicon, if correct they are added to the lexicon as is. Over multiple passes the lexicon and letter to sound rules will improve. As each pass the letter to sound rules are re-generate with the new data making them more correct.

This tecynique has been proved succesful for a number of language cutting the amount to time and effort to perhaps checking thousands of words rather than tens of thousands of words. It also is a structured method that requires only knowledge of the basic language to carry out. Good lexicons can be generated in as little as a coupld of weeks, though to get greater than 95% correctness of words in a language could still take several months work.

As stated above you can never list all the words in a language, but having grateter than 95% coverage with letter to sound rule accuracy grater than 75% you will have a lexicon that is competitive with those that take many year build. In fact because you can build a lexicon in a shorter time it more likely to be consistent and there better for synthesis.

Chapter 8. Building prosodic models

Phrasing

Prosodic phrasing in speech synthesis makes the whole speech more understandable. Due to the size of peoples lungs there is a finite length of time people can talk before they can take a breath, which defines an upper bound on prosodic phrases. However we rarely make our phrases this maximum length and use phrasing to mark groups within the speech. There is the apocryphal story of the speech synthesis example with an unnaturally long prosodic phrase played at a conference presentation. At the end of the phrase the audience all took a large in-take of breathe.

For the most case very simple prosodic phrasing is sufficient. A comparison of various prosodic phrasing techniques is discussed in [taylor98a], though we will cover some of them here also.

For English (and most likely many other language too) simple rules based on punctuation is a very good predictor of prosodic phrase boundaries. It is rare that punctuation exists where there is no boundary, but there will be a substantial number of prosodic boundaries which are not explicitly marked with punctuation. Thus a prosodic phrasing algorithm solely based on punctuation will typically under predict but rarely make a false insertion. However depending on the actual application you wish to use the synthesizer for it may be the case that explicitly adding punctuation at desired phrase breaks is possible and a prediction system based solely on punctuation is adequate.

Festival basically supports two methods for predicting prosodic phrases, though any other method can easily be used. Note that these do not necessary entail pauses in the synthesized output. Pauses are further predicted from prosodic phrase information.

The first basic method is by CART tree. A test is made on each word to predict it is at the end of a prosodic phrase. The basic CART tree returns `B` or `BB` (though may return what you consider is appropriate form break labels as long as the rest of your models support it). The two levels identify different levels of break, `BB` being a used to denote a bigger break (and end of utterance).

The following tree is very simple and simply adds a break after the last word of a token that has following punctuation. Note the first condition is done by a lisp function as we wand to ensure that only the last word in a token gets the break. (Earlier erroneous versions of this would insert breaks after each word in "*1984.*"

```
(set! simple_phrase_cart_tree
'
((lisp_token_end_punc in ("?" "." ":"))
 ((BB))
 ((lisp_token_end_punc in ("'" "\"" "," ";"))
  ((B))
  ((n.name is 0)  ;; end of utterance
   ((BB))
   ((NB))))))
```

This tree is defined `festival/lib/phrase.scm` in the standard distribution and is certainly a good first step in defining a phrasing model for a new language.

To make a better phrasing model requires more information. As the basic punctuation model underpredicts we need information that will find reasonable boundaries within strings of words. In English, boundaries are more likely between content words and function words, because most function words are before the words they related to, in Japanese function words are typically after their relate content words so breaks are more likely between function words and content words. If you have no data to train from, written rules, in a CART tree, can exploited this fact and give a phrasing model that is better than a punctuation only. Basically a rule could be if

the current word is a content word and the next is a function word (or the reverse if that appropriate for a language) and we are more than 5 words from a punctuation symbol them predict a break. We maybe also want to insure that we are also at least five words from predicted break too.

Note the above basic rules aren't optimal but when you are building a new voice in a new language and have no data to train from you will get reasonably far with simple rules like that, such that phrasing prediction will be less of a problem than the other problems you will find in you voice.

To implement such a scheme we need three basic functions: one to determine if the current word is a function of content word, one to determine number of words since previous punctuation (or start of utterance) and one to determine number of words to next punctuation (or end of utterance. The first of these functions is already provided for with a feature, through the feature function `gpos`. This uses the word list in the lisp variable `guess_pos` to determine the basic category of a word. Because in most languages the set of function words is very nearly a closed class they can usually be explicitly listed. The format of the `guess_pos` variable is a list of lists whose first element is the set name and the rest of the list if the words that are part of that set. Any word not a member of any of these sets is defined to be in the set `content`. For example the basic definition for this for English, given in `festival/lib/pos.scm` is

```
(set! english_guess_pos
      '((in of for in on that with by at from as if that against about
     before because if under after over into while without
     through new between among until per up down)
 (to to)
 (det the a an no some this that each another those every all any
      these both neither no many)
 (md will may would can could should must ought might)
 (cc and but or plus yet nor)
 (wp who what where how when)
 (pps her his their its our their its mine)
 (aux is am are was were has have had be)
 (punc "." "," ":" ";" "\"" "'" "(" "?" ")" "!")
 ))
```

The punctuation distance check can be written as a Lisp feature function

```
(define (since_punctuation word)
 "(since_punctuation word)
Number of words since last punctuation or beginning of utterance."
 (cond
   ((null word) 0) ;; beginning or utterance
   ((string-equal "0" (item.feat word "p.lisp_token_end_punc")) 0)
   (t
    (+ 1 (since_punctuation (item.prev word))))))
```

The function looking forward would be

```
(define (until_punctuation word)
 "(until_punctuation word)
Number of words until next punctuation or end of utterance."
 (cond
   ((null word) 0) ;; beginning or utterance
   ((string-equal "0" (token_end_punc word)) 0)
   (t
    (+ 1 (since_punctuation (item.prev word))))))
```

The whole tree using these features that will insert a break at punctuation or between content and function words more than 5 words from a punctuation symbol is as follows

```
(set! simple_phrase_cart_tree_2
'
((lisp_token_end_punc in ("?" "." ":"))
 ((BB))
 ((lisp_token_end_punc in ("'" "\"" "," ";"))
  ((B))
  ((n.name is 0)  ;; end of utterance
   ((BB))
   ((lisp_since_punctuation > 5)
    ((lisp_until_punctuation > 5)
     ((gpos is content)
      ((n.gpos content)
       ((NB))
       ((B)))   ;; not content so a function word
      ((NB)))   ;; this is a function word
     ((NB)))    ;; to close to punctuation
    ((NB)))     ;; to soon after punctuation
   ((NB))))))
```

To use this add the above to a file in your `festvox/` directory and ensure it is loaded by your standard voice file. In your voice definition function. Add the following

```
(set! guess_pos english_guess_pos) ;; or appropriate for your language

(Parameter.set 'Phrase_Method 'cart_tree)
(set! phrase_cart_tree simple_phrase_cart_tree_2)
```

A much better method for predicting phrase breaks is using a full statistical model trained from data. The problem is that you need a lot of data to train phrase break models. Elsewhere in this document we suggest the use of a timit style database or around 460 sentences, (around 14500 segments) for training models. However a database such as this as very few internal utterance phrase breaks. An almost perfect model word predict breaks at the end of each utterances and never internally. Even the f2b database from the Boston University Radio New Corpus [ostendorf95] which does have a number of utterance internal breaks isn't really big enough. For English we used the MARSEC database [roach93] which is much larger (around 37,000 words). Finding such a database for your language will not be easy and you may need to fall back on a purely hand written rule system.

Often syntax is suggested as a strong correlate of prosodic phrase. Although there is evidence that it influences prosodic phrasing, there are notable exceptions [bachenko90]. Also considering how difficult it is to get a reliable parse tree it is probably not worth the effort, training a reliable parser is non-trivial, (though we provide a method for training stochastic context free grammars in the speech tools, see manual for details). Of course if your text to be synthesized is coming from a language system such as machine translation or language generation then a syntax tree may be readily available. In that case a simple rule mechanism taking into account syntactic phrasing may be useful

When only moderate amounts of data are available for training a simple CART tree may be able to tease out a reasonable model. See [hirschberg94] for some discussion on this. Here is a short example of building a CART tree for phrase prediction. Let us assume you have a database of utterances as described previously. By convention we build models in directories under `festival/` in the main database directory. Thus let us create `festival/phrbrk`.

First we need to list the features that are likely to be suitable predictors for phrase breaks. Add these to a file `phrbrk.feats`, what goes in here will depend on what you have, full part of speech helps a lot but you may not have that for your language. The `gpos` described above is a good cheap alternative. Possible features may be

```
word_break
lisp_token_end_punc
lisp_until_punctuation
lisp_since_punctuation
p.gpos
gpos
n.gpos
```

Given this list you can extract features form your database of utterances with the Festival script `dumpfeats`

```
dumpfeats -eval ../../festvox/phrbrk.scm -feats phrbrk.feats \
  -relation Word -output phrbrk.data ../utts/*.utts
```

`festvox/phrbrk.scm` should contain the definitions of the function `until_punctuation`, `since_punctuation` and any other Lisp feature functions you define.

Next we want to split this data into test and train data. We provide a simple shell script called `traintest` which splits a given file 9:1, i.e every 10th line is put in the test set.

```
traintest phrbrk.data
```

As we intend to run `wagon` the CART tree builder on this data we also need create the feature description file for the data. The feature description file consists of a bracketed list of feature name and type. Type may be `int` `float` or categorical where a list of possible values is given. The script `make_wagon_desc` (distributed with the speech tools) will make a reasonable approximation for this file

```
make_wagon_desc phrbrk.data phrbrk.feats phrbrk.desc
```

This script will treat all features as categorical. Thus any `float` or `int` features will be treated categorically and each value found in the data will be listed as a separate item. In our example `lisp_since_punctuation` and `lisp_until_punctuation` are actually float (well maybe even int) but they will be listed as categorically in `phrbrk.desc`, something like

```
...
(lisp_since_punctuation
0
1
2
4
3
5
6
7
8)
...
```

You should change this entry (by hand) to be

```
...
(lisp_since_punctuation float )
...
```

The script cannot work out the type of a feature automatically so you must make this decision yourself.

Now that we have the data and description we can build a CART tree. The basic command for `wagon` will be

```
wagon -desc phrbrk.desc -data phrbrk.data.train -test phrbrk.data.test \
  -output phrbrk.tree
```

You will probably also want to set a *stop* value. The default stop value is 50, which means there must be at least 50 examples in a group before it will consider looking for a question to split it. Unless you have a *lot* of data this is probably too large and a value of 10 to 20 is probably more reasonable.

Other arguments to `wagon` should also be considered. A stepwise approach where all features are tested incrementally to find the best set of features which give the best tree can give better results than simply using all features. Though care should be taken with this as the generated tree becomes optimized from the given test set. Thus a further held our test set is required to properly test the accuracy of the result. In the stepwise case it is normal to split the train set again and call wagon as follows

```
traintest phrbrk.data.train
wagon -desc phrbrk.desc -data phrbrk.data.train.train \
  -test phrbrk.data.train.test \
  -output phrbrk.tree -stepwise
wagon_test -data phrbrk.data.test -desc phrbrk.desc \
  -tree phrbrk.tree
```

Stepwise is particularly useful when features are highly correlated with themselves and its not clear which is best general predictor. Note that stepwise will take *much* longer to run as it potentially must build a large number of trees.

Other arguments to `wagon` can be considered, refer to the relevant chapter in speech tools manual for their details.

However it should be noted that without a good intonation and duration model spending time on producing good phrasing is probably not worth it. The quality of all these three prosodic components is closely related such that if one is much better than there may not be any real benefit.

Accent/Boundary Assignment

Accent and boundary tones are what we will use, hopefully in a theory independent way, to refer to the two main types of intonation event. For English, and for many other languages the prediction of position of the accents and boundaries can be done as an independent process from F0 contour generation itself. This is definite true from the major theories we will be considering.

As with phrase break prediction there are some simple rules that will go a surprisingly long way. And as with most of the other statistical learning techniques simple rules cover most of the work, more complex rules work better, but the best results are from using the sorts of information you were using in rules but statistically training them from a appropriate data.

For English the placement of accents on stressed syllables in all content words is a quite reasonable approximation achieving about 80% accuracy on typical databases. [hirschberg90] is probably the best example of a detailed rule driven approach (for English). CART trees based on the sorts of features Hirschberg uses are quite reasonable. Though eventual these rules become limiting and a richer knowledge source is required to assign accent patterns to complex nominals (see [sproat90}].

However all these techniques quickly come to the stumbling block that although simple so-called discourse neutral intonation is relatively easy achieve, achieving realistic, natural accent placement is still beyond our synthesis systems (though perhaps not for much longer).

The simplest rule for English may be reasonable for other languages. There are even simpler solutions to this, such as fixed prosody, or fixed declination, but apart from debugging a voice these are simpler than is required even for the most basic voices.

For English, adding a simple hat accent on lexically stressed syllables in all content words works surprisingly well. To do this in Festival you need a CART tree to predict accentedness, and rules to add the hat accent (though we will leave out F0 generation until the next section).

A basic tree that predicts accents of stressed syllables in content words is

```
(set! simple_accent_cart_tree
 '
 (
  (R:SylStructure.parent.gpos is content)
   ( (stress is 1)
     ((Accented))
     ((NONE))
   )
 )
)
```

The above tree simply distinguishes accented syllables from non-accented. In theories like ToBI [silverman92], a number of different types of accent are supported. ToBI, with variations, has been applied to a number of languages and may be suitable for yours. However, although accent and boundary types have been identified for various languages and dialects, a computational mechanism for generating and F0 contour from an accent specification often has not yet been specified (we will discuss this more fully below).

If the above is considered too naive a more elaborate hand specified tree can also be written, using relevant factors, probably similar to those used in [hirschberg90]. Following that, training from data is the next option. Assuming a database exists and has been labeled with discrete accent classifications, we can extract data from it for training a CART tree with `wagon`. We will build the tree in `festival/accents/`. First we need a file listing the features that are felt to affect accenting. For this we will predict accents on syllables as that has been used for the English voices created so far, but there is an argument for predict accent placement on a word basis as although accents will need to be syllable aligned, which syllable in a word gets the accent is reasonably well defined (at least compared with predicting accent placement).

A possible list of features for accent prediction is put in the file `accent.feats`.

```
R:Intonation.daughter1.name
R:SylStructure.parent.R:Word.p.gpos
R:SylStructure.parent.gpos
R:SylStructure.parent.R:Word.n.gpos
ssyl_in
syl_in
ssyl_out
syl_out
p.stress
stress
n.stress
pp.syl_break
p.syl_break
syl_break
n.syl_break
nn.syl_break
pos_in_word
position_type
```

We can extract these features from the utterances using the Festival script `dumpfeats`

```
dumpfeats -feats accent.feats -relation Syllable \
    -output accent.data ../utts/*.utts
```

We now need a description file for the features which can be approximated by the speech tools script `make_wagon_desc`

```
make_wagon_desc accent.data accent.feat accent.desc
```

Because this script cannot determine if a feature is categorical, if takes an range of values you must hand edit the output file and change any feature to `float` or `int` if that is what it is.

The next stage is to split the data into training and test sets. If stepwise training is to be used for building the CART tree (which is recommended) then the training data should be further split

```
traintest accent.data
traintest accent.data.train
```

Deciding on a stop value for training depends on the number of examples, though this can be tuned to ensure over-training isn't happening.

```
wagon -data accent.data.train.train -desc accent.desc \
  -test accent.data.train.test -stop 10 -stepwise -output accent.tree
wagon_test -data accent.data.test -desc  accent.desc \
  -tree accent.tree
```

This above is designed to predict accents, and similar tree should be used to predict boundary tones as well. For the most part intonation boundaries are defined to occur at prosodic phrase boundaries so that task is somewhat easier, though if you have a number of boundary tone types in your inventory then the prediction is not so straightforward.

When training ToBI type accent types it is not easy to get the right type of variation in the accent types. Although some ToBI labels have been associated with semantic intentions and including discourse information has been shown help prediction (e.g. [black97a}], getting this acceptably correct is not easy. Various techniques in modifying the training data do seem to help. Because of the low incidence of "*L**" labels in at least the f2b data, duplicating all sample points in the training data with L's does increase the likelihood of prediction and does seem to give a more varied distribution. Alternatively wagon returns a probability distribution for the accents, normally the most probable is selected, this could be modified to select from the distribution randomly based on their probabilities.

Once trees have been built they can be used in a voices as follows. Within the voice definition function

```
(set! int_accent_cart_tree simple_accent_cart_tree)
(set! int_tone_cart_tree simple_tone_cart_tree)
(Parameter.set 'Int_Method Intonation_Tree)
```

or if only one tree is required you can use the simpler intonation method

```
(set! int_accent_cart_tree simple_accent_cart_tree)
(Parameter.set 'Int_Method Intonation_Simple)
```

F0 Generation

Predicting where accents go (and their types) is only half of the problem. We also have build an F0 contour based on these. Note intonation is split between accent placement and F0 generation as it is obvious that accent position influences durations and an F0 contour cannot be generated without knowing the durations of the segments the contour is to be generated over.

There are three basic F0 generation modules available in Festival, though others could be added, by general rule, by linear regression/CART, and by Tilt.

F0 by rule

The first is designed to be the most general and will always allow some form of F0 generation. This method allows target points to be programmatically created for each syllable in an utterance. The idea follows closely a generalization of the implementation of ToBI type accents in [anderson84], where n-points are predicted for each accent. They (and others in intonation) appeal to the notion of baseline and place target F0 points above and below that line based on accent type, position in phrase. The baseline itself is often defined to decline over the phrase reflecting the general declination of F0 over type.

The simple idea behind this general method is that a Lisp function is called for each syllable in the utterance. That Lisp function returns a list of target F0 points that lie within that syllable. Thus the generality of this methods actual lies in the fact that it simply allows the user to program anything they want. For example our simple hat accent can be generated using this technique as follows.

This fixes the F0 range of the speaker so would need to be changed for different speakers.

```
(define (targ_func1 utt syl)
  "(targ_func1 UTT STREAMITEM)
Returns a list of targets for the given syllable."
  (let ((start (item.feat syl 'syllable_start))
        (end (item.feat syl 'syllable_end)))
    (if (equal? (item.feat syl "R:Intonation.daughter1.name") "Accented")
        (list
         (list start 110)
         (list (/ (+ start end) 2.0) 140)
         (list end 100)))))
```

It simply checks if the current syllable is accented and if so returns a list of position/target pairs. A value at the start of the syllable or 110Hz, a value at 140Hz at the mid-point of the syllable and a value of 100 at the end.

This general technique can be expanded with other rules as necessary. Festival includes an implementation of ToBI using exactly this technique, it is based on the rules described in [jilka96] and in the file `festival/lib/tobi_f0.scm`.

F0 by linear regression

This technique was developed specifically to avoid the difficult decisions of exactly what parameters with what value should be used in rules like those of [anderson84]. The first implementation of this work is presented [black96]. The idea is to find the appropriate F0 target value for each syllable based on available features by training from data. A set of features are collected for each syllable and a linear regression model is used to model three points on each syllable. The technique produces reasonable synthesis and requires less analysis of the intonation models that would be required to write a rule system using the general F0 target method described in the previous section.

However to be fair, this technique is also much simpler and there are are obviously a number of intonational phenomena which this cannot capture (e.g. multiple accents on syllables and it will never really capture accent placement with respect to the vowel). The previous technique allows specification of structure but without explicit training from data (though doesn't exclude that) while this technique imposes almost no structure but depends solely on data. The Tilt modeling discussed in the following section tries to balance these two extremes.

The advantage of the linear regression method is very little knowledge about the intonation the language under study needs to be known. Of course if there is knowledge and theories it is usually better to follow them (or at least find the features which influence the F0 in that language). Extracting features for F0 modeling is similar to extracting features for the other models. This time we want the means F0 at the start middle and end of each utterance. The Festival features `syl_startpitch`, `syl_midpitch` and `syl_endpitch` proved this. Note that `syl_midpitch` returns the pitch at the mid of the vowel in the syllable rather than the middle of the syllable.

For a linear regression model all features *must* be continuous. Thus features which are categorical that influence F0 need to be converted. The standard technique for this is to introduce new features, one for each possible value in the class and output values of 0 or 1 for these modified features depending on the value of the base features. For example in a ToBI environment the output of the feature `tobi_accent` will include `H*`, `L*`, `L+H*` etc. In the modified form you would have features of the form `tobi_accent_H*`, `tobi_accent_L*`, `tobi_accent_L_H*`, etc.

The program `ols` in the speech tools takes feature files and description files in exactly the same format as `wagon`, except that all feature must be declared as type `float`. The standard ordinary least squares algorithm used to find the coefficients cannot, in general, deal with features that are directly correlated with others as this causes a singularity when inverting the matrix. The solution to this is to exclude such features. The option `-robust` enables that though at the expense of a longer compute time. Again like `file` a stepwise option is included so that the best subset of features may be found.

The resulting models may be used by the `Int_Targets_LR` module which takes its LR models from the variables `f0_lr_start`, `f0_lr_mid` and `f0_lr_end`. The output of `ols` is a list of coefficients (with the Intercept first). These need to be converted to the appropriate bracket form including their feature names. An example of which is in `festival/lib/f2bf0lr.scm`.

If the conversion of categoricals to floats seems to much work or would prohibitively increase the number of features you could use `wagon` to generate trees to predict F0 values. The advantage is that of a decision tree over the LR model is that it can deal with data in a non-linear fashion, But this is also the disadvantage. Also the decision tree technique may split the data sub-optimally. The LR model is probably more theoretically appropriate but ultimately the results depend on how goods the models sound.

Dump features as with the LR models, but this time there is no need convert categorical features to floats. A potential set of features to do this from (substitute `syl_midpitch` and `syl_endpitch` for the other two models is

```
syl_endpitch
pp.tobi_accent
p.tobi_accent
tobi_accent
n.tobi_accent
nn.tobi_accent
pp.tobi_endtone
R:Syllable.p.tobi_endtone
tobi_endtone
n.tobi_endtone
nn.tobi_endtone
pp.syl_break
```

```
p.syl_break
syl_break
n.syl_break
nn.syl_break
pp.stress
p.stress
stress
n.stress
nn.stress
syl_in
syl_out
ssyl_in
ssyl_out
asyl_in
asyl_out
last_accent
next_accent
sub_phrases
```

The above, of course assumes a ToBI accent labeling, modify that as appropriate for you actually labeling.

Once you have generated three trees predicting values for start, mid and end points in each syllable you will need to add some Scheme code to use these appropriately. Suitable code is provided in `src/intonation/tree_f0.scm` you will need to include that in your voice. To use it as the intonation target module you will need to add something like the following to your voice function

```
(set! F0start_tree f2b_F0start_tree)
(set! F0mid_tree f2b_F0mid_tree)
(set! F0end_tree f2b_F0end_tree)
(set! int_params
 '((target_f0_mean 110) (target_f0_std 10)
   (model_f0_mean 170) (model_f0_std 40)))
(Parameter.set 'Int_Target_Method Int_Targets_Tree)
```

The `int_params` values allow you to use the model with a speaker of a different pitch range. That is all predicted values are converted using the formula

```
(+ (* (/ (- value model_f0_mean) model_f0_stddev)
   target_f0_stddev) target_f0_mean)))
```

Or for those of you who can't real Lisp expressions

```
((value - model_f0_mean) / model_f0_stddev) * target_f0_stddev)+
  target_f0_mean
```

The values in the example above are for converting a female speaker (used for training) to a male pitch range.

Tilt modeling

Tilt modeling is still under development and not as mature as the other methods as described above, but it potentially offers a more consistent solution to the problem. A tilt parameterization of a natural F0 contour can be automatically derived from a waveform and a labeling of accent placements (a simple "*a*" for accents and "*b*" of boundaries) [taylor99]. Further work is being done on trying to automatically find the accents placements too.

For each "*a*" in an labeling four continuous parameters are found: height, duration, peak position with respect to vowel start, and tilt. Prediction models may then be generate to predict these parameters which we feel better capture the dimensions

of F0 contour itself. We have had success in building models for these parameters, [dusterhoff97a], with better results than the linear regression model on comparable data. However so far we have not done any tests with Tilt on languages other than English.

The speech tools include the programs `tilt_analyse` and `tilt_synthesize` to aid model building but we do not yet include fill Festival end support for using the generated models.

Duration

Like the above prosody phenomena, very simple solutions to predicting durations work surprisingly well, though very good solutions are extremely difficult to achieve.

Again the basic strategy is assigning fixed models, simple rules models, complex rule modules, and trained models using the features in the complex rule models. The choice of where to stop depends on the resources available to you and time you wish to spend on the problem. Given a reasonably sized database training a simple CART tree for durations achieves quite acceptable results. This is currently what we do for our English voices in Festival. There are better models out there but we have not fully investigated them or included easy scripts to customize them.

The simplest model for duration is a fixed duration for each phone. A value of 100 milliseconds is a reasonable start. This type of model is only of use at initial testing of a diphone database beyond that it sounds too artificial. The Festival function `SayPhones` uses a fixed duration model, controlled by the value (in ms) in the variable `FP_duration`. Although there is a fixed duration module in Festival (see the manual) its worthwhile starting off with something a little more interesting.

The next level for duration models is to use average durations for the phones. Even when real data isn't available to calculate averages, writing values by hand can be acceptable, basically vowels are longer than consonants, and stops are the shortest. Estimating values for a set of phones can be done by looking at data from another language, (if you are really stuck, see `festival/lib/mrpa_durs.scm}`, to get the basic idea of average phone lengths.

In most languages phones are longer at the phrase final and to a lesser extent phrase initial positions. A simple multiplicative factor can be defined for these positions. The next stage from this is a set of rules that modify the basic average based on the context they occur in. For English the best definition of such rules is the duration rules given in chapter 9, [allen87] (often referred to as the Klatt duration model). The factors used in this may also apply to other languages. A simplified form of this, that we have successfully used for a number of languages, and is often used as our first approximation for a duration rule set is as follows.

Here we define a simple decision tree that returns a multiplication factor for a segment

```
(set! simple_dur_tree
'
  ((R:SylStructure.parent.R:Syllable.p.syl_break > 1 ) ;; clause initial
   ((R:SylStructure.parent.stress is 1)
    ((1.5))
    ((1.2)))
   ((R:SylStructure.parent.syl_break > 1)   ;; clause final
    ((R:SylStructure.parent.stress is 1)
     ((1.5))
     ((1.2)))
    ((R:SylStructure.parent.stress is 1)
     ((ph_vc is +)
      ((1.2))
      ((1.0)))
```

```
((1.0))))))
```

You may modify this adding more conditions as much as you want. In addition to the tree you need to define the averages for each phone in your phone set. For reasons we will explain below the format of this information is "*segname 0.0 average*" as in

```
(set! simple_phone_data
'(
  (# 0.0 0.250)
  (a 0.0 0.080)
  (e 0.0 0.080)
  (i 0.0 0.070)
  (o 0.0 0.080)
  (u 0.0 0.070)
  (i0 0.0 0.040)
  ...
))
```

With both these expressions loaded in your voice you may set the following in your voice definition function. setting up this tree and data as the standard and the appropriate duration module.

```
;; Duration prediction
(set! duration_cart_tree simple_dur_tree)
(set! duration_ph_info simple_phone_data)
(Parameter.set 'Duration_Method 'Tree_ZScores)
```

Though in your voice use voice specific names for the `simple_` variables otherwise you may class with other voices.

It has been shown [campbell91] that a better representation for duration for modeling is *zscores*, that is number of standard deviations from the mean. The duration module used in the above is actually designed to take a CART tree that returns zscores and uses the information in `duration_ph_info` to change that into an absolute duration. The two fields after the phone name are mean and standard deviation. The interpretation of this tree and this phone info happens to give the right result when we use the tree to predict factors and have the stddev field contain the average duration, as we did above.

However no matter if we use zscores or absolutes, a better way to build a duration model is to train from data rather than arbitrarily selecting modification factors.

Given a reasonable sized database we can dump durations and features for each segment in the database. Then we can train a model using those samples. For our English voices we have trained regression models using `wagon`, though we include the tools for linear regression models too.

An initial set of features to dump might be

```
segment_duration
name
p.name
n.name
R:SylStructure.parent.syl_onsetsize
R:SylStructure.parent.syl_codasize
R:SylStructure.parent.R:Syllable.n.syl_onsetsize
R:SylStructure.parent.R:Syllable.p.syl_codasize
R:SylStructure.parent.position_type
R:SylStructure.parent.parent.word_numsyls
pos_in_syl
syl_initial
syl_final
R:SylStructure.parent.pos_in_word
p.seg_onsetcoda
```

```
seg_onsetcoda
n.seg_onsetcoda
pp.ph_vc
p.ph_vc
ph_vc
n.ph_vc
nn.ph_vc
pp.ph_vlng
p.ph_vlng
ph_vlng
n.ph_vlng
nn.ph_vlng
pp.ph_vheight
p.ph_vheight
ph_vheight
n.ph_vheight
nn.ph_vheight
pp.ph_vfront
p.ph_vfront
ph_vfront
n.ph_vfront
nn.ph_vfront
pp.ph_vrnd
p.ph_vrnd
ph_vrnd
n.ph_vrnd
nn.ph_vrnd
pp.ph_ctype
p.ph_ctype
ph_ctype
n.ph_ctype
nn.ph_ctype
pp.ph_cplace
p.ph_cplace
ph_cplace
n.ph_cplace
nn.ph_cplace
pp.ph_cvox
p.ph_cvox
ph_cvox
n.ph_cvox
nn.ph_cvox
R:SylStructure.parent.R:Syllable.pp.syl_break
R:SylStructure.parent.R:Syllable.p.syl_break
R:SylStructure.parent.syl_break
R:SylStructure.parent.R:Syllable.n.syl_break
R:SylStructure.parent.R:Syllable.nn.syl_break
R:SylStructure.parent.R:Syllable.pp.stress
R:SylStructure.parent.R:Syllable.p.stress
R:SylStructure.parent.stress
R:SylStructure.parent.R:Syllable.n.stress
R:SylStructure.parent.R:Syllable.nn.stress
R:SylStructure.parent.syl_in
R:SylStructure.parent.syl_out
R:SylStructure.parent.ssyl_in
R:SylStructure.parent.ssyl_out
R:SylStructure.parent.parent.gpos
```

By convention we build duration models in `festival/dur/`. We will save the above feature names in `dur.featnames`. We can dump the features with the command

```
dumpfeats -relation Segment -feats dur.featnames -output dur.feats \
    ../utts/*.utt
```

This will put all the features in the file `dur.feats`. For `wagon` we need to build a feature description file, we can build a first approximation with the `make_wagon_desc` script available with the speech tools

```
make_wagon_desc dur.feats dur.featnames dur.desc
```

You will then need to edit `dur.desc` to change a number of features from their categorical list (lots of numbers) into type `float`. Specifically for the above list the features `segment_duration`, `R:SylStructure.parent.parent.word_numsyls`, `pos_in_syl`, `R:SylStructure.parent.pos_in_word`, `R:SylStructure.parent.syl_in`, `R:SylStructure.parent.syl_out`, `R:SylStructure.parent.ssyl_in` and `R:SylStructure.parent.ssyl_out` should be declared as floats.

We then need to split the data into training and test sets (and further split the train set if we are going to use stepwise CART building.

```
traintest dur.feats
traintest dur.feats.train
```

We can no build a model using wagon

```
wagon -data dur.feat.train.train -desc dur.desc \
    -test dur.feats.train.test -stop 10 -stepwise \
    -output dur.10.tree
wagon_test -data dur.feats.test -tree dur.10.tree -desc dur.desc
```

You may wish to remove all examples of silence from the data as silence durations typically has quite a different distribution from other phones. In fact it is common that databases include many examples of silence which are not of natural length as they are arbitrary parts of the initial and following silence around the spoken utterances. Their durations are not something that should be trained for.

These instructions above will build a tree that predicts absolute values. To get such a tree to work with the zscore module simply make the stddev field above 1. As stated above using zscores typically give better results. Although the correlation of these duration models in the zscore domain may not be as good as training models predicting absolute scores when those predicted scores are convert back into the absolute domain we have found (for English) that the correlations are better, and RMSE smaller.

In order to train a zscore model you need to convert the absolute segment durations, to do that you need the means and standard deviations for each segment in your phoneset.

There is a whole branch of possible mappings for the distribution of durations: zscores, logs, logs-zscores, etc or even more complex functions [bellegarda98]. These variations do give some improvements. The intention is to map the distribution to a normal distribution which makes it easier to learn.

Other learning techniques, particularly Sums of Products model ([sproat98] chapter 5), which has been shown to training better even on small amounts of data.

Another technique, which although shouldn't work is to borrow a models trained for another language for which data is available. Actually the duration model used in Festival for the US and UK voices is the same, it was in fact trained from the f2b database, a US English database. As the phone sets are different for US and UK English we trained the models using phonetic features rather than phone names, and trained them in the zscore domain keeping the actual phone names and means and standard deviations separate. Although the models were slightly better if we included the phone names themselves, it was only slightly better and the models were also substantially larger (and took longer to train). Using the phonetic feature offers a

more general model (it works for UK English), more compact, quicker learning time and with only a small cost in performance.

Also in the German voice developed at OGI, the same English duration model was used. The results are acceptable and are at least better than any hand written rule system that could be written. Improvements in that model are probably only possible by training on real German data. Note however such cross language borrowing of models is unlikely to work in general but there may be cases where it is a reasonable fall back position.

Prosody Research

Note that the above descriptions are for the easy implementation of prosody models which unfortunately means that the models will not be perfect. Of course no models will be perfect but with some work it is often possible to improve the basic models or at least make them more appropriate to the synthesis task. For example if your intend use of your synthesis voice is primarily for dialog systems training one news caster speech will not give the best effect. Festival is designed as a research system as well as tool to build languages so it is well adapted to prosody research.

One thing which clearly shows off how imporoverished our prosodic models are is the comparing of predicted prosody with natural prosody. Given a label file and an F0 Target file the following code will generate\ that utterance using the current voice

```
(define (resynth labfile f0file)
  (let ((utt (Utterance SegF0))) ; need some u to start with
    (utt.relation.load utt 'Segment labfile)
    (utt.relation.load utt 'Target f0file)
    (Wave_Synth utt))
)
```

The format of the label file should be one that can be read into Festival (e.g. the XLabel format) For example

```
#
  0.02000 26  pau ;
  0.09000 26  ih ;
  0.17500 26  z ;
  0.22500 26  dh ;
  0.32500 26  ae ;
  0.35000 26  t ;
  0.44500 26  ow ;
  0.54000 26  k ;
  0.75500 26  ey ;
  0.79000 26  pau ;
```

The target file is a little more complex again it is a label file but with features "pos" and "F0" at each stage. Thus the format for a naturally rendered version of the above would be.

```
#
0.070000 124 0 ; pos 0.070000 ; f0 133.045230 ;
0.080000 124 0 ; pos 0.080000 ; f0 129.067890 ;
0.090000 124 0 ; pos 0.090000 ; f0 125.364600 ;
0.100000 124 0 ; pos 0.100000 ; f0 121.554800 ;
0.110000 124 0 ; pos 0.110000 ; f0 117.248260 ;
0.120000 124 0 ; pos 0.120000 ; f0 115.534490 ;
0.130000 124 0 ; pos 0.130000 ; f0 113.769620 ;
0.140000 124 0 ; pos 0.140000 ; f0 111.513180 ;
0.240000 124 0 ; pos 0.240000 ; f0 108.386380 ;
0.250000 124 0 ; pos 0.250000 ; f0 102.564100 ;
```

```
0.260000 124 0 ; pos 0.260000 ; f0 97.383600 ;
0.270000 124 0 ; pos 0.270000 ; f0 97.199710 ;
0.280000 124 0 ; pos 0.280000 ; f0 96.537280 ;
0.290000 124 0 ; pos 0.290000 ; f0 96.784970 ;
0.300000 124 0 ; pos 0.300000 ; f0 98.328150 ;
0.310000 124 0 ; pos 0.310000 ; f0 100.950830 ;
0.320000 124 0 ; pos 0.320000 ; f0 102.853580 ;
0.370000 124 0 ; pos 0.370000 ; f0 117.105770 ;
0.380000 124 0 ; pos 0.380000 ; f0 116.747730 ;
0.390000 124 0 ; pos 0.390000 ; f0 119.252310 ;
0.400000 124 0 ; pos 0.400000 ; f0 120.735070 ;
0.410000 124 0 ; pos 0.410000 ; f0 122.259190 ;
0.420000 124 0 ; pos 0.420000 ; f0 124.512020 ;
0.430000 124 0 ; pos 0.430000 ; f0 126.476430 ;
0.440000 124 0 ; pos 0.440000 ; f0 121.600880 ;
0.450000 124 0 ; pos 0.450000 ; f0 109.589040 ;
0.560000 124 0 ; pos 0.560000 ; f0 148.519490 ;
0.570000 124 0 ; pos 0.570000 ; f0 147.093260 ;
0.580000 124 0 ; pos 0.580000 ; f0 149.393750 ;
0.590000 124 0 ; pos 0.590000 ; f0 152.566530 ;
0.670000 124 0 ; pos 0.670000 ; f0 114.544910 ;
0.680000 124 0 ; pos 0.680000 ; f0 119.156750 ;
0.690000 124 0 ; pos 0.690000 ; f0 120.519990 ;
0.700000 124 0 ; pos 0.700000 ; f0 121.357320 ;
0.710000 124 0 ; pos 0.710000 ; f0 121.615970 ;
0.720000 124 0 ; pos 0.720000 ; f0 120.752700 ;
```

This file was generated from a waveform using the folloing command

```
pda -s 0.01 -otype ascii -fmax 160 -fmin 70 wav/utt003.wav |
awk 'BEGIN @{ printf("#\n") @}
    @{ if ($1 > 0)
       printf("%f 124 0 ; pos %f ; f0 %f ; \n",
            NR*0.010,NR*0.010,$1) @}' >Targets/utt003.Target
```

The utetrance may then be rendered as

```
festival> (set! utt1 (resynth "lab/utt003.lab" "Targets/utt003.utt"))
```

Note that this method will loose a little in diphone selection. If your diphone database uses consonant cluster allophones it wont be possible to properly detect these as there is no syllabic structure in this. That may or may not be important to you. Even this simple method however clearly shows how important the right prosody is to the understandability of a string of phones.

We have successfully done this on a number of natural utterances. We extracted the labels automatically by using the aligner discussed in the diphone chapter. As we were using diphones from the same speaker as the natural utterances (KAL) the alignment is surprisingly good and trivial to do. You must however synthesis the utterance first and save the waveform and labels. Note you should listen to ensure that the synthesizer has generated the right labels (as much as that is possible), including breaks in the same places. Comparing synthesized utterances with natural ones quickly shows up many problems in synthesis.

Prosody Walkthrough

This section gives a walkthrough of a set of basic scripts that can be used to build duration and F0 models. The results will be reasonable but they are designed to be language independent and hence more appropriate models will almost certainly give better results. We have used these methods when building diphone voices for new

languages when we know almost nothing explicit about the language structure. This walkthrough however explcitly covers most of the major steps and hence will be useful as a basis for building new better models.

In many ways this process is simialr to the limited domain voice building process. here we will design a set of prompts which are believed to cover the prosody that we wish to model, we record and label the data and then build models from the utterances built from the natural speech. In fact the basic structure for this uses the limited domain scripts for the initial part of the process.

The basic stages of this task are

- Design database
- Setup directory structure
- Synthesizing prompts (for labeling)
- Recording prompts
- Phonetically label prompts
- Extract pitchmarks and F0 contour
- Build utterance structures
- For duration models
  - extract means and standard deviations of phone durations
  - extract features for prediction
  - build feature description files
  - Build regression model to predict durations
  - construct scheme file with duration model
- For F0 models
  - extract features for prediction
  - build feature description files
  - Build regression model to preict F0 at start, mid and end of syllable
  - construct scheme file with F0 model

Design database

The object here is to cpature enough speech in prosodic style that you wish your syntehsizer to use. Note as prosodic modeling is still and extremely difficult area all models are extremely imporerished (especially the very simple models we are presenting here), thus do not be too ambitious. However it is worthwhile consider if you wish dialog (i.e. conversational speech) or prose (i.e. read speech). Prose can be news reader style or story telling style. Most synthesizers are trained on news reader style becuase its fairly consistent and believe to be easier to model, and reading paragraphs of text is seens as a basic apllication for text to speech synthesizers. However today with more dialog systems such prosodic models are often not as appropriate.

Ideally your database will be marked up with prosodic tagging that your voice talent will understand and be able to deliver appropriately. Designing such a database isn't easy but when starting off in new languages anything may be better than fixed durations and a naive declining F0. Thus simply a list of 500 sentences from newspapers may give rise to better models than.

Suppose you have your 500 sentences, construct a prompt list as is done with the limited domain constuction. That is, you need a file of the form.

```
( sent_0001 "She had your dark suit in greasy washwater all year.")
( sent_0002 "Don't make me carry an oily rag like that.")
( sent_0003 "They wanted to go on a barge trip.")
...
```

Setup directory structure

As with the rest of the festvox tools, you need to set the following to environment variables to allow them to work properly. In `bash` or other Bourne shell compatibles type, with the appropriate pathnames for you installation of the Edinburgh Speech Tools an Festvox itself.

```
export FESTVOXDIR=/home/awb/projects/festvox
export ESTDIR=/home/awb/projects/1.4.1/speech_tools
```

For `csh` and its derivative you should type

```
setenv FESTVOXDIR /home/awb/projects/festvox
setenv ESTDIR /home/awb/projects/1.4.1/speech_tools
```

As the basic structure is so similar to the limited domain building structure, first you should all that setup procedure. If you are building prosodic models for an already existing limited domain then you do not need this part.

```
mkdir cmu_timit_awb
cd cmu_timit_awb
$FESTVOXDIR/src/ldom/setup_ldom cmu timit awb
```

The arguments are, institution, domain type, and speaker name.

After setting this up you need to also setup the extra directories and scripts need to build prosody models. This is done by the command

```
$FESTVOXDIR/src/prosody/setup_prosody
```

You shold copy your database files as created in the previous section into `etc/`.

Synthesizing prompts

We then synthesizer the prompts. As we are trying to collect natural speech these prompts should not normally be presented to the voice talent as they may then copy the syntehsizer intonation, which would almost certainly be a bad thing. As this will sometimes be the first serious use of a new diphone syntehsizerin a new language, (with impoverished prosody models) it is important to check that the prompts can be generate phonetically correct. This may require more additions to the lexicon and/or more token to word rules. We synthesize the prompts for two reasons. First, to use for autolabeling in that the synthesized prompts will be aligned using dtw against what the speaker actually says. Second we are trying to construct festival utterances structures for each utterance in this database with natural durations and F0. so we may learn from them.

You should change the line setting the "closest" voice

```
(set! cmu_timit_awb::closest_voice 'voice_kal_diphone)
```

This is in the file `festvox/cmu_timit_awb_ldom.scm`. This is the voice that will be used to syntehsized the prompts. Often this will be your new diphone voice.

Ideally we would like these utterances to also have natural phone sequences, such that schwas, allophones such as flaps, and post-lexical rules have been applied. At present we do not include that here though for more serious prosody modeling such phonomena should be included in the utterance structures here.

The prompts can be synthesizer by the command

```
festival -b festvox/build_ldom.scm '(build_prompts "etc/timit.data")'
```

Recording the prompts

The usual caveats apply to recording, (the Section called *Recording under Unix* in Chapter 4) and the issues on selecting a speaker.

As prosody modeling is difficult, and if you are inexperienced in building such models, it is wise not to attempt anything hard. Just building reliable models for default unmarked intonation is very useful if your current models are simply the default fixed intonation. Thus the senetences should be read in a natural but not too varied style.

Recording can be done with `pointyclicky` or `prompt_them`. If you are using `prompt_them`, you should modify that script so that it does not play the prompts, as they will confuse the speaker. The speaker should simply read the text (and markup, if present).

```
pointyclicky etc/timit.data
```

or

```
bin/prompt_them etc/timit.data
```

Phonetically label prompts

After recording the spoken utterances must be labeled

```
bin/make_labs prompt-wav/*.wav
```

This is one of the computationally expensive parts of the process and for longer sentences it can require much memory too.

After autolabeling it is always worthwhiel to inspect the labels and correct mistakes. Phrasing can particularly cause problems so adding or deleting silences can make the derived prosody models much more accurate. You can use emulabel to to this.

```
emulabel etc/emu_lab
```

Extract pitchmarks and F0

At this point we diverge from the process used for building limited domain synthesizers. You can construct such synthesizers from the same recordings, maybe you wish more appropriate prosodic models for the fallback synthesizer. But at this poijnt we need to extract the pitchmark in a slightl different way. We are intending to extract F0 contours for all non-silence parts of the speech signal. We do this by extracting pitchmarks for the voiced sections alone then (in the next section) interpolating the F0 through the non-voiced (but non-silence) sections.

the Section called *Extracting pitchmarks from waveforms* in Chapter 4 discusses the setting of parameters to get `bin/make_pm_wave` to work for a particular voice. In this case we need those same parameters (which should be found by experiment). These shold be copied from `bin/make_pm_wave` and added to `bin/make_F0_pm` in the variable `PM_ARGS`. The distribution contains something like

```
PM_ARGS='-min 0.0057 -max 0.012 -def 0.01 -wave_end -lx_lf 140 -lx_lo 111 -lx_hf 80 -lx_ho 51 -
med_o 0'
```

Importnantly this differs from the parameters in `bin/make_pm_wave` as we do not use the `-fill` option to fill in pitchmarks over the rest of the waveform.

The second part of this section is the construction of an F0 contour which is build from the extracted pitchmarks. Unvoiced speech sections are assigned an F0 contour by interpolation from the voiced section around it, and the result is smnoothed. The label files are used to define which parts of the signal are silence and which are speech.

The variable `SILENCE` in `bin/make_f0_pm` must be modified to reflect the symbol used for silence in your phoneset.

Once the pitchmark parameters have be determined, and the appropriate `SILENCE` value set you can extract the smoothed F0 by the command

```
bin/make_f0_pm wav/*.wav
```

You can view the F0 contrours with the command

```
emulabel etc/emu_f0
```

Build utterance structures

With the labels and F0 created we can now rebuild the utterance structures by syntehsizing the prompt snad merging in the values from the natural durations and F0 from the naturally spoken utterances.

```
festival -b festvox/build_ldom.scm '(build_utts "etc/timit.data")'
```

Duration models

The script `bin/make_dur_model` contains all of the following commands but it is wise to understand the stages as due to errors in labeling it may not all run completely smoothly and small fixes may be required.

We are building a duration model using a CART tree to predict zscore values for phones. Zscores (number of standard deviations from the mean) have often been used in duration modeling as they allow a certain amount of normalization over different phones.

You shold first look at the script `bin/make_dur_model` and edit the following three variable values

```
SILENCENAME=SIL
VOICENAME='(kal_diphone)'
MODELNAME=cmu_us_kal
```

these should contain the name for silence in your phoneset, the call for the voice you are building the model for (or at least one that uses the same phoneset), and finally the name for the model, which can be the same `INST_LANG_VOX` part of the voice you call.

The first stage is to find the means and standard deviations for each phone. A festival script in the festival distribution is used to load in all the utetrances and a calculate these values. With the command

```
durmeanstd -output festival/dur/etc/durs.meanstd festival/utts/*.utt
```

You should check `festival/dur/etc/durs.meanstd`, the generated file to ensure that the numbers look raosnable. If there is only one example of a particular phone, the standard deviation cannot be calculated and the value is given as `nan` (not-a-number). Thus must be changed to a standard numeric value (say one-third or the mean). Also some of the values in this table maybe adversely affected by bad labeling so you may wish to hand modify the values, or go back and correct the labeling.

The next stage is extract the features from which we will predict the durations. The list of features extracted as in `festival/dur/etc/dur/feats`. These cover phonetic context, syllable, word position etc. These may or may not be appropriate for your new language or domain and you you may wish to add to these before doing the extraction. The extraction process takes each phoneme and dumps the named feature values for that phone into a file. This uses the standard festival script `dumpfeats` to do this. The command looks like

```
$DUMPFEATS -relation Segment -eval $VOICENAME \
    -feats festival/dur/etc/dur.feats =
    -output festival/dur/feats/%s.feats \
    -eval festival/dur/etc/logdurn.scm \
    festival/utts/*.utt
```

These feature files are then concatenated into a single file which is then split (90/10) into traing and test sets. The training set is further split force use as a held-out testset used in the training phase. Also at this stage we remove all silence phones form the training and test set. This is, perhaps naively, because the distribution of silences is very wide and often files contain silences at the start and end of utterances which themslves aren't part of the speech content (they're just the edges) and having these in the training set can skew the results.

This is done by the commands

```
cat festival/dur/feats/*.feats | \
    awk '{if ($2 != "'$SILENCENAME'") print $0}' >festival/dur/data/dur.data
bin/traintest festival/dur/data/dur.data
bin/traintest festival/dur/data/dur.data.train
```

For `wagon` the CART tree builder to work it needs to know what possible values each feature can take. This can mostly be determined automatically but some features may

have values that could be either numeric or classes, thus we use a post-processing function on the automatically generated description file to get our desired result.

```
$ESTDIR/bin/make_wagon_desc festival/dur/data/dur.data \
     festival/dur/etc/dur.feats festival/dur/etc/dur.desc
festival -b --heap 2000000 festvox/build_prosody.scm \
    $VOICENAME '(build_dur_feats_desc)'
```

Now we can build the model itself. A key factor in the time this takes (and the accuracy of the model) is the "stop" value, that is the number of examples that must exist before a split searched for. The smaller this number the longer the search will be, though up to a certasin point the more accurate the model will be. But at some level this will over train. The default in the distribution is 50 which may or may not be appropriate. Not for large databases and for smaller values of `STOP` the training may take days even on a fast processor.

Although we have guessed a reasonable value for this for databases of around 50-1000 utterances it may not be appropriate for you.

The learning technique used is basic CART tree growing but with an important extention which makes the process much more robust on unseen data but unfortunately much more computationally expensive. The `-stepwise` option on wagon incrementally searches for the best features to use in building three, in addition to at each iteration finding the best questions about each feature that best model data. If you want a quicker result removing the `-stepwise` option will give you that.

The basic wagon command is

```
wagon -data festival/dur/data/dur.data.train.train \
    -desc festival/dur/etc/dur.desc \
    -test festival/dur/data/dur.data.train.test \
    -stop $STOP \
    -output festival/dur/tree/$PREF.S$STOP.tree \
    -stepwise
```

To tets the results on data not used inthe training we use the command

```
wagon_test -heap 2000000 -data festival/dur/data/dur.data.test \
        -desc festival/dur/etc/dur.desc \
        -tree festival/dur/tree/$PREF.S$STOP.tree
```

Interpreting the results isn't easy in isolation. The smaller the RMSE (root mean squared error) the better and the larger the correlation is the better (it should never be greater than 1, and should never be below 0, though if you model is very bad it can be below 0). For English, with this script on a Timit database we get an RMSE value of 0.81 and correlation of 0.58, on the test data. Note these values are not in the abosolute domain (i.e. seconds) they are in the zscore domain.

The final stage, probabaly after a number of iterations of the build process we must package model into a scheme file that can be used with a voice. This scheme file contains the means and standard deviations (so we can convert the predicted values back into seconds) and the prediction tree itself. We also add in predictions for the silence phone by hand. The comamnd to generate this is

```
festival -b --heap 2000000 \
      festvox/build_prosody.scm $VOICENAME \
      '(finalize_dur_model "'$MODELNAME'" "'$PREF.S$STOP.tree'")'
```

This will generate a file `festvox/cmu_timit_awb_dur.scm`. If you model name is the same as the basic diphone voice you intend to use it in you cna simply copy this file to the `festvox/` directory of your diphone voice and it will automatically work. But it is worth explaining what this install process really is. The duration model scheme file contains two lisp expression setting the the variables `MODELNAME::phone_durs` and `MODELNAME::zdurtree`. To uses these in a voice you must load this file, typically by adding

```
(require 'MODELNAME_dur)
```

to the diphone voice definition file (`festvox/MODELNAME_diphone.scm`). And then get the voice defintion to use these new variables. This is done by the commands in the voice definition function

```
;; Duration prediction
(set! duration_cart_tree MODELNAME::zdurtree)
(set! duration_ph_info MODELNAME::phone_durs)
(Parameter.set 'Duration_Method 'Tree_ZScores)
```

F0 contour models

(what about accents ?)

extract features for prediction build feature description files Build regression model to preict F0 at start, mid and end of syllable construct scheme file with F0 model

Chapter 9. Corpus development

This chapter discusses the techniques used to design a good corpus for recording for use in general speech synthesis. The basic requirements for a speech synthesis corpus are:

- Phonetically and prosodically balanced
- Targeted toward the intended domain(s)
- Easy to say by voice talent without mistakes
- Short enough for the voice talent to be willing to say it.

To make life easier in designing prompt list we have included scripts which go some way to aid the process. The general idea is to take a very large amount of text and automatically find "nice" utterances that match these criteria.

The CMU ARCTIC Database prompt list [kominek 2003] was created very much in this way, though with an earlier version of the scripts.

As with most of our work there is a single script, that does a number of basic stages. The default is reasonable in many cases, but with prompt selection it is always worth hand checking and potentially modifying and refining the process.

The basic idea is to limit the chosen utterances to those of a reasonable length (5-15 words), only choose sentences with high frequency words (which should be easier to say and less ambiguous in pronunciation, also restrict to words that are in the lexicon (avoiding letter to sound issues).

The script `make_nice_prompts` is set up for two classes of language, Latin script languages and non-Latin script languages. Though as the non-Latin case is much more varied you may need to modify things. We have successfully used it for UTF-8 encoded Hindi.

For the Latin-script languages (as opposed "asis" cases) we downcase the text when looking for variation. Although some Latin based language make a significant case distinction, e.g. German, this is a reasonable route to avoid sentences with too many repeated words in them.

First gather lots of text data. When we say lots we mean at least millions of words, or even 10s on millions of words. This basic selection process is aimed at getting sentences for general voices and hence as large amount of starting data as possible is important.

Please also note the copyright of the data you are selecting from. In CMU ARCTIC we used out-of-copyright texts from the Gutenberg project, so there would be no issue in distributing the data. Copyright law in many countries allows for small subsets for copyright data, but this fair use is often argued by some. There may note actually be a good solution to this, News stories, are typically copyright but the press agency releasing them. Licenses on LDC data are often sufficient for using such texts to build prompts and then having no restriction on the voices generated, but the database itself may be under question. If you care about distribution (free or selling) you will need to address these issues.

The first stage once you have collected you data is check its encoding. Make sure its all the same encoding. Also check its reasonable. For example the Europal data is nice and clean (as conditioned for Machine Translation models) but the punctuation has been separated from the words. You may want to to de-htmlify your data before passing it to the selection routines.

Once you have a set of nice relatively clean data, you use the first stage of the script. This finds the word frequencies of all the tokens in the text data.

```
$FESTVOXDIR/src/promptselect/make_nice_prompts find_freq TEXT0 TEXT1 TEXT2
```

Give all the text files as arguments to this script. The working files will be created in the current directory, but the text file arguments may be pathnames.

The next stage is to build a Festival lexicon for the most frequent words. By default we select the top 5000 words, which has proven a reasonable choice. You can optionally override the 5000 with an argument.

```
$FESTVOXDIR/src/promptselect/make_nice_prompts make_freq_lex
```

The next stage processes each sentence in the large text database to find those utterances that are "nice". That of reasonable length, has only words in the frequency lexicon, no strange punctuation, capitals at the beginning, and punctuation at the end, and a few other heuristic rule conditions. These seem to work well for the Latin-script languages (though it is possible the conditions are overly strict for some languages).

Although Festival's text front end is used for processing the text, you do not need to build any language specific text front end (at least not normally). Finding nice prompts is considered one of the very first parts of buildings support for a new language, so we are aware that there will be almost no resources available for the target language yet.

Because this process is using Festival's front end, it is not fast, as it needs to process the whole text database. It is not unusual for this to take a number of hours to process. While processing "nice" utterances are written to `data_nice.data`. You should check this regularly in case there is some inappropriate condition in the rules and you are getting the wrong type of data.

```
$FESTVOXDIR/src/promptselect/make_nice_prompts find_nice TEXT0 TEXT1 ...
```

Note this will only search for the first 100,000 nice utterances, from the data set, you can change the number in the script if you want more (or less).

Once the "nice" utterances are found you can now find those nice utterances that have the best phonetic coverage. There are two mechanism available here. Because this is often the very first stage in building support for a new language, no lexicon and phonetics are available, thus selecting based on phonetic is not an option. Therefore we provide a simpler technique that selections based on letter coverage (in fact di-letter coverage). This is often a reasonable solution, but it will depend on the language whether this is reasonable solution or not. Note that for English, in spite of its somewhat poor relationship between orthography and pronunciation, this is reasonable, so don't exclude this as a possibility without trying it.

Letter selection will find the subset of the nice utterances that has the best letter coverage. It is a greedy algorithm, but this is usually sufficient. This process also takes a number to define the number of utterances it is looking for. The process will be applied multiple times to the remaining data until that number is reached. If there isn't enough data to select from it might loop for ever. By default it looks for 1000 utterance, which is not unreasonable for a unit selection voices, 500 is probably sufficient for a CLUSTERGEN voice. But, as they say, your mileage may vary.

```
$FESTVOXDIR/src/promptselect/make_nice_prompts select_letter_n
```

If you do have a pronunciation lexicon for you language, you can also do select based on segments rather than letters. We have not done exhaustive comparisons of how valuable segment selection is over letter selection. It is clear that although probably important, it is probably less important that selecting a good voice talent, or recording the prompts in a high quality manner. Two stages are required for segment selection. The first stage is to render the nice prompts from words to segments

```
$FESTVOXDIR/src/promptselect/make_nice_prompts synth_seg
```

Then greedily select the utterances with the best di-phone coverage.

```
$FESTVOXDIR/src/promptselect/make_nice_prompts select_seg
```

We do not yet support `select_seg_n`.

After selection the nice prompts will me in `data.done.data`. Look at it. Do not expect it to be perfect. I have never done this for a new language, without having to do it multiple times until I get something reasonable. Even once you have the result, it is worth while checking each utterance and correcting and/or rejecting it for other reasons, such as ungrammatical, hard to read, ambiguous words etc. Be bold and get rid of weird sentences, it will save you trouble later. The selection process is deliberately designed to have redundancy as speech is a variable medium and we can never be sure what exact the voice talent will say, or how the unit selection process will select the units from the database.

It is wise to first go through every sentence and attempt to record it and at that time decide if the sentence is actually a reasonable utterance to include in the prompt set for that language.

The final stage extracts the vocabulary of the selected prompt set. You can use this vocab list to start building your pronunciation lexicon as you will need that to build your speech databases (unless you are using an orthography based selection technique).

```
$FESTVOXDIR/src/promptselect/make_nice_prompts find_vocab
```

You can also do the whole process with the command

```
$FESTVOXDIR/src/promptselect/make_nice_prompts do_all TEXT0 TEXT1 ...
```

As really the `find_nice` stage takes up about 98% of processing time, redoing the other parts each time isn't unreasonable.

Non-Latin-script languages

For non-Latin-script languages there are options that seem to work well, if the language has spaces between words. We have used this quite extensively for UTF-8 encoded languages (Arabic and Hindi). For these language use

```
$FESTVOXDIR/src/promptselect/make_nice_prompts select_seg
$FESTVOXDIR/src/promptselect/make_nice_prompts find_freq_asis TEXT0 TEXT1 ...
$FESTVOXDIR/src/promptselect/make_nice_prompts make_freq_lex
$FESTVOXDIR/src/promptselect/make_nice_prompts find_nice_asis TEXT0 TEXT1 ...
$FESTVOXDIR/src/promptselect/make_nice_prompts select_letter_n
$FESTVOXDIR/src/promptselect/make_nice_prompts find_vocab_asis
```

You can also do the whole process with the command

```
$FESTVOXDIR/src/promptselect/make_nice_prompts do_all_asis TEXT0 TEXT1 ...
```

For languages that do not have spaces between the words (Chinese, Japanese, Thai etc), the above techniques will not work. We have used the above techniques for Chinese, by first segmenting the data into words.

Chapter 10. Waveform Synthesis

This part of the book discusses the techniques required to actually create the speech waveform from a complete phonetic and prosodic description.

Traditionally we can identify three different methods that are used for creating waveforms from such descriptions.

Articulatory synthesis is a method where the human vocal tract, tongue, lips, etc are modeled in a computer. This method can produce very natural sounding samples of speech but at present it requires too much computational power to be used in pratical systems.

Formant synthesis is a technique where the speech signal is broken down into iverlapping parts, such as formats (hence the name), voicing, asperation, nasality etc. The output of individual generators are then composed from streams of parameters describing the each sub-part of the speech. With carefully tuned parameters the quality of the speech can be close to that of natural speech, showing that this encoiding is sufficient to represent human speech. However automatic prediction of these parameters is harder and the typical quality of synthesizers based on this technology can be very understandable but still have a distinct no-human or robotic quality. Format synthesizers are best typified MITalk, the work of Dennis Klatt, and what was later commercialised as DECTalk. DECTalk is at present probably the most familar form of speech synthesis to the world. Dr Stephen Hawkins, Lucindian Professor of Mathematics at Cambridge University, has lost the use of his voice and uses a formant synthesizer for his speech.

Concatenative synthesis is the third form which we will spend the most time on. Here waveforms are created by concatenating parts of natural speech record from humans.

In the search for more natural synthesis there has been a move to use recorded human speech rather than techniques to construct it from its fundamental parts. The potential for using recorded speech has in some sense been made possible by the increase in power of computer and storage media. Format synthesizers require a relatively small amount of data but concatenative speech synthesizer typically require much more disk space to contain the inventory of speech sounds. And more recently the size databases used has grown further as there is some relationhip between voice quality and database size.

The techniques described here and the following chapters are concerned solely of concatenative synthesis. Concatenative synthesis techniques not only give the most natural sounding speech synthesis, it also is the most accessible to the general user in that it is quite easy for us to record speech and the technique used here to analyse it are, to the most part, automatic.

The area of concatenative systems can be viewed as a complex continunium, as there are many choices in selecting unit type, size etc. We have included examples and techniques from the most conservative (i.e. most likely to work) to the forefront of the art of voice building. It is important to understand the space of possible synthesis techniques in order to select the best one for your particular application. The resources required to develop each of the basic types of concatenative synthesizers varies greatly. It is possible to get a general speech synthesizer working in English in under an hour, though its quality isn't very good (though can be understandable). At the other extreme, in the area of speech synthesis we have yet to develop a system that is both natural and flexible to satisfy all synthesis requirements so the task of building a voice may take a lifetime. However the following chapter outline techniques which can be completed by an interest person in as little a day or at most a week which can produce high, natural sounding voices suitable for a wide range of computer speech applications.

In order for synthesis of any piece of text we need to have examples of every unit in the language to be synthesized. At some extreme this would mean you'd need recordings of every sentence (or even paragraph) of everything that needs to be said. This of course is impractical and defeats the whole purpose of a having a synthe-

sizer. Thus we need to make some simplifying assumptions. The simplest (and most extreme) is to assume that speech is made of up strings of discrete phonemes. US English has (by one definiton) 43 different phonemes. That is the fundamental sounds in the language, thus the word, "bit" is made up of three phones /B/ /IH/ and /T/. The word "beat" however is made up of the phonemes /B/ /IY/ and /T/.

In the following chapter we consider the absolute simplest waveform synthesizer that consists of recording each phone int he language and the resequencing them to form new words. Although this is a easy and quick synthesizer to build it is immediately obvious that it is not of very good quality. The reason being thate human speech just doesn't consist of isloated phonemes concatenated to gether but that there are articulatory effects that cross over multiple phones (co-articulation). Thus the more practical technique is to build a *diphone* synthesizer where we record each phoneme in the context of each other phoneme.

However speech is more varied than that although we can modify the selected diphones to obtain the desired prosody, such modification is not as good as if it were actual spoken by a human. Thus the area of general *unit selection* synthesis has grown where the datbases we select from has many more examples of speech, in more contexts and not just one example of each phoneme-phoneme transition in the language. The size and design of a databases most suitable for unit selection is difficult and we discuss this in the following chapter. The techniques required to find the most approrpiate unit, taking into account, phonetic context, word position, phrase position as well prosodic context is important but finding the right palance of these features is still something of an art. In Chapter 12 we present a number techniques and experiments to build such synthesizers.

Although *unit selection* synthesizer clear offer the best quality synthesis, their databases are substantial piece of work. They must be preoperly labeled and although we include automatic alignment techniques there will always be mistakes and hand correction is certainly both desirable and worthwhile. But that takes time and certain skills to do. When the unit selection is bad due to bad labels inappropriate weight of features or just simply not enough good examples int he dtaabase to choose from the quality can be serverely worse than diphones so the work in tuning a unit selection synthesizer is as much avoiding the bad examples as improving the good ones. The third chapter on waveform syntehsizers offers a very rpactical compramise between diphone (safe) synthesizers and unit selection (exciting) synthesizers. In Chapter 5 on limited domain synthesis, we discuss how to target your recorded database to a particular application and get the benefits of the high quality of unit selection synthesis without the requirement of very carefully labeled databases. In many case this thrid option is the most practical.

Of course it is possible to combine approaches. This requries more care in the design but sometimes offers the best of all techniques. Having a targeted limited domain synthesizer which can cover most of the desired language will be good but falling back on good unit selection synthesizer for unknown words may be a real posibility.

Key choices:
  size, type, prosodic modification, number of occurrences
Key positions in the space
  uniphones, diphones
  unit selection, limited domain vs open

Need diagram for space of synthesizers

Chapter 11. Diphone databases

This chapter describes the processes involved in designing, listing recording, and using a diphone database for a language.

Diphone introduction

The basic idea behind building diphone databases is to explicitly list all possible phone-phone transitions in a language. This makes the incorrect but practical and simplifying assumption that co-articulatory effects never go over more than two phones. The exact definition of *phone* here is in general nontrivial, and what a "standard" phone set should be is not uncontroversial -- various allophonic variations, such as light and dark /l/, may also be included. Unlike generalized unit selection where multiple occurrences of phones may exists with various distinguishing features, in a diphone database only one occurrence of each diphone is recorded. This makes selection much easier but also makes for a large laborious collection task.

In general, the number of diphones in a language is the square of the number of phones. However, in natural human languages, there are phonotactic constraints -- some phone-phone pairs, even whole classes of phones-phone combinations, may not occur at all. These gaps are common in the world's languages. The exact definition of *never exist* is also problematic. Humans can often generate those so-called non-existent diphones if they try, and one must always think about phone pairs that cross over word boundaries as well, but even then, certain combinations cannot exist; for example, /hh/ /ng/ in English is probably impossible (we would probably insert a schwa). /ng/ may really only appears after the vowel in a syllable (in coda position); however, in other languages it can appear in syllable-initial position. /hh/ cannot appear at the end of a syllable, though sometimes it may be pronounced when trying to add aspiration to open vowels.

Diphone synthesis, and more generally any concatenative synthesis method, makes an absolutely fixed choice about which units exist, and in circumstances where something else is required, a mapping is necessary. When humans are given a context where an unusual phone is desired, for example in a foreign word, they will (often) attempt to produce it even though it falls outside their basic phonetic vocabulary. The articulatory system is flexible enough to produce (or attempt to produce) unfamiliar phones, as we all share the same underlying physical structures. Concatenative synthesizers, however, have a fixed inventory, and cannot reasonably be made to produce anything outside their pre-defined vocabulary. Formant and articulatory synthesizers have the advantage here. This is a basic trade off, concatenative synthesizers typically produce much more natural synthesis than formants synthesizer but at the cost of being only able to produce those phones defined within their inventory.

Since we wish to build voices for arbitrary text-to-speech systems which may include unusual phones, some mapping, typically at the lexical level, can be used to ensure all the required diphones lie within the recorded inventory. The resulting voice will therefore be limited, and unusual phones will lie outside its range. This in many cases is acceptable though if the voice is specifically to be used for pronouncing Scottish place names it would be advisable to include the /X/ phone as in "loch".

In addition to the base phones, various allophonic variations may also be considered. Flapping, as when the /t/ becoming a /dx/ in the word "butter" is an example of an allophonic variation reduction which occurs naturally in American English, and including flaps in the phone set makes the synthetic speech more natural. Stressed and unstressed vowels in Spanish, consonant cluster /r/ verses lone /r/ in English, intersyllabic diphones verses intra-syllabic ones -- variations like these are well worth considering. Ideally, all such possible variations should be included in a diphone list, but the more variations you include, the larger the diphone set will be -- remember the general rule that the number of diphones is nearly the square of the number of phones. This affects recording time, labeling time and ultimately the database size.

Duplicating all the vowels (e.g. stressed/unstressed versions) will significantly increase the database size.

These inventory questions are open, and depending on the resources you are willing or able to devote, can be extended considerably. It should be clear, however, that such a list is simply a basic set. Alternative synthesis methods and inventories of different unit sizes may produce better results for the amount of work (or data collected). Demi-syllable databases and mixed inventory methods such as Hadifix [portele96] may give better results under some conditions. Still, controlling the inventory and using acoustic measures rather than linguistic knowledge to define the space of possible units in your inventory has also been attempted as in work like Whistler [huang97]. The most extreme view where the unit inventory is not predefined at all but based solely on what is available in general speech databases is CHATR [campbell96].

Although generalized unit selection can produce *much* better synthesis than diphone techniques, using more units makes selecting appropriate ones more difficult. In the basic strategy presented in this section, selection of the appropriate unit from the diphone inventory is trivial, while in a system like CHATR, selection of the appropriate unit is a significantly difficult problem. (See Chapter 12 on unit selection for more discussion of such techniques). With a harder selection task, it is more likely that mistakes will be made, which in unit selection can give some selections which are much worse worse that diphones, even though other examples may be better.

Defining a diphone list

Since diphones need to be cleanly articulated, various techniques have been proposed to elicit them from subjects. One technique is to use target words embedded carrier sentences to ensure that the diphones are pronounced with acceptable duration and prosody (i.e. consistently). We have typically used nonsense words that iterate through all possible combinations; the advantage of this is that you don't need to search for natural examples that have the desired diphone, the list can be more easily checked and the presentation is less prone to pronunciation errors than if real words were presented. The words look unnatural but collecting all diphones in not a particularly natural thing to do. See [isard86] or [stella83] for some more discussion on the use of nonsense words for collecting diphones.

For best results, we believe the words should be pronounced with consistent vocal effort, with as little prosodic variation as possible. In fact pronouncing them in a monotone is ideal. Our nonsense words consist of a simple carrier form with the diphones (where appropriate) being taken from a middle syllable. Except where schwa and syllabic consonants are involved that syllable should normally be a full stressed one.

Some example code is given in `src/diphone/darpaschema.scm`. The basic idea is to define classes of diphones, for example: vowel consonant, consonant vowel, vowel vowel and consonant consonant. Then define carrier contexts for these and list the cases. Here we use Festival's Scheme interpreter to generate the list though any scripting language is suitable. Our intention is that the diphone will come from a middle syllable of the nonsense word so it is fully articulated and minimize the articulatory effects at the start and end of the word.

For example to generate all vowel vowel diphone we define a carrier

```
(set! vv-carrier '((pau t aa t) (t aa pau)))
```

And we define a simple function that will enumerate all vowel vowel transitions

```
(define (list-vvs)
  (apply
   append
   (mapcar
```

```
(lambda (v1)
  (mapcar
   (lambda (v2)
     (list
      (string-append v1 "-" v2)
      (append (car vv-carrier) (list v1 v2) (car (cdr vv-carrier)))))
   vowels))
 vowels)))
```

For those of you who aren't used to reading Lisp this simple lists all possible combinations or in some potentially more readable format (in an imaginary language)

```
for v1 in vowels
  for v2 in vowels
    print pau t aa t $v1 $v2 t aa pau
```

The actual Lisp code returns a list of diphone names and phone string. To be more efficient, the DARPAbet example produces consonant-vowel and vowel-consonant diphones in the same nonsense word, which reduces the number of words to be spoken quite significantly. Your voice talent will appreciate this.

Although the idea seems simple to begin with, simply listing all contexts and pairs, there are other constraints. Some consonants can only appear in the onset of a syllable (before the vowel), and others are restricted to the coda.

While one can collect all the diphones without considering where they fall in a syllable, it often makes sense to collect diphones in different syllabic contexts. Consonant clusters are the obvious next set to consider; thus the example DARPAbet schema includes simple consonant clusters with explicit syllable boundaries. We also include syllabic consonants though these may be harder to pronounce in all contexts. You can add other phenomena too, but this is at the cost of not only making the list longer (and making it take longer to record), but making it harder to produce. You must consider how easy it is for your voice talent to pronounce them, and how consistent they can be about it. For example, not all American speakers produce flaps (/dx/) in all of the same contexts (and some may produce them even when you ask them not to), and its quite difficult for some to pronounce them, which can lead to production/transcription mismatches.

A second and related problem is language interference, which can cause phoneme crossover. Because of the prevalence of English, especially in electronic text, how many "foreign" phone should be considered for addition? For example, should /w/ be include for German speakers, (maybe), /t-i/ for Japanese (probably) or both /b/ and /v/ for Spanish speakers ("B de burro / V de vaca"). This problem is made difficult by the fact that the people you are recording will often be fluent or nearly fluent in English, and hence already have reasonably ability in phones that are not in their native language. If you are unfamiliar with the phone set and constraints on a language, it pays off considerably to either ask someone (like a linguist!) who knows the language analytically (not just by intuition), to check the literature, or to do some research.

To the degree that they are expected to appear, regardless of their status in the target language per se, foreign phones should be considered for the inventory. Remember that in most languages, nowadays, making no attempt to accommodate foreign phones is considered ignorant at least and possibly even arrogant.

Ultimately, when more complex forms are needed, extending the "diphone" set becomes prohibitive and has diminishing returns. Obviously there are phonetic differences between onset and coda positions, co-articulatory effects which go over more then one phone, stress differences, intonational accent differences, and phrase-positional difference to name but a few. Explicitly enumerating all of these, or even deciding the relative importance of each, is a difficult research question, and arguably

shouldn't be done in an abstract, linguistically generated fashion from a strict interpretation of the language. Identifying these potential differences and finding an inventory which takes into account the actual distinctions a speaker makes is far more productive and is the fundamental part of many new research directions in concatenative speech synthesis. (See the discussion in the introduction above).

However you choose to construct the diphone list, and whatever examples you choose to include, the the tools and scripts included with this document require that it be in a particular format.

Each line should contain a file id, a prompt, and a diphone name (or list of names if more than one diphone is being extracted from that file). The file id is used to in the filename for the waveform, label file, and any other parameters files associated with the nonsense word. We usually make this distinct for the particular speaker we are going to record, e.g. their initials and possibly the language they are speaking.

The prompt is presented to the speaker at recording time, and here it contains a string of the phones in the nonsense word from which the diphones will be extracted. For example the following is taken from the DARPAbet-generated list

```
( uk_0001 "pau t aa b aa b aa pau" ("b-aa" "aa-b") )
( uk_0002 "pau t aa p aa p aa pau" ("p-aa" "aa-p") )
( uk_0003 "pau t aa d aa d aa pau" ("d-aa" "aa-d") )
( uk_0004 "pau t aa t aa t aa pau" ("t-aa" "aa-t") )
( uk_0005 "pau t aa g aa g aa pau" ("g-aa" "aa-g") )
( uk_0006 "pau t aa k aa k aa pau" ("k-aa" "aa-k") )
...
( uk_0601 "pau t aa t ey aa t aa pau" ("ey-aa") )
( uk_0602 "pau t aa t ey ae t aa pau" ("ey-ae") )
( uk_0603 "pau t aa t ey ah t aa pau" ("ey-ah") )
( uk_0604 "pau t aa t ey ao t aa pau" ("ey-ao") )
...
( uk_0748 "pau t aa p - r aa t aa pau" ("p-r") )
( uk_0749 "pau t aa p - w aa t aa pau" ("p-w") )
( uk_0750 "pau t aa p - y aa t aa pau" ("p-y") )
( uk_0751 "pau t aa p - m aa t aa pau" ("p-m") )
...
```

Note the explicit syllable boundary marking - for the consonant-consonant diphones is used to distinguish them from the consonant cluster examples that appear later.

Synthesizing prompts

To help keep pronunciation consistent we suggest synthesizing prompts and playing them to your voice talent at collection time. This helps the speaker in two ways -- if they mimic the prompt they are more likely to keep a fixed prosodic style; it also reduces the number of errors where the speaker vocalizes the wrong diphone. Of course for new languages where a set of diphones doesn't already exist, producing prompts is not easy, however giving approximations with diphone sets from other languages may work. The problem then is that in producing prompts from a different phone set, the speaker is likely to mimic the prompts hence the diphone set will probably seem to have a foreign pronunciation, especially for vowels. Furthermore, mimicing the synthesizer too closely can remove some of the speaker's natural voice quality, which is under their (possibly subconscious) control to some degree.

Even when synthesizing prompts from an existing diphone set, you must be aware that that diphone set may contain errors or that certain examples will not be synthesized appropriately (e.g. consonant clusters). Because of this, it is still worthwhile monitoring the speaker to ensure they say things correctly.

The basic code for generating the prompts is in `src/diphone/diphlist.scm`, and a specific example for DARPA phone set for American English in

`src/diphone/us_schema.scm`. The prompts can be generated from the diphone list as described above (or at the same time). The example code produces the prompts and phone labels files which can be used by the aligning tool described below.

Before synthesizing, the function `Diphone_Prompt_Setup` is called, if it has been defined. You should define this to set up the appropriate voices in Festival, as well as any other initialization you might need -- for example, setting the fundamental frequency (F0) for the prompts that are to be delivered in a monotone (disregarding so-called microprosody, which is another matter). This value is set through the variable `FP_F0` and should be near the middle of the range for the speaker, or at least somewhere comfortable to deliver. For the DARPAbet diphone list for KAL, we have:

```
(define (Diphone_Prompt_Setup)
 "(Diphone_Prompt_Setup)
Called before synthesizing the prompt waveforms.  Uses the kal_dphone
voice for prompting and sets F0."
 (voice_kal_diphone)  ;; US male voice
 (set! FP_F0 90)      ;; lower F0 than ked
 )
```

If the function `Diphone_Prompt_Word` is defined, it will be called after the basic prompt-word utterance has been created, and before the actual waveform synthesis. This may be used to map phones to other phones, set durations or whatever you feel appropriate for your speaker/diphone set. For the KAL set, we redefined the syllabic consonants to their full consonant forms in the prompts, since the ked diphone database doesn't actually include syllabics. Also, in the example below, instead of using fixed (100ms) durations we make the diphones use a constant scaling factor (here, 1.2) times the average duration of the phones.

```
(define (Diphone_Prompt_Word utt)
  "(Diphone_Prompt_Word utt)
Specify specific modifications of the utterance before synthesis
specific to this particular phone set."
  ;; No syllabics in kal so flip them to non-syllabic form
  (mapcar
   (lambda (s)
     (let ((n (item.name s)))
       (cond
        ((string-equal n "el")
         (item.set_name s "l"))
        ((string-equal n "em")
         (item.set_name s "m"))
        ((string-equal n "en")
         (item.set_name s "n")))))
   (utt.relation.items utt 'Segment))
  (set! phoneme_durations kd_durs)
  (Parameter.set 'Duration_Stretch '1.2)
  (Duration_Averages utt))
```

By convention, the prompt waveforms are saved in `prompt-wav/`, and their labels in `prompt-lab/`. The prompts may be generated after the diphone list is given using the following command:

```
festival festvox/us_schema.scm festvox/diphlist.scm
festival> (diphone-gen-schema "us" "etc/usdiph.list")
```

If you already have a diphone list schema generated in the file `etc/usdiphlist`, you can do the following

```
festival festvox/us_schema.scm festvox/diphlist.scm
festival> (diphone-gen-waves "prompt-wav" "prompt-lab" "etc/usdiph.list")
```

Another useful example of the setup functions is to generate prompts for a language for which no synthesizer exists yet -- to "bootstrap" from one language to another. A simple mapping can be given between the target phoneset and an existing synthesizer's phone set. We don't know if this will be sufficient to actually use as prompts, but we have found that it is suitable to use these prompts for automatic alignment.

The example here is using the `voice_kal_diphone` speaker, a US English speaker, to produce prompts for japanese phone set, this code is in `src/diphones/ja_schema.scm`

The function `Diphone_Prompt_Setup` calls the kal (US) voice, sets a suitable F0 value, and sets the option `diph_do_db_boundaries` to `nil`. This option allows the diphone boundaries to be dumped into the prompt label files, but this doesn't work when cross-language prompting is done, as the actual phones don't match the desired ones.

```
(define (Diphone_Prompt_Setup)
 "(Diphone_Prompt_Setup)
Called before synthesizing the prompt waveforms.  Cross language prompts
from US male (for gaijin male)."
 (voice_kal_diphone)  ;; US male voice
 (set! FP_F0 90)
 (set! diph_do_db_boundaries nil) ;; cross-lang confuses this
 )
```

At synthesis time, each Japanese phone must be mapped to an equivalent (one or more) US phone. This is done though a simple table. set in `nhg2radio_map` which gives the closest phone or phones for the Japanese phone (those unlisted remain the same).

Our mapping table looks like this

```
(set! nhg2radio_map
     '((a aa)
 (i iy)
 (o ow)
 (u uw)
 (e eh)
 (ts t s)
 (N n)
 (h hh)
 (Qk k)
 (Qg g)
 (Qd d)
 (Qt t)
 (Qts t s)
 (Qch t ch)
 (Qj jh)
 (j jh)
 (Qs s)
 (Qsh sh)
 (Qz z)
 (Qp p)
 (Qb b)
 (Qky k y)
 (Qshy sh y)
 (Qchy ch y)
 (Qpy p y ))
 (ky k y)
 (gy g y)
 (jy jh y)
```

```
(chy ch y)
(shy sh y)
(hy hh y)
(py p y)
(by b y)
(my m y)
(ny n y)
(ry r y)))
```

Phones that are not explicitly mentioned map to themselves (e.g. most of the consonants).

Finally we define `Diphone_Prompt_Word` to actually do the mapping. Where the mapping involves more than one US phone we add an extra segment to the Segment (defined in the Festival manual) relation and split the duration equally between them. The basic function looks like

```
(define (Diphone_Prompt_Word utt)
  "(Diphone_Prompt_Word utt)
Specify specific modifications of the utterance before synthesis
specific to this particular phone set."
  (mapcar
   (lambda (s)
     (let ((n (item.name s))
    (newn (cdr (assoc_string (item.name s) nhg2radio_map))))
       (cond
  ((cdr newn)  ;; its a dual one
   (let ((newi (item.insert s (list (car (cdr newn))) 'after)))
     (item.set_feat newi "end" (item.feat s "end"))
     (item.set_feat s "end"
      (/ (+ (item.feat s "segment_start")
     (item.feat s "end"))
        2))
     (item.set_name s (car newn))))
  (newn
   (item.set_name s (car newn)))
  (t
   ;; as is
   ))))
    (utt.relation.items utt 'Segment))
  utt)
```

The label file produced from this will have the original desired language phones, while the acoustic waveform will actually consist of phones in the target language. Although this may seem like cheating, we have found this to work for Korean and Japanese from English, and is likely to work over many other language combination pairs. For autolabeling as the nonse word phone names are pre-defined alignment just needs to be the best matching path and as long as the phones are distinctive from the ones around them this alignment method is likely to work.

Recording the diphones

The object of recording diphones is to get as uniform a set of pronunciations as possible. Your speaker should be relaxed, not be suffering for a cold, or cough, or a hangover. If something goes wrong with the recording and some of the examples need to be re-recorded it is important that the speaker has as similar a voice as with the original recording, waiting for another cold to come along is not reasonable, (though some may argue that the same hangover can easily be induced). Also to try to keep the voice potentially repeatable it is wise to record at the same time of day, morning is a good idea. The points on speaker selection and recording in the previous section should also be borne in mind.

The recording environment should be reconstructable, so that the conditions can be set up again if needed. Everything should be as well-defined as possible, as far as gain settings, microphone distances, and so on. Anechoic chambers are best, but general recording studios will do. We've even done recording in an open room, with care this works (make sure there's little background noise from computers, air conditioning, outside traffic etc). Of course open rooms aren't ideal but they are better than open noisey rooms.

The distance between the speaker and the microphone is crucial. A head mounted mike helps keep this constant; the Shure SM-2 headset, for instance, works well with the mic positioned at 8mm from the lips or so. This can be checked with a ruler. Considering the cost and availability of headmounted microphones and rulers, you should really consider using them. While even fixed microphones like the Shure SM-57 can be used well by professional voice talent, we strongly recommend a good headset mic.

Ultimately, you need to split the recordings into individual files, one for each prompt. Ideally this can be done while recording on a file-by-file basis, but as that may not be practical and some other technique can be used, such as recording onto DAT and transferring the data to disk (and downsampling) later. Files might contain 50-100 nonsense words each. In this case we hand label the words, taking into account any duplicates caused be errors in the recording. The program `ch_wave` in the Edinburgh Speech Tools (EST) offers a function to split a large file into individual files based on a label file. We can use this to get our individual files. You may also add an identifiable noise during recording and automatically detect that as a split point, as is often done at the Oregon Graduate Instititute.. They typically use two different noises that can easily be distinguished and use one for "*OK*" and "*BAD*" this can make the splitting of the files into the individual nonsense words easier. Note you will also need to split the electroglottograph (EGG) signal exactly the same way, if you are using one.

No matter how you split these, you should be aware that there will still often be mistakes, and checking by listening will help.

We now almost always record directly to disk on a computer using a sound card; see the Section called *Recording under Unix* in Chapter 4 for recording setup details. There can be a reduction in the quality of the recording due to poor quality audio hardware in computers (and often too much noise), though at least the hardware issue is getting to be less of a problem these days. There are lots of advantages to recording directly to disk, as the stage of digitising, transfering and spliting the offline records is laborious and prone to error.

Labeling the diphones

Labeling nonsense words is *much* easier than labeling continuous speech, whether it is by hand or automatically. With nonsense words, it is completely defined which phones are there and they are (hopefully) clearly articulated.

We have had significant experience in hand labeling diphones, and with the right tools it can be done fairly quickly (e.g. 20 hours for 2500 nonsense words) even if it is a mind-numbing exercise which your voice talent may offer you little sympathy for after you've made them babble for hours in a box with electrodes on their throat (optional). But labeling can't realistically be done for more than an hour or two at any one time. As a minimum, the start of the preceding phone to the first phone in the diphone, the changeover, and the end of the second phone in the diphone should be labeled. Note we recommend phone boundary labeling as that is much better defined than phone middle marking. The diphone will, by default be extracted from the middle of phone one to the middle of phone two.

Your data set conventions may include the labeling of closures within stops explicitly. Thus you would expect the label `tcl` at the end of the silence part of a `/t/` and a label `t` after the burst. This way the diphone boundary can automatically be placed within the silence part of the stop. The label `DB` can be used when explicit diphone

boundaries are desireable; this is useful within phones such as diphthongs where the temporal middle need not be the most stable part.

Another place when specific diphone boundaries are recommended is in the phone-to-silence diphones. The phones at the end of words are typically longer than word internal phones, and tend to trail off in energy. Thus the midpoint of a phone immediately before a silence typically has much less energy than the midpoint of a word internal phone. Thus, when a diphone is to be concatenated to a phone-silence diphone, there would be a big jump in energy (as well as other related spectral characteristics). Our solution to this is explicitly label a diphone boundary near the beginning of the phone before the silence (about 20% in) where the energy is much closer to what it will be in the diphone that will precede it.

If you are using explicit closures, it is worth noting that stops at the start of words don't seem to have a closure part; however it is a good idea to actually label one anyway, if you are doing this by hand. Just "steal" a suitable short piece of silence from the preceding part of the waveform.

Because the words will often have very varying amounts of silence around them, it is a good idea to label multiple silences around the word, so that the silence immediately before the first phone is about 200-300 ms, and labeling the silence before that as another phone; likewise with the final silence. Also, as the final phone before the end silence may trail off, we recommend that the end of the last phone come at the very end of any signal thus appear to include silence within it. Then label the real silence (200-300 ms) after it. The reason for this is if the end silence happens to include some part of the spoken signal, and if this is duplicated, as is the case when duration is elongated, an audible buzz can be introduced.

Because labeling of diphone nonsense words is such a constrained task we have included a program for automatically providing a labeling for the spoken prompts. This requires that prompts be generated for the diphone database. The aligner uses those prompts to do the aligning. Though its not actually necessary that the prompts were used as prompts they do need to be generated for this alignment process. This is not the only means for alignment; you may also, for instance, use a speech recognizer, such as CMU Sphinx, to segment (align) the data.

The idea behind the aligner is to take the prompt and the spoken form and derive mel-scale cepstral parameterizations (and their deltas) of the files. Then a DTW (dynamic time warping) algorithm is used to find the best alignment between these two sets of features. Then the prompt label file is used to index through the alignment to give a label file for the spoken nonsense word. This is largely based on the techniques described in [malfrere97], though this general technique has been used for many years.

We have tested this aligner on a number of existing hand-labeled databases to compare the quality of the alignments with respect to the hand labeling. We have also tested aligning prompts generated from a language different from that being recorded. To do this there needs to be reasonable mapping between the language phonesets.

Here are results for automatically finding labels for the ked (US English) by aligning them against prompts generated by three different voices

ked itself

mean error 14.77ms stddev 17.08

mwm (US English)

mean error 27.23ms stddev 28.95

gsw (UK English)

mean error 25.25ms stddev 23.923

Note that gsw actually gives better results than mwm, even though it is a different dialect of English. We built three diphone index files from each of the label sets generated from there alignment processes. ked-to-ked was the best, and only marginally worse that the database made from the manually produced labels. The database from mwm and gsw produced labels were a little worse but not unacceptably so. Considering a significant amount of careful corrections were made to the manually produced labels, these automatically produced labels are still significantly better than the first pass of hand labels.

A further experiment was made across languages; the ked diphones were used as prompts to align a set of Korean diphones. Even though there are a number of phones in Korean not present in English (various forms of aspirated consonants), the results are quite usable.

Whether you use hand labeling or automatic alignment, it is always worthwhile doing some hand-correction after the basic database is built. Mistakes (sometimes systematic) always occur and listening to substantial subset of the diphones (or them all if you resynthesize the nonsense words) is definitely worth the time in finding bad diphones. The diva is in the details.

The script `festvox/src/diphones/make_labs` will process a set of prompts and their spoken (recorded) form generating a set of label files, to the best of its ability. The script expects the following to already exist

`prompt-wav/`

The waveforms as synthesized by Festival

`prompt-lab/`

The label files corresponding to the synthesized prompts in `prompt-wav`.

`prompt-cep/`

The directory where the cepstral feature streams for each prompt will be saved.

`wav/`

The directory holding the nonsense words spoken by your voice talent. The should have the same file id as the waveforms in `prompt-wav/`.

`cep/`

The directory where the cepstral feature streams for the recorded material will be saved.

`lab/`

The directory where the generated label files for the spoke words in `wav/` will be saved.

To run the script over the prompt waveforms

```
bin/make_labs prompt-wav/*.wav
```

The script is written so it may be use used in parallel on multiple machines if you want to distribute the process. On a Pentium Pro 200MHz, which you probably won't be able to find any more, a 2000 word diphone databases can be labeled in about 30 minutes. Most of that time is in generating the cepstrum coefficients. This is down to a few minutes at most on a dual Pentium III 550.

Once the nonsense words have been labeled, you need to build a diphone index. The index identifies which diphone comes from which files, and from where. This can be automatically built from the label files (mostly). The Festival script `festvox/src/diphones/make_diph_index` will take the diphone list (as used above), find the occurrence of each diphone in the label files, and build an index.

The index consists of a simple header, followed by a single line for each diphone: the diphone name, the fileid, start time, mid-point (i.e. the phone boundary) and end time. The times are given in *seconds* (note that early versions of Festival, using a different diphone synthesizer module, used milliseconds for this. If you have such an old version of Festival, it's time to update it).

An example from the start of a diphone index file is

```
EST_File index
DataType ascii
NumEntries 1610
IndexName ked2_diphone
EST_Header_End
y-aw kd1_002 0.435 0.500 0.560
y-ao kd1_003 0.400 0.450 0.510
y-uw kd1_004 0.345 0.400 0.435
y-aa kd1_005 0.255 0.310 0.365
y-ey kd1_006 0.245 0.310 0.370
y-ay kd1_008 0.250 0.320 0.380
y-oy kd1_009 0.260 0.310 0.370
y-ow kd1_010 0.245 0.300 0.345
y-uh kd1_011 0.240 0.300 0.330
y-ih kd1_012 0.240 0.290 0.320
y-eh kd1_013 0.245 0.310 0.345
y-ah kd1_014 0.305 0.350 0.395
...
```

Note the number of entries field must be correct; if it is too small it will (often confusingly) ignore the entries after that point.

This file can be created with a diphone list file and the lab files in by the command

```
$FESTVOXDIR/src/diphones/make_diph_index etc/usdiph.list dic/kaldiph.est
```

You should check that this has successfully found all the named diphones. When an diphone is not found in a label file, an entry with zeroes for the start, middle, and end is generated, which will produce a warning when being used in Festival, but it is worth checking in advance.

The `make_diph_index` program will take the midpoint between phone boundaries for the diphone boundary, unless otherwise specified with the label `DB`. It will also automatically remove underscores and dollar symbols from the diphone names before searching for the diphone in the label file, and it will only find the first occurrence of the diphone.

Extracting the pitchmarks

Festival, in its publically distributed form, currently only supports residual excited Linear-Predictive Coding (LPC) resynthesis [hunt89]. It does support PSOLA [moulines90], though this is not distributed in the public version. Both of these techniques are *pitch synchronous*, that is there require information about where pitch periods occur in the acoustic signal. Where possible, it is better to record with an electroglottograph (EGG, also known as a laryngograph) at the same time as the voice signal. The EGG records electrical activity in the glottis during speech, which makes it easier to get the pitch moments, and so they can be more precisely found.

Although extracting pitch periods from the EGG signal is not trivial, it is fairly straightforward in practice, as The Edinburgh Speech Tools include a program `pitchmark` which will process the EGG signal giving a set of pitchmarks. However it is not fully automatic and requires someone to look at the result and make some decisions to change parameters that may improve the result.

The first major issue in processing the signal is deciding which way is up. From our experience, we have seen the signal inverted in some cases and it is necessary to identify the direction in order for the rest of the processing to work properly. In general we've found the CSTR's LAR output is upside down while OGI's and CMU's output is the right way up, though this can even flip from file to file. If you find inverted signals, you should add `-inv` to the arguments to `pitchmark`.

The object is to produce a single mark at the peak of each pitch period and "fake" or "phantom" periods during unvoiced regions. The basic command we have found that works for us is

```
pitchmark lar/file001.lar -o pm/file001.pm -otype est \
   -min 0.005 -max 0.012 -fill -def 0.01 -wave_end
```

It is worth doing one or two by hand and confirming that a reasonable pitch periods are found. Note that the `-min` and `-max` arguments are speaker-dependent. This can be moved towards the fixed F0 point used in the prompts, though remember the speaker will not have been exactly constant. The script `festvox/src/general/make_pm` can be copied and modified (for the particular pitch range) and run to generate the pitchmarks

```
bin/make_pm lar/*.lar
```

If you don't have an EGG signal for your diphones, the alternative is to extract the pitch periods using some other signal processing function. Finding the pitch periods is similar to finding the F0 contour and, although harder than finding it from the EGG signal, with clean laboratory-recorded speech, such as diphones, it is possible. The following script is a modification of the `make_pm` script above for extracting pitchmarks from a raw waveform signal. It is not as good as extracting from the EGG file, but it works. It is more computationally intensive, as it requires rather high order filters. The value should change depending on the speaker's pitch range.

```
for i in $*
do
  fname=‘basename $i .wav‘
  echo $i
  $ESTDIR/bin/ch_wave -scaleN 0.9 $i -F 16000 -o /tmp/tmp$$.wav
  $ESTDIR/bin/pitchmark /tmp/tmp$$.wav -o pm/$fname.pm \
        -otype est -min 0.005 -max 0.012 -fill -def 0.01 \
        -wave_end -lx_lf 200 -lx_lo 71 -lx_hf 80 -lx_ho 71 -med_o 0
done
```

If you are extracting pitch periods automatically, it is worth taking more care to check the signal. We have found that recording consistency and bad pitch extraction the two most common causes of poor quality synthesis.

See the Section called *Extracting pitchmarks from waveforms* in Chapter 4 for a more detailed discussion on how to do this.

Building LPC parameters

Currently the only publically distributed signal processing method in Festival is residual excited LPC. To use this, you must extract LPC parameters and LPC residual files for each file in the diphone database. Ideally, the LPC analysis should be done pitch-synchronously, thus requiring that pitch marks are created before the LPC analysis takes place.

A script suitable for generating the LPC coefficients and residuals is given in `festvox/src/general/make_lpc` and is repeated here.

```
for i in $*
do
  fname=‘basename $i .wav‘
  echo $i

  # Potential normalise the power
  #$ESTDIR/bin/ch_wave -scaleN 0.5 $i -o /tmp/tmp$$.wav
  # resampling can be done now too
  #$ESTDIR/bin/ch_wave -F 11025 $i -o /tmp/tmp$$.wav
  # Or use as is
  cp -p $i /tmp/tmp$$.wav
  $ESTDIR/bin/sig2fv /tmp/tmp$$.wav -o lpc/$fname.lpc \
         -otype est -lpc_order 16 -coefs "lpc" \
         -pm pm/$fname.pm -preemph 0.95 -factor 3 \
         -window_type hamming
  $ESTDIR/bin/sigfilter /tmp/tmp$$.wav -o lpc/$fname.res \
          -otype nist -lpcfilter lpc/$fname.lpc -inv_filter
  rm /tmp/tmp$$.wav
done
```

Note the (optional) use of `ch_wave` to attempt to normalize the power in the wave to a percentage of its maximum. This is a very crude method for making the waveforms have a reasonably equivalent power. Wildly different power fluctuations in power between segments is likely to be noticed when they are joined. Differing power in the nonsense words may occur if not enough care has been taking in the recording. Either the settings on the recording equipment have been changed (bad) or the speaker has changed their vocal effort (worse). It is important that this should be avoided as the above normalization does not make the problem of different power go away it only makes the problem slightly less bad.

A more elaborate power normaliziation has been successful, but it is a little harder, though it was definitely successful for the KED US American voice that had major power fluctuations over different recording sesssions. The idea is to find the power during vowels in each nonsense word, then find the mean power for each vowel overall files. Then, for each file, find the average factor difference for each actual vowel with the mean for that vowel and scale the waveform according to that value. We now provided a basic script which does this

```
bin/find_powerfacts lab/*.lab
```

This script creates (among others) `etc/powfacts` which if it exists, is used to normalize the power of each waveform file during the making of the LPC coefficients.

We generate a set of `ch_wave` commands that extract the parts of the wave from that are vowels (using `-start` and `-end` options, make the output be in ascii `-otype raw -ostype ascii` and use a simple script to calculate the RMS power. We then calculate the mean power for each vowel with another awk script using the result as a table, then finally we process the fileid, actual vowel power information to generate a power factor to by averaging the ration of each vowel's actual power to the mean power for that vowel. You may wish to still modify the power further after this if it is too low or high.

Note that power normalization is intended to remove artifacts caused by different recording environment, i.e. the person moved from the microphone, the levels were changed etc. they should not modify the intrinsic power differences in the phones themselves. The above techniques try to preserve the intrinsic power, which is why we take the average over all vowels in a nonsense word, though you should listen to the results and make the ultimate decision yourself.

If all has been recorded properly, of course, individual power modification should be unnecessary. Once again, we can't stress enough how important it is to have good and consistent recording conditions, so as to avoid steps like this.

If you want to generate a database using a different sampling rate than the recordings were made with, this is the time to resample. For example an 8KHz or 11.025KHz will be smaller than a 16KHz database. If the eventual voice is to be played over the telephone, for example, there is little point in generating anything but 8Khz. Also it will be faster to synthesize 8Khz utterances than 16Khz ones.

The number of LPC coefficients used to represent each pitch period can be changed depending on sample rate you choose. Hearsay, reasonable experience, and perhaps some theoretical underpining, suggests the following formula for calculating the order

```
(sample_rate/1000)+2
```

But that should only be taken as a rough guide though a larger sample rate deserves a greater number of coeeficients.

Defining a diphone voice

The easiest way to define a voice is to start from the skeleton scheme files distributed. For English voices see Chapter 20, and for non-English voices see Chapter 19 for detailed walkthroughs.

Although in many cases you'll want to modify these files (sometimes quite substantially), the basic skeleton files will give you a good grounding, and they follow some basic conventions of voice files that will make it easier to integrate your voice into the Festival system.

Checking and correcting diphones

This probably sounds like we're repeating ourselves here, and we are, because it's quite important for the overall quality of the voice: once you have the basic diphone database working it is worthwhile systematically testing it as it is common to have mistakes. These may be mislabeling, and mispronunciation for the phones themselves. Two possible strategies are possible for testing both of which have their advantages. This first is a simple exhaustive synthesis of all diphones. Ideally, the diphone prompts are exactly the set of utterances that test each and every diphone. using the `SayPhones` function you can synthesize and listen to each prompt. Actually, for a first pass, it may even be useful to synthesize each nonsense word without listening as some of the problems missing files, missing diphones, badly extracted pitchmarks will show up without you having to listen to at all.

When a problem occurs, trace back why, check the entry in the diphone index, then check the label for the nonsense word, then check how that label matches the actually waveform file itself (display the waveform with the label file and spectrogram to see if the label is correct).

Listing all the problems that could occur is impossible. What you need to do is break down the problem and find out where it might be occurring. If you just get apparent garbage being synthesized, take a look at the synthesized waveform

```
(set! utt1 (SayPhones '(pau hh ah l ow pau)))
(utt.save.wave utt1 "hello.wav")
```

Is it *garbage*, can you recognized any part of it? It could be a byte swap problem or a format problem for your files. Can your nonsense word file be played and displayed as is? Can your LPC residual files be played and displayed. Residual files should look like very low powered waveform files and sound very buzzy when played but almost recognizable if you know what is being said (sort of like Kenny from South Park).

If you can recognize some of what is being said but it is fairly uniformly garbled it is possible your pitchmarks are not being aligned properly. Use some display mechanism to see where the pitchmarks are. These should be aligned (during voiced speech) with the peaks in the signal.

If all is well except for some parts of the signal are bad or overflowed, then check the diphone where the errors occur.

There are a number of solutions to problems that may save you some time, for the most part they should be considered cheating, but they may save having to re-record, which is something that you will probably want to avoid if at all possible.

Note that some phones are very similar, particular the left half side of most stops are indistinguishable, as the consist of mostly silence. Thus if you find you didn't get a good `SOMETHING-p` diphone you can easily make it use the `SOMETHING-b` diphone instead. You can do this by hand editing the diphone index file accordingly.

The linguists among you may not find that acceptable, but you can go further, the burst part of /p/ and /b/ isn't that different when it comes down to it and if is it just one or two diphones you can simply map those too. Considering problems are often in one or two badly articulated phones replace a /p/ with a /b/ (or similar) in one or two diphones may not be that bad.

Once, however, the problems become systematic over a number of phones re-recording them should be considered. Though remember if you do have to re-record you want to have as similar an environment as possible which is not always easy. Eventually you may need to re-record the whole database again.

Recording diphone databases is not an exact science, although we have a fair amount of experience in recording these databases, they never completely go as planned. Some apparently minor problem often occurs, noise on the channel, slightly different power over two sessions. Even when everything seems the same and we can't identify any difference between two recording environments we have found that some voices are better than others for building diphone databases. We can't immediately say why, we discussed some of these issues above in selecting a speaker but there is still some other parameters which we can't identify so don't be disheartened when you database isn't as good as you hoped, ours sometimes fail too.

Diphone check list

The section contains a quick check list of the processes required to constructing a working diphone database. Each part is discussed in detail above.

- Choose phoneset: Find an appropriate phoneset for the language, if possible using an existing standard. If you already have a good lexicon in the desired language, we recommend that you use that phone set.
- Construct diphone list: Construct the diphone list with appropriate carrier words. Either using an existing list or generating one from the examples. Consider what allophones, consonant clusters, etc., you also wish to record.
- Synthesize prompts: Synthesize prompts from an existing voice, if possible. Even when a few phones are missing from that voice it can still be useful to have the speaker listen to prompts as it keeps then focussed on minimal prosody and normalized vocal effort as well as reminding them what they need to say.
- Record words: Record the words in the best possible conditions you can. Bad recordings can never be corrected later. Ideally, you would use an anechoic chamber with voice from close talking mike and larynograph channels.
- Hand label/align phones: If you used prompts you can probably use the provided aligner to get a reasonable first pass at the phone labels. Alternatively, find a different aligner, or do it by hand.

- Extract pitchmarks: Extract the pitchmarks from the recorded signal, either from the EGG signal, or by the more complicated approach of extracting them from the speech signal itself.
- Build parameter files: If you don't have PSOLA, extract the LPC parameters and residuals from the speech signal, with power normalization if you feel its necessary.
- Build database itself: Build the diphone index, correcting any obvious labeling errors then test the database itself. Running significant tests to correct any further labeling errors.
- Test and check database: Systematically check the database by synthesizing the prompts again and synthesizing general text.

Chapter 12. Unit selection databases

This chapter discusses some of the options for building waveform synthesizers using unit selection techniques in Festival. This is still very much an on-going research question and we are still adding new techniques as well as improving existing ones often so the techniques described here are not as mature as the techniques as described in previous diphone chapter.

By "unit selection" we actually mean the selection of some unit of speech which may be anything from whole phrase down to diphone (or even smaller). Technically diphone selection is a simple case of this. However typically what we mean is unlike diphone selection, in unit selection there is more than one example of the unit and some mechanism is used to select between them at run-time.

ATR's CHATR [hunt96] system and earlier work at that lab [nuutalk92] is an excellent example of one particular method for selecting between multiple examples of a phone within a database. For a discussion of why a more generalized inventory of units is desired see [campbell96] though we will reiterate some of the points here. With diphones a fixed view of the possible space of speech units has been made which we all know is not ideal. There are articulatory effects which go over more than one phone, e.g. /s/ can take on artifacts of the roundness of the following vowel even over an intermediate stop, e.g. "*spout*" vs "*spit*". But its not just obvious segmental effects that cause variation in pronunciation, syllable position, word/phrase initial and final position have typically a different level of articulation from segments taken from word internal position. Stressing and accents also cause differences. Rather than try to explicitly list the desired inventory of all these phenomena and then have to record all of them a potential alternative is to take a natural distribution of speech and (semi-)automatically find the distinctions that actually exist rather predefining them.

The theory is obvious but the design of such systems and finding the appropriate selection criteria, weighting the costs of relative candidates is a non-trivial problem. However techniques like this often produce very high quality, very natural sounding synthesis. However they also can produce some very bad synthesis too, when the database has unexpected holes and/or the selection costs fail.

Two forms of unit selection will discussed here, not because we feel they are the best but simply because they are the ones actually implemented by us and hence can be distributed. These should still be considered research systems. Unless you are specifically interested or have the expertise in developing new selection techniques it is not recommended that you try these, if you need a working voice within a month and can't afford to miss that deadline then the diphone option is safe, well tried and stable. In you need higher quality and know something about what you need to say, then we recommend the limited domain techniques discussed in the following chapter. The limited domain synthesis offers the high quality of unit selection but avoids much (all ?) of the bad selections.

Cluster unit selection

This is a reimplementation of the techniques as described in [black97c]. The idea is to take a database of general speech and try to cluster each phone type into groups of acoustically similar units based on the (non-acoustic) information available at synthesis time, such as phonetic context, prosodic features (F0 and duration) and higher level features such as stressing, word position, and accents. The actually features used may easily be changed and experimented with as can the definition of the definition of acoustic distance between the units in a cluster.

In some sense this work builds on the results of both the CHATR selection algorithm [hunt96] and the work of [donovan95], but differs in some important and significant ways. Specifically in contrast to [hunt96] this cluster algorithm pre-builds CART trees to select the appropriate cluster of candidate phones thus avoiding the computation-

ally expensive function of calculating target costs (through linear regression) at selection time. Secondly because the clusters are built directly from the acoustic scores and target features, a target estimation function isn't required removing the need to calculate weights for each feature. This cluster method differs from the clustering method in [donovan95] in that it can use more generalized features in clustering and uses a different acoustic cost function (Donovan uses HMMs), also his work is based on sub-phonetic units (HMM states). Also Donovan selects one candidate while here we select a group of candidates and finds the best overall selection by finding the best path through each set of candidates for each target phone, in a manner similar to [hunt96] and [iwahashi93] before.

The basic processes involved in building a waveform synthesizer for the clustering algorithm are as follows. A high level walkthrough of the scripts to run is given after these lower level details.

- Collect the database of general speech.
- Building utterance structures for your database using the techniques discussed in the Section called *Utterance building* in Chapter 3.
- Building coefficients for acoustic distances, typically some form of cepstrum plus F0, or some pitch synchronous analysis (e.g. LPC).
- Build distances tables, precalculating the acoustic distance between each unit of the same phone type.
- Dump selection features (phone context, prosodic, positional and whatever) for each unit type.
- Build cluster trees using `wagon` with the features and acoustic distances dumped by the previous two stages
- Building the voice description itself

Choosing the right unit type

before you start you must make a decision about what unit type you are going to use. Note there are two dimensions here. First is *size*, such as phone, diphone, demi-syllable. The second *type* itself which may be simple phone, phone plus stress, phone plus word etc. The code here and the related files basically assume unit *size* is *phone*. However because you may also include a percentage of the previous unit in the acoustic distance measure this unit size is more effectively phone plus previous phone, thus it is somewhat diphone like. The cluster method has actual restrictions on the unit size, it simply clusters the given acoustic units with the given feature, but the basic synthesis code is currently assuming phone sized units.

The second dimension, type, is very open and we expect that controlling this will be a good method to attained high quality general unit selection synthesis. The parameter `clunit_name_feat` may be used define the unit type. The simplest conceptual example is the one used in the limited domain synthesis. There we distinguish each phone with the word it comes from, thus a *d* from the word *limited* is distinct from the *d* in the word *domain*. Such distinctions can hard partition up the space of phones into types that can be more manageable.

The decision of how to carve up that space depends largely on the intended use of the database. The more distinctions you make less you depend on the clustering acoustic distance, but the more you depend on your labels (and the speech) being (absolutely) correct. The mechanism to define the unit type is through a (typically) user defined feature function. In the given setup scripts this feature function will be called `lisp_INST_LANG_NAME::clunit_name`. Thus the voice simply defines the function `INST_LANG_NAME::clunit_name` to return the unit type for the given segment. If you wanted to make a diphone unit selection voice this function could simply be

```
(define (INST_LANG_NAME::clunit_name i)
  (string_append
   (item.name i)
   "_"
   (item.feat i "p.name")))
```

This the unittype would be the phone plus its previous phone. Note that the first part of a unit name is assumed to be the phone name in various parts of the code thus although you make think it would be neater to return `previousphone_phone` that would mess up some other parts of the code.

In the limited domain case the word is attached to the phone. You can also consider some demi-syllable information or more to differentiate between different instances of the same phone.

The important thing to remember is that at synthesis time the same function is called to identify the unittype which is used to select the appropriate cluster tree to select from. Thus you need to ensure that if you use say diphones that the your database really does not have *all* diphones in it.

Collecting databases for unit selection

Unlike diphone database which are carefully constructed to ensure specific coverage, one of the advantages of unit selection is that a much more general database is desired. However, although voices may be built from existing data not specifically gathered for synthesis there are still factors about the data that will help make better synthesis.

Like diphone databases the more cleanly and carefully the speech is recorded the better the synthesized voice will be. As we are going to be selecting units from different parts of the database the more similar the recordings are, the less likely bad joins will occur. However unlike diphones database, prosodic variation is probably a good thing, as it is those variations that can make synthesis from unit selection sound more natural. Good phonetic coverage is also useful, at least phone coverage if not complete diphone coverage. Also synthesis using these techniques seems to retain aspects of the original database. If the database is broadcast news stories, the synthesis from it will typically sound like read news stories (or more importantly will sound best when it is reading news stories).

Although it is too early to make definitive statements about what size and type of data is best for unit selection we do have some rough guides. A Timit like database of 460 phonetically balanced sentences (around 14,000 phones) is not an unreasonable first choice. If the text has not been specifically selected for phonetic coverage a larger database is probably required, for example the Boston University Radio News Corpus speaker `f2b` [ostendorf95] has been used relatively successfully. Of course all this depends on what use you wish to make of the synthesizer, if its to be used in more restrictive environments (as is often the case) tailoring the database for the task is a very good idea. If you are going to be reading a lot of telephone numbers, having a significant number of examples of read numbers will make synthesis of numbers sound much better (see the following chapter on making such design more explicit).

The database used as an example here is a TIMIT 460 sentence database read by an American male speaker.

Again the notes about recording the database apply, though it will sometimes be the case that the database is already recorded and beyond your control, in that case you will always have something legitimate to blame for poor quality synthesis.

Preliminaries

Throughout our discussion we will assume the following database layout. It is highly recommended that you follow this format otherwise scripts, and examples will fail. There are many ways to organize databases and many of such choices are arbitrary, here is our "arbitrary" layout.

The basic database directory should contain the following directories

`bin/`

Any database specific scripts for processing. Typically this first contains a copy of standard scripts that are then customized when necessary to the particular database

`wav/`

The waveform files. These should be headered, one utterances per file with a standard name convention. They should have the extension `.wav` and the fileid consistent with all other files through the database (labels, utterances, pitch marks etc).

`lab/`

The segmental labels. This is usually the master label files, these may contain more information that the labels used by festival which will be in `festival/relations/Segment/`.

`lar/`

The EGG files (larynograph files) if collected.

`pm/`

Pitchmark files as generated from the lar files or from the signal directly.

`festival/`

Festival specific label files.

`festival/relations/`

The processed labeled files for building Festival utterances, held in directories whose name reflects the relation they represent: `Segment/`, `Word/`, `Syllable/` etc.

`festival/utts/`

The utterances files as generated from the `festival/relations/` label files.

Other directories will be created for various processing reasons.

Building utterance structures for unit selection

In order to make access well defined you need to construct Festival utterance structures for each of the utterances in your database. This (in is basic form) requires labels for: segments, syllables, words, phrases, F0 Targets, and intonation events. Ideally these should all be carefully hand labeled but in most cases that's impractical. There are ways to automatically obtain most of these labels but you should be aware of the inherit errors in the labeling system you use (including labeling systems that involve human labelers). Note that when a unit selection method is to be used that fundamentally uses segment boundaries its quality is going to be ultimately determined by the quality of the segmental labels in the databases.

For the unit selection algorithm described below the segmental labels should be using the same phoneset as used in the actual synthesis voice. However a more detailed phonetic labeling may be more useful (e.g. marking closures in stops) mapping that information back to the phone labels before actual use. Autoaligned databases typically aren't accurate enough for use in unit selection. Most autoaligners are built using speech recognition technology where actual phone boundaries are not the primary measure of success. General speech recognition systems primarily measure words correct (or more usefully semantically correct) and do not require phone boundaries to be accurate. If the database is to be used for unit selection it is very important that the phone boundaries are accurate. Having said this though, we have successfully used the aligner described in the diphone chapter above to label general utterance where we knew which phone string we were looking for, using such an aligner may be a useful first pass, but the result should always be checked by hand.

It has been suggested that aligning techniques and unit selection training techniques can be used to judge the accuracy of the labels and basically exclude any segments that appear to fall outside the typical range for the segment type. Thus it, is believed that unit selection algorithms should be able to deal with a certain amount of noise in the labeling. This is the desire for researchers in the field, but we are some way from that and the easiest way at present to improve the quality of unit selection algorithms at present is to ensure that segmental labeling is as accurate as possible. Once we have a better handle on selection techniques themselves it will then be possible to start experimenting with noisy labeling.

However it should be added that this unit selection technique (and many others) support what is termed "optimal coupling" [conkie96] where the acoustically most appropriate join point is found automatically at run time when two units are selected for concatenation. This technique is inherently robust to at least a few tens of millisecond boundary labeling errors.

For the cluster method defined here it is best to construct more than simply segments, durations and an F0 target. A whole syllabic structure plus word boundaries, intonation events and phrasing allow a much richer set of features to be used for clusters. See the Section called *Utterance building* in Chapter 3 for a more general discussion of how to build utterance structures for a database.

Making cepstrum parameter files

In order to cluster similar units in a database we build an acoustic representation of them. This is is also still a research issue but in the example here we will use Mel cepstrum. Interestingly we do not generate these at fixed intervals, but at pitch marks. Thus have a parametric spectral representation of each pitch period. We have found this a better method, though it does require that pitchmarks are reasonably identified.

Here is an example script which will generate these parameters for a database, it is included in `festvox/src/unitsel/make_mcep`.

```
for i in $*
do
  fname=‘basename $i .wav‘
  echo $fname MCEP
  $SIG2FV $SIG2FVPARAMS -otype est_binary $i -o mcep/$fname.mcep -pm pm/$fname.pm -
window_type hamming
done
```

The above builds coefficients at fixed frames. We have also experimented with building parameters pitch synchronously and have found a slight improvement in the

usefulness of the measure based on this. We do not pretend that this part is particularly neat in the system but it does work. When pitch synchronous parameters are build the clunits module will automatically put the local F0 value in coefficient 0 at load time. This happens to be appropriate from LPC coefficients. The script in `festvox/src/general/make_lpc` can be used to generate the parameters, assuming you have already generated pitch marks.

Note the secondary advantage of using LPC coefficients is that they are required any way for LPC resynthesis thus this allows less information about the database to be required at run time. We have not yet tried pitch synchronous MEL frequency cepstrum coefficients but that should be tried. Also a more general duration/number of pitch periods match algorithm is worth defining.

Building the clusters

Cluster building is mostly automatic. Of course you need the `clunits` modules compiled into your version of Festival. Version 1.3.1 or later is required, the version of `clunits` in 1.3.0 is buggy and incomplete and will not work. To compile in `clunits`, add

```
ALSO_INCLUDE += clunits
```

to the end of your `festival/config/config` file, nad recompile. To check if an installation already has support for `clunits` check the value of the variable `*modules*`.

The file `festvox/src/unitsel/build_clunits.scm` contains the basic parameters to build a cluster model for a databases that has utterance structures and acoustic parameters. The function `build_clunits` will build the distance tables, dump the features and build the cluster trees. There are many parameters are set for the particular database (and instance of cluster building) through the Lisp variable `clunits_params`. An reasonable set of defaults is given in that file, and reasonable run-time parameters will be copied into `festvox/INST_LANG_VOX_clunits.scm` when a new voice is setup.

The function `build_clunits` runs through all the steps but in order to better explain what is going on, we will go through each step and at that time explain which parameters affect the substep.

The first stage is to load in all the utterances in the database, sort them into segment type and name them with individual names (as `TYPE_NUM`. This first stage is required for all other stages so that if you are not running `build_clunits` you still need to run this stage first. This is done by the calls

```
(format t "Loading utterances and sorting types\n")
(set! utterances (acost:db_utts_load dt_params))
(set! unittypes (acost:find_same_types utterances))
(acost:name_units unittypes)
```

Though the function `build_clunits_init` will do the same thing.

This uses the following parameters

`name STRING`

A name for this database.

`db_dir FILENAME`

This pathname of the database, typically `.` as in the current directory.

`utts_dir FILENAME`

The directory contain the utterances.

`utts_ext FILENAME`

The file extention for the utterance files

`files`

The list of file ids in the database.

For example for the KED example these parameters are

```
(name 'ked_timit)
(db_dir "/usr/awb/data/timit/ked/")
(utts_dir "festival/utts/")
(utts_ext ".utt")
(files ("kdt_001" "kdt_002" "kdt_003" ... ))
```

In the examples below the list of fileids is extracted from the given prompt file at call time.

The next stage is to load the acoustic parameters and build the distance tables. The acoustic distance between each segment of the same type is calculated and saved in the distance table. Precalculating this saves a lot of time as the cluster will require this number many times.

This is done by the following two function calls

```
(format t "Loading coefficients\n")
(acost:utts_load_coeffs utterances)
(format t "Building distance tables\n")
(acost:build_disttabs unittypes clunits_params)
```

The following parameters influence the behaviour.

`coeffs_dir FILENAME`

The directory (from db_dir) that contains the acoustic coefficients as generated by the script `make_mcep`.

`coeffs_ext FILENAME`

The file extention for the coefficient files

`get_std_per_unit`

Takes the value `t` or `nil`. If `t` the parameters for the type of segment are normalized by finding the means and standard deviations for the class are used. Thus a mean mahalanobis euclidean distance is found between units rather than simply a euclidean distance. The recommended value is `t`.

`ac_left_context FLOAT`

The amount of the previous unit to be included in the the distance. 1.0 means all, 0.0 means none. This parameter may be used to make the acoustic distance sensitive to the previous acoustic context. The recommended value is `0.8`.

`dur_pen_weight FLOAT`

The penalty factor for duration mismatch between units.

`f0_pen_weight FLOAT`

The penalty factor for F0 mismatch between units.

`ac_weights (FLOAT FLOAT ...)`

The weights for each parameter in the coefficeint files used while finding the acoustic distance between segments. There must be the same number of weights as there are parameters in the coefficient files. The first parameter is (in normal operations) F0. Its is common to give proportionally more weight to F0 that to each individual other parameter. The remaining parameters are typically MFCCs (and possibly delta MFCCs). Finding the right parameters and weightings is one the key goals in unit selection synthesis so its not easy to give concrete recommendations. The following aren't bad, but there may be better ones too though we suspect that real human listening tests are probably the best way to find better values.

An example is

```
(coeffs_dir "mcep/")
(coeffs_ext ".mcep")
(dur_pen_weight 0.1)
(get_stds_per_unit t)
(ac_left_context 0.8)
(ac_weights
  (0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5))
```

The next stage is to dump the features that will be used to index the clusters. Remember the clusters are defined with respect to the acoustic distance between each unit in the cluster, but they are indexed by these features. These features are those which will be available at text-to-speech time when no acoustic information is available. Thus they include things like phonetic and prosodic context rather than spectral information. The name features may (and probably should) be over general allowing the decision tree building program `wagon` to decide which of theses feature actual does have an acoustic distinction in the units.

The function to dump the features is

```
(format t "Dumping features for clustering\n")
(acost:dump_features unittypes utterances clunits_params)
```

The parameters which affect this function are

`fests_dir FILENAME`

The directory when the features will be saved (by segment type).

`feats LIST`

The list of features to be dumped. These are standard festival feature names with respect to the Segment relation.

For our KED example these values are

```
(feats_dir "festival/feats/")
(feats
    (occurid
     p.name p.ph_vc p.ph_ctype
        p.ph_vheight p.ph_vlng
        p.ph_vfront  p.ph_vrnd
        p.ph_cplace  p.ph_cvox
     n.name n.ph_vc n.ph_ctype
        n.ph_vheight n.ph_vlng
        n.ph_vfront  n.ph_vrnd
        n.ph_cplace  n.ph_cvox
     segment_duration
```

```
seg_pitch p.seg_pitch n.seg_pitch
R:SylStructure.parent.stress
seg_onsetcoda n.seg_onsetcoda p.seg_onsetcoda
R:SylStructure.parent.accented
pos_in_syl
syl_initial
syl_final
R:SylStructure.parent.syl_break
R:SylStructure.parent.R:Syllable.p.syl_break
pp.name pp.ph_vc pp.ph_ctype
   pp.ph_vheight pp.ph_vlng
   pp.ph_vfront  pp.ph_vrnd
   pp.ph_cplace pp.ph_cvox))
```

Now that we have the acoustic distances and the feature descriptions of each unit the next stage is to find a relationship between those features and the acoustic distances. This we do using the CART tree builder `wagon`. It will find out questions about which features best minimize the acoustic distance between the units in that class. `wagon` has many options many of which are apposite to this task though it is interesting that this learning task is interestingly closed. That is we are trying to classify *all* the units in the database, there is no test set as such. However in synthesis there will be desired units whose feature vector didn't exist in the training set.

The clusters are built by the following function

```
(format t "Building cluster trees\n")
(acost:find_clusters (mapcar car unittypes) clunits_params)
```

The parameters that affect the tree building process are

`tree_dir FILENAME`

the directory where the decision tree for each segment type will be saved

`wagon_field_desc LIST`

A filename of a wagon field descriptor file. This is a standard field description (field name plus field type) that is require for wagon. An example is given in `festival/clunits/all.desc` which should be sufficient for the default feature list, though if you change the feature list (or the values those features can take you may need to change this file.

`wagon_progname FILENAME`

The pathname for the `wagon` CART building program. This is a string and may also include any extra parameters you wish to give to `wagon`.

`wagon_cluster_size INT`

The minimum cluster size (the wagon `-stop` value).

`prune_reduce INT`

This number of elements in each cluster to remove in pruning. This removes the units in the cluster that are furthest from the center. This is down within the wagon training.

`cluster_prune_limit INT`

This is a post wagon build operation on the generated trees (and perhaps a more reliably method of pruning). This defines the maximum number of units that will be in a cluster at a tree leaf. The wagon cluster size the minimum size. This is

usefully when there are some large numbers of some particular unit type which cannot be differentiated. Format example silence segments without context of nothing other silence. Another usage of this is to cause only the center example units to be used. We have used this in building diphones databases from general databases but making the selection features only include phonetic context features and then restrict the number of diphones we take by making this number 5 or so.

`unittype_prune_threshold INT`

When making complex unit types this defines the minimal number of units of that type required before building a tree. When doing cascaded unit selection synthesizers its often not worth excluding large stages if there is say only one example of a particular demi-syllable.

Note that as the distance tables can be large there is an alternative function that does both the distance table and clustering in one, deleting the distance table immediately after use, thus you only need enough disk space for the largest number of phones in any type. To do this

```
(acost:disttabs_and_clusters unittypes clunits_params)
```

Removing the calls to `acost:build_disttabs` and `acost:find_clusters`.

In our KED example these have the values

```
(trees_dir "festival/trees/")
(wagon_field_desc "festival/clunits/all.desc")
(wagon_progname "/usr/awb/projects/speech_tools/bin/wagon")
(wagon_cluster_size 10)
(prune_reduce 0)
```

The final stage in building a cluster model is collect the generated trees into a single file and dumping the unit catalogue, i.e. the list of unit names and their files and position in them. This is done by the lisp function

```
(acost:collect_trees (mapcar car unittypes) clunits_params)
(format t "Saving unit catalogue\n")
(acost:save_catalogue utterances clunits_params)
```

The only parameter that affect this is

`catalogue_dir FILENAME`

the directory where the catalogue will be save (the `name` parameter is used to name the file).

Be default this is

```
(catalogue_dir "festival/clunits/")
```

There are a number of parameters that are specified with a cluster voice. These are related to the run time aspects of the cluster model. These are

`join_weights FLOATLIST`

This are a set of weights, in the same format as `ac_weights` that are used in optimal coupling to find the best join point between two candidate units. This is different from `ac_weights` as it is likely different values are desired, particularly increasing the F0 value (column 0).

`continuity_weight FLOAT`

The factor to multiply the join cost over the target cost. This is probably not very relevant given the the target cost is merely the position from the cluster center.

`log_scores 1`

If specified the joins scores are converted to logs. For databases that have a tendency to contain non-optimal joins (probably any non-limited domain databases), this may be useful to stop failed synthesis of longer sentences. The problem is that the sum of very large number can lead to overflow. This helps reduce this. You could alternatively change the continuity_weight to a number less that 1 which would also partially help. However such overflows are often a pointer to some other problem (poor distribution of phones in the db), so this is probably just a hack.

`optimal_coupling INT`

If `1` this uses optimal coupling and searches the cepstrum vectors at each join point to find the best possible join point. This is computationally expensive (as well as having to load in lots of cepstrum files), but does give better results. If the value is `2` this only checks the coupling distance at the given boundary (and doesn't move it), this is often adequate in good databases (e.g. limited domain), and is certainly faster.

`extend_selections INT`

If `1` then the selected cluster will be extended to include any unit from the cluster of the previous segments candidate units that has correct phone type (and isn't already included in the current cluster). This is experimental but has shown its worth and hence is recommended. This means that instead of selecting just units selection is effectively selecting the beginnings of multiple segment units. This option encourages far longer units.

`pm_coeffs_dir FILENAME`

The directory (from `db_dir` where the pitchmarks are

`pm_coeffs_ext FILENAME`

The file extension for the pitchmark files.

`sig_dir FILENAME`

Directory containing waveforms of the units (or residuals if Residual LPC is being used, PCM waveforms is PSOLA is being used)

`sig_ext FILENAME`

File extension for waveforms/residuals

`join_method METHOD`

Specify the method used for joining the selected units. Currently it supports `simple`, a very naive joining mechanism, and `windowed`, where the ends of the units are windowed using a hamming window then overlapped (no prosodic modification takes place though). The other two possible values for this feature are `none` which does nothing, and `modified_lpc` which uses the standard UniSyn module to modify the selected units to match the targets.

`clunits_debug 1/2`

With a value of `1` some debugging information is printed during synthesis, particularly how many candidate phones are available at each stage (and any extended ones). Also where each phone is coming from is printed.

With a value of `2` more debugging information is given include the above plus joining costs (which are very readable by humans).

Building a Unit Selection Cluster Voice

The previous section gives the low level details ofin the building of a cluster unit selection voice. This section gives a higher level view with explict command that you should run. The steps involved in building a unit selection voices are basically the same as that for building a limited domain voice (Chapter 5). Though in for general voices, in constrast to ldom voice, it is much more important to get all parts correct, from pitchmarks to labeling.

The following tasks are required:

- Read and understand all the issues regarding the following steps
- Design the prompts
- Record the prompts
- Autolabel the prompts
- Build utterance structures for recorded utterances
- Extract pitchmark and build LPC coefficients
- Building a clunit based synthesizer from the utterances
- Testing and tuning

The following are the commands that you must type (assuming all the other hardwork has been done beforehand. It is assume that the environment variables `FESTVOXDIR` and `ESTDIR` have been set to point to their respective directories. For example as

```
export FESTVOXDIR=/home/awb/projects/festvox
export ESTDIR=/home/awb/projects/1.4.3/speech_tools
```

Next you must select a name for the voice, by convention we use three part names consisting of a institution name, a language, and a speaker. Make a directory of that name and change directory into it

```
mkdir cmu_us_awb
cd cmu_us_awb
```

There is a basic set up script that will construct the directory structure and copy in the template files for voice building. If a fourth argument is given, it can be name one of the standard prompts list.

For example the simplest is `uniphone`. This contains three sentences which contain each of the US English phonemes once (if spoken appropriately). This prompt set is hopelessly minimal for any high quality synthesis but allows us to illustrate the process and allow you to build a voice quickly.

```
$FESTVOXDIR/src/unitsel/setup_clunits cmu us awb uniphone
```

Alternatively you can copy in a prompt list into the etc directory. The format of these should be in the standard "data" format as in

```
( uniph_0001 "a whole joy was reaping." )
( uniph_0002 "but they've gone south." )
( uniph_0003 "you should fetch azure mike." )
```

Note the spaces after the initial left parenthesis are significant, and double quotes and backslashes within the quote part must be escaped (with backslash) as is common in Perl or Festival itself.

The next stage is to generate waveforms to act as prompts, or timing cues even if the prompts are not actually played. The files are also used in aligning the spoken data.

```
festival -b festvox/build_clunits.scm '(build_prompts "etc/uniphone.data")'
```

Use whatever prompt file you are intending to use. Note that you may want to add lexical entries to `festvox/WHATEVER_lexicon.scm` and other text analysis things as desired. The purpose is that the prompt files match the phonemes that the voice talent will actually say.

You may now record, assuming you have prepared the recording studio, gotten written permission to record your speaker (and explained to them what the resulting voice might be used for), checked recording levels and sound levels and shield the electrical equipment as much as possible.

```
./bin/prompt_them etc/uniphone.data
```

After recording the recorded files should be in `wav/`. It is wise to check that the are actually there and sound like you expected. Getting the recording quality as high as possible is fundamental to the success of building a voice.

Now we must label the spoken prompts. We do this my matching the synthesized prompts with the spoken ones. As we know where the phonemes begin and end in the synthesized prompts we can then map that onto the spoken ones and find the phoneme segments. This technique works fairly well, but it is far from perfect and it is worthwhile to at least check the result, and most probably fix the result by hand.

```
./bin/make_labs prompt-wav/*.wav
```

Especially in the case of the uniphone synthesizer, where there is one and only one occurrence of each phone they all must be correct so its important to check the labels by hand. Note for large collections you may find the full Sphinx based labeling technique better the Section called *Labeling with Full Acoustic Models* in Chapter 14).

After labeling we can build the utterance structure using the prompt list and the now labeled phones and durations.

```
festival -b festvox/build_clunits.scm '(build_utts "etc/uniphone.data")'
```

The next stages are concerned with signal analysis, specifically pitch marking and cepstral parameter extraction. There are a number of methods for pitch mark extraction and a number of parameters within these files that may need tuning. Good pitch periods are important. See the Section called *Extracting pitchmarks from waveforms* in Chapter 4 . In its simplest case the follow may work

```
./bin/make_pm_wave wav/*.wav
```

The next stage it find the Mel Frequency Cepstral Coefficents. This is done pitch synchronously and hence depends on the pitch periods extracted above. These are used for clustering and for join measurements.

```
./bin/make_mcep wav/*.wav
```

Now we can do the main part of the build, building the cluster unit selection synthesizer. This consists of a number os stages all based on the controlling Festival script. The parameters of which are described above.

```
festival -b festvox/build_clunits.scm '(build_clunits "etc/uniphone.data")'
```

For large databases this can take some time to run as there is a squared aspect to this based on the number of instances of each unit type.

Diphones from general databases

As touched on above the choice of an inventory of units can be viewed as a line from a small inventory phones, to diphones, triphones to arbitrary units. Though the direction you come from influences the selection of the units from the database. CHATR [campbell96] lies firmly at the "arbitrary units" end of the spectrum. Although it can exclude bad units from its inventory it is very much "*everything minus some*" view of the world. Microsoft's Whistler [huang97] on the other hand, starts off with a general database base but selects typical units from it. Thus its inventory is substantially smaller than the full general database the units are extracted from. At the other end of the spectrum we have the fixed pre-specified inventory like diphone synthesis as has bee described in the previous chapter.

In this section we'll give some examples of moving along the line from the fixed prespecified inventory to the words the more general inventories but these techniques still have a strong component of prespecification.

Firstly lets us assume you have a general database that is labeled with utterances as described above. We can extract a standard diphone database from this general database, however unless the database was specifically designed, a general database is unlikely to have diphone coverage. Even when phonetically rich databases are used such as Timit there is likely to be very few vowel-vowel diphones as they are comparatively rare. But as these diphone are rare we may be able to do with out them and hence it is at least an interesting exercise to extract an as complete as possible diphone index from a general database.

The simplest method is to linearly search for all phone-phone pairs in the phone set through all utterances simply taking the first example. Some same code is given in `src/diphone/make_diphs_index.scm`. This basic idea is to load in all the utterances in a database, and index each segment by is phone name and succeeding phone name. Then various selection techniques can be use to select from the multiple candidates of each diphone (or you can split the indexing further). After selection a diphone index file can be saved.

The utterances to load are identified by a list of fileids. For example if the list of fileids (without parenthesis) is in the file `etc/fileids`, the following will builds a diphone index.

```
festival .../make_diphs_utts.scm
...
```

```
festival> (set! fileids (load "etc/fileids" t))
...
festival> (make_diphone_index fileids "dic/f2bdiph.est")
```

Note that as this diphone index will contain a number of holes you will need to either augment it with "*similar*" diphones or process your diphone selections through `UniSyn_module_hooks` as described in the previous chapter.

As you complicate the selection, and the number of diphones you used from the database you will need to complicate the names used to identify the diphones themselves. The convention of using underscores for syllable internal consonant clusters and dollars for syllable initial consonants can be followed, but you will need to go further if you wish to start introducing new feature such as phrase finality and stress. Eventually going to a generalized naming scheme (type and number) as used by the cluster selection technique described above, will prove worth while. Also using CART trees, through hand written and fully deterministic (one candidate at the leafs), will be a reasonable algorithm to select between hand stipulated alternatives with reasonable backoff strategies.

Another potential direction is to use the acoustic costs used in the clustering methods described in the previous section. These can be used to identify what the most typical unit in a cluster are (the mean distances from all other units are given in the leafs). Pruning these trees until the cluster only contain a single example should help to improve synthesis, in that variation in the feature in the "diphone" index will then be determined by the features specified in the cluster train algorithm. Of course though as you limit the number of distinct units types the more prosodic modification will be required by your signal processing algorithm, which requires that you have good pitch marks.

If you already have an existing database but don't wish to go to full unit selection, such techniques are probably quite feasible and worth further investigation.

Chapter 13. Statistical Parametric Synthesis

Building a CLUSTERGEN Statistical Parametric Synthesizer

This method, inspired the work of Keiichi Tokuda and NITECH's HMM Speech Synthesis Toolkit, is a method for building statistical parametric synthesizers from databases of natural speech. Although the result is still not as crisp as a well done unit selection voice, this method is much easier to get a nice clear synthetic voice that models the original speaker well.

Although this method is partially "tagged on to" the clunits method, it is actually quite independent. The tasks are as follows.

- Read and understand all the issues regarding the following steps
- Set up the directory structure
- Record or import the prompts and prompt list
- Label the data with the HMM-state sized segments
- Build utterance structures for recorded utterances
- Extract F0, voicing and mcep coefficients.
- Build a CLUSTERGEN voice
- Build an HMM-state duration model
- Testing

We assume you have read the rest of this chapter (though, in reality, we know you probably haven't), thus the descriptions here are quite minimal.

First make an empty directory and in it run the `setup_cg` setup command.

```
mkdir cmu_us_awb_arctic
cd cmu_us_awb_arctic
$FESTVOXDIR/src/clustergen/setup_cg cmu us awb_arctic
```

In you already have an existing voice running `setup_cg` will only copy in the necessary files for clustergen, however I recommend starting from scratch as I don't know when you created your previous voice and I'm not sure of its exact state.

Now you need to get your waveform files and prompt file. Put your waveform files in the `wav/` and your prompt file in `etc/txt.done.data`. Note you should probably use `bin/get_wavs` to copy the wavefiles so that they get power normalized and get changed to a reasonable format (16KHz, 16bit, RIFF format).

In you are going to record them in your current directory, you should call

```
./bin/do_build build_prompts
```

first to generate example waveforms, then use

```
./bin/prompt_them etc/txt.done.data 1
```

To prompt you and record the prompts. You *must* check that the recording actually works. It should generate recordings in the `wav/`. You can use `$ESTDIR/bin/na_play` to play the waveform files. `prompt_them` can be stopped with ctrl-c and restarted at the line number given as the second argument.

The next stage is to label the data. If you aren't *very* knowledgeable about labeling in clustergen, you should use the EHMM labeler. EHMM constructs the labels in the right format for segments and HMM states. and matches them properly with what

the synthesizer generates for the prompts. Using other labels is likely to cause more problems. Even if you already have other labels use EHMM first.

```
./bin/do_build build_prompts
./bin/do_build label
./bin/do_build build_utts
```

The EHMM labeler has been shown to be very reliable, and can nicely deal with silence insertion. It isn't very fast though and will take several hours. You can check the file `ehmm/mod/log100.txt` to see the Baum-Welch iterations, there will probably be 20-30. The ARCTIC a-set takes about 3-4 hours to label.

Parametric synthesis require a reversible parameterization, this set up here uses a form of mel cepstrum, the same version that is used by NITECH's basic HTS build. Parameter build is in two parts building the F0 and building the mceps themselves. Then these are combined into a single parameter file for each utterance in the database.

```
./bin/do_clustergen f0
./bin/do_clustergen mcep
./bin/do_clustergen voicing
./bin/do_clustergen combine_coeffs_v
```

The mcep part takes the longest. Note that the F0 part now tries to estimate the range of the F0 on the speaker and modifies parameters for the F0 extraction program. (The F0 params are saved in `etc/f0.params`.)

If you want to have a test set of utterances, you can separate out some of your prompt list. The test set should be put in the file `etc/txt.done.data.test` The follow commands will make a training and test set (every 10th prompt in the test set, the other 9 in the training set).

```
./bin/traintest etc/txt.done.data
cat etc/txt.done.data.train >etc/txt.done.data
```

The next stage is to generate is to build the parametric model. There parts are required for this. This first is very quick and simply puts the state (and phone) names into their respective files. It assumes a file `etc/statenames` which is generate by EHMM. The second stage build the parametric models itself. The last builds a duration model for the state names

```
./bin/do_clustergen generate_statenames
./bin/do_clustergen cluster
./bin/do_clustergen dur
```

The resulting voice should now work

```
festival festvox/cmu_us_awb_arctic_cg.scm
...
festival> (voice_cmu_us_awb_arctic_cg)
...
festival> (SayText "This is a little example.")
```

The voice can be packaged for distribution by the command

```
./bin/do_clustergen festvox_dist
```

This will generation `festvox_cmu_us_awb_arctic_cg.tar.gz` which will be quite small compared to a clunit voice made with the same databases. Because only the parameters are kept (in fact only means and standard deviations of clusters of of parameters) which do not include residual or excitation information the result is something orders of magnitude smaller that a full unit selection voices.

There two other options in the clustergen voice build. These involve modeling trajectories rather than individual vectors. They give objectively better results (though marginal subjectively better results for the voices we have tested). Instead of the line

```
./bin/do_clustergen cluster
```

You can run

```
./bin/do_clustergen trajectory
```

or the slightly better

```
./bin/do_clustergen trajectory_ola
```

These two options may run after the simple version of the voice.

You can test your voice with held out data, if you did this in the above step that created `etc/txt.done.data.test` You can run

```
$FESTVOXDIR/src/clustergen/cg_test resynth cgp
```

NOTE: This no longer works automatically, as you need static mceps and ccoefs for this to work. This will create parameter files (and waveform files) in `test/cgp`. The output of the `cg_test` is also four measures the mean difference for all features in the parameter vector, for F0 alone, for all but F0, and MCD (mel ceprstral distortion).

Chapter 14. Labeling Speech

In the early days of concatenative speech synthesis every recorded prompt had to be hand labeled. Although a significant task, very skilled and mind bogglingly tedious it was a feasible task to attempt when databases were relatively and the time to build a voice was measure in years. With the increase in size of database and the demand for much faster turnaround we have moved away from hand labeling to automatic labeling.

In this section we will only touch the the aspects of *what* we need labeled in recorded data but discuss what techniques are available for *how* to label it. As discussed before phonemes are a useful but incomplete inventory of units that should be identified but other aspects of lexical stress, prosody, allophonic variations etc are certainly worthy of consideration.

In labeling recorded prompts for synthesis we rely heavily on the work that has been done in the speech recognition community. For synthesis we do, however, have different goals. In ASR (automatic speech recognition) we are trying to find the most likely set of phones that are in a given acoustic observation. In synthesis labeling, however we know the sequence of phones spoken, assuming the voice talent spoke the prompt properly, and wish to find out where those phones are in the signal. We care, very deeply, about the boundaries of segments, while ASR can be achieve adequately performance by only concerning itself with the centers, and hence has rightly been optimized for that.

* *AWB: that point deserves more discussion, though maybe not here*

.

There are other distinctions from the ASR task, in synthesis labeled we are concerned with a singled speaker, that is, if the synthesizer is going to work well, very carefully performed and consistently recorded. This does make things easier for the labeling task. However in synthesize labeling we are also concerned about prosody, and spectral variation, much more than in ASR.

We discuss two specific techniques for labeling record prompts here, which each have their advantages and limitations. Procedures running these are discussed at the end of each section.

The first technique uses *dynamic time warping* alignment techniques to find the phone boundaries in a recorded prompt by align it against a synthesized utterance where the phone boundary are know. This is computationally easier than second technique and works well for small databases which do not have full phonetic coverage.

The second technique uses *Baum-Welch training* to build complete ASR acoustic models from the the database. This takes sometime, but if the database is phonetically balanced, as should be the case in databases designed for speech synthesis voices, can work well. Also this technique can work well on databases in languages that do not yet have a synthesizer, hence making the dynamic time warping technique hard without cross-language phone mapping techniques.

Labeling with Dynamic Time Warping

DTW (dynamic time warping) is a technique for aligning some new recording with some known one. This technique was used in early speech recognition systems which had limit vocabularies as it requires a acoustic signal for each word/phrase to be recognized. This technique is sometime still used in matching two audio signal in command and control situations, for example in some cell-phone for voice dialing.

What is important in DTW alignment is that it can deal with signals that have varying durations. The idea has been around for many years, though its application to labeling in synthesis is relative new. The work here is based on the detail published in [malfrere].

Comparing raw acoustic score is unlikely to given god results so comparisons are done in then spectral domain. Following ASR techniques we will use Mel Frequency Cepstral Coefficients to represent the signal, and also following ASR we will include delta MFCCs (the different between the current MFCC vector and the previous MFCC vector). However for the DTW algorithm the content of the vectors is somewhat irrelevant, and are merely treated as vectors.

The next stage is define a distance function between two vectors, conventionally we use Euclidean Distance defined as

root (sumof(i-n) (v0i - v1i)^2

Weights could be considered too.

The search itself is best picture as a large matrix. The algorithm then searches for the best path through the matrix. At each node it finds the distance between the two current vectors and sums it with the smallest of three potential previous states. That is one of i-1,j, i,j-1, or i-1,j-1. If two signals were identical the best path would be the diagonal through the matrix, if one part of the signal is shorter or longer than the corresponding one horizontal or vertical parts will have less cost.

matrix diagram (more than one)

* *AWB: describe the make_labs stuff and cross-language phone mapping*

Labeling with Full Acoustic Models

A second method for labeling is also available. Here we train full acoustic HMM models on the recorded data. We build a database specific speech recognition engine and use that engine to label the data. As this method can work from recorded prompts plus orthography (and a method to produce phone strings from that orthography), this works well when you have no synthesizer to bootstrap from. However such training requires that the database has a suitable number of examples of triphones in it. Here we have an advantage. As the requirements for a speech synthesis data, that it is has a good distribution of phonemes, is the same as that require for acoustic modeling, a good speech synthesis databases should produce a good acoustic model for labeling. Although there is no neatly defined definition of what "good" is, we can say that you probably need at least 400 utterances, and at least 15,000 segments. 400 sentences all starting with "The time is now, ..." probably wont do.

Other large database synthesis techniques use the same basic techniques to not just label the database but define the units to be selected. [Donovan95] and others label there data with an acoustic model build (with Baum-Welch training) and use the defined HMM states (typically 3-5 per phoneme) as the units for selecting. [Tokuda9?] actually use the state models themselves to generate the units, but again use the same basic techniques for labeling.

For training we use Carnegie Mellon University's SphinxTrain and Sphinx speech recognition system. There are other accessible training systems out there, HTK being the most famous, but SphinxTrain is the one we are most familiar with, and we have some control over its updates so can better ensure it remains appropriate for our synthesis labeling task. As voice building is complex, acoustic model building is similarly so. SphinxTrain has been reliably used to labeling hundreds of databases in many different languages but making it utterly robust against unseen data is very hard so although we have tried to minimize the chance of things going wrong (in non-obvious ways), we will not be surprised that when you try this processing on some new database there may be some problems.

SphinxTrain (and sphinx) have a number of restrictions which we need to keep in mind when labeling a set of prompts. These a re code limitations, and may be fixed

in future versions of SphinxTrain/Sphinx. For the most part the are not actually serious restrictions, just minor prompts that the setup scripts need to work around. The scripts cater for these limitations, and mostly will all go unseen by the user, unless of course something goes wrong.

Specifically, sphinx folds case on all phoneme names, so the scripts ensure that phone names are distinct irrespective of upper and lower case. This is done by prepending "CAP" in front of upper case phone names. Secondly there can only be up to 255 phones. This is likely only to be problem when SphinxTrain phones are made more elaborate than simple phones, so mostly wont be a problem. The third noted problem is limitation on the length and complexity of utterances. The transcript files has a line length limit as does the lexicon. For "nice" utterances this is never a problem but for some of our databases especially those with paragraph length utterances, the training and/or the labeling itself fails.

Sphinx2 is a real-time speech recognition system made available under a free software license. It is available from http://cmusphinx.org. The source is available from http://sourceforge.net/projects/cmusphinx/. For these tests we used version `sphinx2-0.4.tar.gz`. SphinxTrain is a set of programs and scripts that allow the building of acoustic models for Sphinx2 (and Sphinx3). You can download SphinxTrain from http://sourceforge.net/projects/cmusphinx/[3]. Note that Sphinx2 must be compiled and installed while SphinxTrain can run in place. On many systems steps like these should give you working versions.

```
tar zxvf sphinx2-0.4.tar.gz
mkdir sphinx2-runtime
export SPHINX2DIR=‘pwd‘/sphinx2-runtime
cd sphinx2 ./configure --prefix=$SPHINX2DIR
make
make install
cd ..
tar zxvf SphinxTrain-0.9.1-beta.tar.gz
cd SphinxTrain
./configure
make
export SPHINXTRAINDIR=‘pwd‘/SphinxTrain
```

Now that we have sphinx2 and SphinxTrain installed we can prepare our FestVox voice for training. Before starting the training process you must create utterance files for each of the prompts. This can be done with the conventional festival script.

```
festival -b festvox/build_clunits.scm ’(build_prompts "etc/txt.done.data")’
```

This generates label files in `prompt-lab/` and waveform files in `prompt-wav/` which technically are not needed for this labeling process. Utterances are saved in `prompt-utt/`. At first it was thought that the prompt file `etc/txt.done.data` would be sufficient but the synthesis process is likely to resolve pronunciations in context, though post-lexical rules etc, that would make naive conversion of the words in the prompt list to phone lists wrong in general so the transcription for SphinxTrain is generated from the utterances themselves which ensures that they resulting labels can be trivially mapped back after labeling. Thus the word names generate by in this process are somewhat arbitrary though often human readable. The word names are the word themselves plus a number (to ensure uniqueness in pronunciations). Only "nice" words are printed as is, i.e. those containing only alphabetic characters, others are mapped to the word "w" with an appropriate number following. Thus hyphenated, quoted, etc words will not cause a problem for the SphinxTrain code.

After the prompt utterances are generated we can setup the SphinxTrain directory `st/`. All processing and output files are done within that directory until the file con-

version of labels back into the voice's own phone set and put in `lab/`. Note this process takes a long time, at least several hours and possible several days if you have a particularly slow machine or particularly large database. Also this may require around a half a gigabyte of space.

The script `./bin/sphinxtrain` does the work of converting the FestVox database into a form suitable for SphinxTrain. In all there are 6 steps: setup, building files, converting waveforms, the training itself, alignment and conversion of label files back into FestVox format. The training stage itself consist of 11 parts and by far takes the most time.

This script requires the environment variables `SPHINXTRAINDIR` and `SPHINX2DIR` to be set point to compiled versions of SphinxTrain and Sphinx2 respectively, as shown above.

The first step is to set up the sub-directory `st/` where the training will take place.

```
./bin/sphinxtrain setup
```

The training database name will be taken from your `etc/voice.defs`, if you don't have one of those use

```
$FESTVOXDIR/src/general/guess_voice_defs
```

The next stage is to convert the database prompt list into a transcription file suitable for SphinxTrain,; construct a lexicon, and phone file etc. All of the generate files will be put in `st/etc/`. Note because of various limitations in Sphinx2 and SphinxTrain, the lexicon (`.dic`), and transcription (`.transcription`) will not have what you might thing are sensible values. The word names are take from the utts if they consist of only upper and lower case characters. A number is added to make them unique. Also if another work exists with the same pronunciation but different word it may be assigned a differ name from what you expect. The word names in the SphinxTrain files are only there to help debugging and are really referring to specific instances of words in the utterance (to ensure the pronunciations are preserved with respect to homograph disambiguation and post lexical rules. If people complain about these being confusing I will make all words simple "w" followed by a number.

```
./bin/sphinxtrain files
```

The next stage is to generate the mfcc's for SphinxTrain unfortunately these must be in a different format from the mfcc's used in FestVox, also SphinxTrain only supports raw headered files, and NIST header files, so we copy the waveform files in `wav/` into the `st/wav/` directory converting them to NIST headers

```
./bin/sphinxtrain feats
```

Now we can start the training itself. This consists of eleven stages each which will be run automatically.

- Module 0 checks the basic files for training. There should be no errors at this stage
- Module 1 builds the vector quantization parameters.
- Module 2 builds context-independent phone models. This runs Baum-Welch over the data building context-independent HMM phone models. This runs for several passes until convergences (somewhere between 4 and 15 passes). There may be some errors on some files (especially long, or badly transcribed ones), but a small number of errors here (with the identified file being "ignored" should be ok.

- Module 3 makes the untied model definition.
- Module 4 builds context dependent models.
- Module 5a builds trees for asking questions for tied-states.
- Module 5b builds trees. One for each state in each HMM. This part takes the longest time.
- Module 6 prunes trees.
- Module 7 retrain context dependent models with tied states.
- Module 8 deleted interpolation
- Module 9 convert the generated models to Sphinx2 format

All of the above stages should be run together with the command as

```
./bin/sphinxtrain train
```

Once trained we can use these models to align the labels against the recorded prompts.

```
./bin/sphinxtrain align
```

Some utterances may fail to be labeled at this point, either because they are too long, or their orthography does not match the acoustics. There is not simple solution for this at present. For some you wimple not get a label file, and you can either label the utterance by hand, exclude if from the data, or split it into a smaller file. Other times Sphinx2 will crash and you'll need to remove the utterances from the `st/etc/*.align` and `st/etc/*.ctl` and run the script `./bin/sphinx_lab` by hand.

The final stage is to take the output from the alignment and convert the labels back into their FestVox format. If everything worked to this stage, this final stage should be uneventful.

```
./bin/sphinxtrain labs
```

There should be a set of reasonable phone labels in `prompt-lab/`. These can the be merged into the original utterances with the command

```
festival -b festvox/build_clunits.scm '(build_utts "etc/txt.done.data")'
```

Prosodic Labeling

FO, Accents, Phrases etc.

Notes

1. http://cmusphinx.org
2. http://sourceforge.net/projects/cmusphinx/
3. http://www.speech.cs.cmu.edu/SphinxTrain/

Chapter 15. Evaluation and Improvements

This chapter discusses evaluation of speech synthesis voices and provides a detailed procedure to allow diagnostic testing of new voices.

Evaluation

Now that you have built your voice, how can you tell if it works, and how can you find out what you need to make it better. This chapter deals with some issues of evaluating a voice in Festival. Some of the points here also apply to testing and improving existing voices too.

The evaluation of speech synthesis is notoriously hard. Evaluation in speech recognition was the major factor in making general speech recognition work. Rigourous tests on well defined data made the evaluation of different techniques possible. Though in spite of its success the strict evaluation criteria as used in speech recognition can cloud the ultimate goal. It is important always to remember that tests are there to evaluate a systems performance rather than become the task itself. Just as techniques can overtrain on data it is possible to over train on the test data and/or methodology too thus loosing the generality and purpose of the evaluation.

In speech recognition a simple (though naive) measure of phones or words correct gives a reasonable indicator of how well a speech recognition system works. In synthesis this a lot harder. A word can have multiple pronunciations, so it is much harder to automatically test if a synthesizer's phoneme accuracy, besides much of the quality is not just in if it is correct but if it "sounds good". This is effectly the crux of the matter. The only real synthesis evaluation technique is having a human listen to the result. Humans individually are not very reliably testers of systems, but humans in general are. However it is usually not feasible to have testers listen to large amounts of synthetic speech and return a general goodness score. More specific tests are required.

Although listening tests are the ultimate, because they are expensive in resources (undergraduates are not willing to listing to bad synthesis all day for free), and the design of listening tests is a non-trivial task, there are a number of more general tests which can be run at less expenses and can help greatly.

It is common that a new voice in Festival (or any other speech synthesis systems), has limitations and it is wise to test what the limitations are and decide if such limitations are acceptable or not. This depends a lot on what you wish to use your voice for. For example if the voice a Scottish English voice to be primarily used as the output of a Chinese speech tranlation system, the vocabulary is constained by the translation system itself so a large lexicon is probabaly not much of an issue, but the vocabulary will include many anglosized (calenodianized ?) versions of Chinese names, which are not common in standard English so letter-to-sound rules should be made more sensitive for that input. If the system is to be used to read address lists, it should be able to tokenize names and address appropriately, and if it is to be used in a dialogue system the intonation model should be able to deal with questions and continuations properly. Optimizing your voices for the most common task, and minimizing the errors is what evaluation is for.

Does it work at all?

It is very easy to build a voice and get it to say a few phrases and think that the job is done. As you build the voice it is worth testing each part as you built it to ensure it basically performs as expected. But once its all together more general tests are needed. Before you submit it to any formal tests that you will use for benchmarking and grading progrees in the voice, more basic tests should be carried out.

In fact it is stating such initial tests more concretely. *Every* we have ever built has always had a number mistakes in it that can be trivially fixed. Such as the mfccs were not generated after fixing the pitchmarks. Therefore you syould go through each stage of the build procedure and ensure it really did do what you though it should do, especially if you are totally convinced that section worked perfectly.

Try to find around 100-500 sentences to play through it. It is amazing home many general problems are thrown up when you extend your test set. The next stage is to play so *real* text. That may be news text from the web, output from your speech translation system, or some email. Initially it is worth just synthesizing the whole set without even listening to it. Problems in analysis and missing diphones etc may be shown up just in the processing of the text. Then you want to listen to the output and identify problems. This make take some amount of investigation. What you want to do is identify *where* the problem is, is it bad tex analysis, bad lexical entry, a prosody problem, or a waveform synthesis problem. You may need to synthesizes parts of the text in isolation (e.g. using the Festival function `SayText` and look at the structure of the utterance generated, e.g. using the function `utt.features`. For example to see what words have been identified from the text analysis

```
(utt.features utt1 'Word '(name))
```

Or to see the phones generated

```
(utt.features utt1 'Segment '(name))
```

Thus you can view selected parts of an utterance and find out if it is being created as you intended. For some things a graphical display of the utterance may help.

Once you identify *where* the problem is you need to decide how to fix it (or if it is worth fixing). The problem may be a number of different places:

- Phonetic error: the acoustics of a unit doesn't match the label. This may be because the speaker said the wrong word/phoneme or the labeller had the wrong. Or possible some other acoustic variant that has not been considered
- Lexical error: the word is pronounced with the wrong string of phonemes/stress/tone. Either the lexical entry is wrong or the letter to sound rules are not doing ht right thing. Or there are multiple valid pronunciations for that word (homographs) and the wrong one is selectec because the homograph disambiguation is wrong, or there is not a disambiguator.
- Text error: the text analysis doesn't deal properly with the word. It may be that a punctuation system is spoken (or not spoken) as expected, titles, symbols, compounds etc aren't dealt with properly
- Some other error: some error that is not one of the above. As you progress in correction and tuningm errors in the category will grow and you must find some way to avoid such errors.

Before rushing out and getting one hundred people to listen to your new synthetic voice, it is worth doing significant internal testing and evaluation, informally to find errors and test them. Remember the purpose of evaluation in this case is to find errors and fix them. We are not, at least not at this stage, evaluating the voices on an abstract scale, where unseen test data, and blind testing is important.

Formal Evaluation Tests

Once you yourself and your immediate colleages have tests the voice you will want more formal evaluation metrics. Again we are looking at diagnositic evluation, comparative eveluation between different commercial synthesizers is quite a different task.

In our English checks we used Wall Street Journal and Time magazine articles (around 10 millions words in total). Many unusual words apear only in one article (e.g proper names) which are less important to add to the lexicon, but unusual words that appear across articales are more likely to appear again so should be added.

Be aware that using data will cause your coverage to be biased towards that type of data. Our databases are mostly collected in the early 90s and hence have good coverage for the Gulf War, and the changes in Eastern Europe but our ten million words have no occurences of the words "*Sojourner*" or "*Lewinski*" whcih only appear in stories later in the decade.

A script is provided in `src/general/find_unknowns` which will analyze given text to find which words do not appear in the current lexicon. You should use the `-eval` option to specify the selection of your voice. Note this checks to see which words are not in the lexicon itself, it replaces what ever letter-to-sound/ unknown word function you specified and saves any words for which that function is called in the given output file. For example

```
find_unknowns -eval '(voice_ked_diphone)' -output cmudict.unknown \
      wsj/wsj-raw/00/*
```

Normally you would run this over your database then cummulate the unknown words, then rerun the unknown words synthesizing each and listening to them to evaluate if your LTS system produces reasonable results. Fur those words which do have acceptable pronunciations add them to your lexicon.

Sematically unpredictable sentences

One technique that has been used to evaluation speech synthesis quality is testing against semantically unpredictable sentences.

%%%%%%%%%%%%%%%%%%%%%%
Discussion to be added
%%%%%%%%%%%%%%%%%%%%%%

Debugging voices

Chapter 16. Markup

SABLE, JSML, VoiceXML, Docbook, tts_modes

Chapter 17. Concept-to-speech

Dialog systems, speech-to-speech translation, getting more that just text.

Chapter 18. Deployment

client/server access, footprint, scaling up, backoff strategies, signal commpression.

Chapter 19. A Japanese Diphone Voice

In this chapter we work through a full example of creating a voice given that most of the basic construction work (model building) has been done. Pariticularly this discusses the scheme files, and conventions for keeping a voices together and how you can go about packaging it for general use.

Ultimately a voice in Festival will consist of a diphone database, a lexicon (and lts rules) and a number of scheme files that offer the complete voice. When people other than the developer of a voice wish to use your newly developed voice it is only that small set of files that are required and need to be distributed (freely or otherwise). By convention we have distributed diphone group files, a single file holding the index, and diphone data itself, and a set scheme files that describe the voice (and its necessary models).

Basic skeleton files are included in the festvox distribution. If you are unsure how to go about building the basic files it is recommended you follow this schema and modify these to your particular needs.

By convention a voice name consist of an institution name (like *cmu*, *cstr*, etc), if you don't have an insitution just use *net*. Second you need to identify the language, there is an ISO two letter standard for it fails to distinguish dialects (such as US and UK English) so it need not be strictly followed. However a short identifier for the language is probably prefered. Third you identify the speaker, we have typically used three letter initials which are the initials of the person speaker but any name is reasonable. If you are going to build a US or UK English voice you should look Chapter 20.

The basic processes you will need to address

- construct basic template files
- generate phoneset definition
- generate diphone schema file
- generate prompts
- record speaker
- label nonsense words
- extract picthmarks and LPC coeffcient
- test phone synthesis
- add lexicon/LTS support
- add tokenization
- add prosody (phrasing, durations and intonation)
- test and evaluate voice
- package for distribution

As with all parts of `festvox`: you must set the following enviroment variables to where you have installed versions of the Edinburgh Speech Tools and the festvox distribution

```
export ESTDIR=/home/awb/projects/1.4.1/speech_tools
export FESTVOXDIR=/home/awb/projects/festvox
```

In this example we will build a Japanese voice based on awb (a gaijin). First create a directory to hold the voice.

```
mkdir ~/data/cmu_ja_awb_diphone
cd ~/data/cmu_ja_awb_diphone
```

You will need in the regions of 500M of space to build a voice. Actually for Japanese its probably considerably less, but you must be aware that voice building does require disk space.

Construct the basic directory structure and skeleton files with the command

```
$FESTVOXDIR/src/diphones/setup_diphone cmu ja awb
```

The three arguments are, institution, language and speaker name.

The next stage is define the phoneset in `festvox/cmu_ja_phones.scm`. In many cases the phoneset for a language has been defined, and it is wise to follow convention when it exists. Note that the default phonetic features in the skeleton file may need to be modified for other languages. For Japanese, there are standards and here we use a set similar to the ATR phoneset used by many in Japan for speech processing. (This file is included, but *not* automatically installed, in `$FESTVOXDIR/src/vox_diphone/japanese`

Now you must write the code that generates the diphone schema file. You can look at the examples in `festvox/src/diphones/*_schema.scm`. This stage is actually the first *difficult* part, getting thsi right can be tricky. Finding all possible phone-phone in a language isn't as easy as it seems (especially as many possible ones don't actually exist). The file `festvox/ja_schema.scm` is created providing the function `diphone-gen-list` which returns a list of nonsense words, each consisting of a list of, list of diphones and a list of phones in the nonsense word. For example

```
festival> (diphone-gen-list)
((("k-a" "a-k") (pau t a k a k a pau))
 (("g-a" "a-g") (pau t a g a g a pau))
 (("h-a" "a-h") (pau t a h a h a pau))
 (("p-a" "a-p") (pau t a p a p a pau))
 (("b-a" "a-b") (pau t a b a b a pau))
 (("m-a" "a-m") (pau t a m a m a pau))
 (("n-a" "a-n") (pau t a n a n a pau))
 ...)
```

In addition to generating the diphone schema the `ja_schema.scm` also should provied the functions `Diphone_Prompt_Setup`, which is called before generating the prompts, and `Diphone_Prompt_Word`, which is called before waveform synthesis of each nonsense word.

`Diphone_Prompt_Setup`, should be used to select a speaker to generate the prompts. Note even though you may not use the prompts when recording they are necessary for labeling the spoken speech, so you still need to generate them. If you haeva synthesizer already int eh language use ti to generate the prompts (assuming you can get it to generate from phone lists also generate label files). Often the MBROLA project already has a waveform synthesizer for the language so you can use that. In this case we are going to use a US English voice (kal_diphone) to generate the prompts. For Japanese that's probably ok as the Japanese phoneset is (mostly) a subset of the English phoneset, though using the generated prompts to prompt the user is probably not a good idea.

The second function `Diphone_Prompt_Word`, is used to map the Japanese phone set to the US English phone set so that waveform synthesis will work. In this case a simple map of Japanese phone to one or more English phones is given and the code simple changes the phone name in the segment relation (and adds a new new segment in the multi-phone case).

Now we can generate the diphone schema list.

```
festival -b festvox/diphlist.scm festvox/ja_schema.scm \
   '(diphone-gen-schema "ja" "etc/jadiph.list")'
```

Its is worth checking `etc/jadiph.list` by hand to you are sure it contains all the diphone you wish to use.

The diphone schema file, in this case `etc/jadiph.list`, is a fundamentally key file for almost all the following scripts. Even if you generate the diphone list by some method other than described above, you should generate a schema list in exactly this format so that everything esle will work, modifying the other scripts for some other format is almost certainly a waste of your time.

The schema file has the following format

```
( ja_0001 "pau t a k a k a pau"  ("k-a" "a-k"))
( ja_0002 "pau t a g a g a pau"  ("g-a" "a-g") )
( ja_0003 "pau t a h a h a pau"  ("h-a" "a-h") )
( ja_0004 "pau t a p a p a pau"  ("p-a" "a-p") )
( ja_0005 "pau t a b a b a pau"  ("b-a" "a-b") )
( ja_0006 "pau t a m a m a pau"  ("m-a" "a-m") )
( ja_0007 "pau t a n a n a pau"  ("n-a" "a-n") )
( ja_0008 "pau t a r a r a pau"  ("r-a" "a-r") )
( ja_0009 "pau t a t a t a pau"  ("t-a" "a-t") )
...
```

In this case it has 297 nonsense words.

Next we can generate the prompts and their label files with the following command The to synthesize the prompts

```
festival -b festvox/diphlist.scm festvox/ja_schema.scm \
   '(diphone-gen-waves "prompt-wav" "prompt-lab" "etc/jadiph.list")'
```

Occasionally when you are building the prompts some diphones requested in the prompt voice don't actually exists (especially when you are doing cross-language prompting). Thus the generated prompt has some default diphone (typically silence-silence added). This is mostly ok, as long as its not happening multiple times in the same nonsence word. The speaker just should be aware that some prompts aren't actually correct (which of course is going to be true for all prompts in the cross-language prompting case).

The stage is to record the prompts. See the Section called *Recording under Unix* in Chapter 4 for details on how to do this under Unix (and in fact other techniques too). This can done with the command

```
bin/prompt_them etc/jadiph.list
```

Depending on whether you want the prompts actually to be played or not, you can edit `bin/prompt_them` to comment out the playing of the prompts.

Note a third argument can be given to state which nonse word to begin prompting from. This if you have already recorded the first 100 you can continue with

```
bin/prompt_them etc/jadiph.list 101
```

The recorded prompts can the be labeled by

```
bin/make_labs prompt-wav/*.wav
```

And the diphone index may be built by

```
bin/make_diph_index etc/awbdiph.list dic/awbdiph.est
```

If no EGG signal has been collected you can extract the pitchmarks by

```
bin/make_pm_wave wav/*.wav
```

If you do have an EGG signal then use the following instead

```
bin/make_pm lar/*.lar
```

A program to move the predicted pitchmarks to the nearest peak in the waveform is also provided. This is almost always a good idea, even for EGG extracted pitch marks

```
bin/make_pm_fix pm/*.pm
```

Getting good pitchmarks is important to the quality of the synthesis, see the Section called *Extracting pitchmarks from waveforms* in Chapter 4 for more discussion.

Because there is often a power mismatch through a set of diphone we provided a simple method for finding what general power difference exist between files. This finds the mean power for each vowel in each file and calculates a factor with respect to the overal mean vowel power. A table of power modifiers for each file can be calculated by

```
bin/find_powerfactors lab/*.lab
```

The factors cacluated by this are saved in `etc/powfacts`.

Then build the pitch-synchronous LPC coefficients, which used the power factors if they've been calculated.

```
bin/make_lpc wav/*.wav
```

This should get you to the stage where you can test the basic waveform synthesizer. There is still much to do but initial tests (and correction of labeling errors etc) can start now. Start festival as

```
festival festvox/cmu_ja_awb_diphone.scm "(voice_cmu_ja_awb_diphone)"
```

and then enter string of phones

```
festival> (SayPhones '(pau k o N n i ch i w a pau))
```

In addition to the waveform generate part you must also provide text analysis for your language. Here, for the sake of simplicity we assume that the Japanese is provided in romanized form with spaces between each word. This is of course not the case for normal Japanese (and we are working on a proper Japanese front end). But at present this shows the general idea. Thus we edit `festvox/cmu_ja_token.scm` and add (simple) support for numbers.

As the relationship between romaji (romanized Japanese) and phones is almost trivial we write a set of letter-to-sound rules, by hand that expand words into their phones. This is added to `festvox/cmu_ja_lex.scm`.

For the time being we just use the default intonation model, though simple rule drive improvements are possible. See `festvox/cmu_ja_awb_int.scm`. For duration, we add a mean value for each phone in the phoneset to `fextvox/cmu_ja_awb_dur.scm`.

These three japanese specific files are included in the distribution in `festvox/src/vox_diphone/japanese/`.

Now we have a basic synthesizer, although there is much to do, we can now type (romanized) text to it.

```
festival festvox/cmu_ja_awb_diphone.scm "(voice_cmu_ja_awb_diphone)"
...
festival> (SayText "boku wa gaijin da yo.")
```

The next part is to test and improve these various initial subsystems, lexicons, text analysis prosody, and correct waveform synthesis problem. This is ane endless task but you should spend significantly more time on it that we have done for this example.

Once you are happy with the completed voice you can package it for distribution. The first stage is to generate a group file for the diphone database. This extracts the subparts of the nonsense words and puts them into a single file offering something smaller and quicker to access. The groupfile can be built as follows.

```
festival festvox/cmu_ja_awb_diphone.scm "(voice_cmu_ja_awb_diphone)"
...
festival (us_make_group_file "group/awblpc.group" nil)
...
```

The `us_` in the function names stands for `UniSyn` (the unit concatenation subsystem in Festival) and nothing to do with US English.

To test this edit `festvox/cmu_ja_awb_diphone.scm` and change the choice of databases used from separate to grouped. This is done by commenting out the line (around line 81)

```
(set! cmu_ja_awb_db_name (us_diphone_init cmu_ja_awb_lpc_sep))
```

and uncommented the line (around line 84)

```
(set! cmu_ja_awb_db_name (us_diphone_init cmu_ja_awb_lpc_group))
```

The next stage is to integrate this new voice so that festival may find it automatically. To do this you should add a symbolic link from the voice directory of Festival's English voices to the directory containing the new voice. Frist cd to festival's voice directory (this will vary depending on where your version of festival is installed)

```
cd /home/awb/projects/1.4.1/festival/lib/voices/japanese/
```

creating the language directory if it does not already exists. Add a symbolic link back to where your voice was built

```
ln -s /home/awb/data/cmu_ja_awb_diphone
```

Now this new voice will be available for anyone runing that version festival started from any directory, without the need for any explicit arguments

```
festival
...
festival> (voice_cmu_ja_awb_diphone)
```

```
...
festival> (SayText "ohayo gozaimasu.")
...
```

The final stage is to generate a distribution file so the voice may be installed on other's festival installations. Before you do this you must add a file `COPYING` to the directory you built the diphone database in. This should state the terms and conditions in which people may use, distribute and modify the voice.

Generate the distribution tarfile in the directory above the festival installation (the one where `festival/` and `speech_tools/` directory is).

```
cd /home/awb/projects/1.4.1/
tar zcvf festvox_cmu_ja_awb_lpc.tar.gz \
  festival/lib/voices/japanese/cmu_ja_awb_diphone/festvox/*.scm \
  festival/lib/voices/japanese/cmu_ja_awb_diphone/COPYING \
  festival/lib/voices/japanese/cmu_ja_awb_diphone/group/awblpc.group
```

The completed files from building this crude Japanese example are available at http://festvox.org/examples/cmu_ja_awb_diphone/.

Notes

1. http://festvox.org/examples/cmu_ja_awb_diphone/

Chapter 20. US/UK English Diphone Synthesizer

When building a new diphone based voice for a supported language, such as English, the upper parts of the systems can mostly be taken from existing voices, thus making the building task simpler. Of course, things can still go wrong, and its worth checking everything at each stage. This section gives the basic walkthrough for building a new US English voice. Support for building UK (southern, RP dialect) is also provided this way. For building non-US/UK synthesizers see Chapter 19 for a similar walkthrough but also covering the full text, lexicona nd prosody issues which we can subsume in this example.

Recording a whole diphone set usually takes a number of hours, *if* everything goes to plan. Construction of the voice after recording may take another couple of hours, though much of this is CPU bound. Then hand-correction may take at least another few hours (depending on the quality). Thus if all goes well it is possible to construct a new voice in a day's work though usually something goes wrong and it takes longer. The more time you spend making sure the data is correctly aligned and labeled, the better the results will be. While something can be made quickly, it can take much longer to do it very well.

For those of you who have ignored the rest of this document and are just hoping to get by by reading this, good luck. It may be possible to do that, but considering the time you'll need to invest to build a voice, being familar with the comments, at least in the rest of this chapter, may be well worth the time invested.

The tasks you will need to do are:

- construct basic template files
- generate prompts
- record nonsense words
- autolabel nonsense words
- generate diphone index
- generate pitchmarks and LPC coefficients
- Test, and hand fix diphones
- Build diphone group files and distribution

As with all parts of `festvox`, you must set the following environment variables to where you have installed versions of the Edinburgh Speech Tools and the festvox distribution

```
export ESTDIR=/home/awb/projects/1.4.1/speech_tools
export FESTVOXDIR=/home/awb/projects/festvox
```

The next stage is to select a directory to build the voice. You will need in the order of 500M of diskspace to do this, it could be done in less, but its better to have enough to start with. Make a new directory and cd into it

```
mkdir ~/data/cmu_us_awb_diphone
cd ~/data/cmu_us_awb_diphone
```

By convention, the directory is named for the institution, the language (here, `us` English) and the speaker (`awb`, who actually speaks with a Scottish accent). Although it can be fixed later, the directory name is used when festival searches for available voices, so it is good to follow this convention.

Build the basic directory structure

```
$FESTVOXDIR/src/diphones/setup_diphone cmu us awb
```

the arguments to `setup_diphone` are, the institution building the voice, the language, and the name of the speaker. If you don't have a institution we recommend you use `net`. There is an ISO standard for language names, though unfortunately it doesn't allow distinction between US and UK English, so in general we recommend you use the two letter form, though for US English use `us` and UK English use `uk`. The speaker name may or may nor be there actual name.

The setup script builds the basic directory structure and copies in various skeleton files. For languages `us` and `uk` it copies in files with much of the details filled in for those languages, for other languages the skeleton files are much more skeletal.

For constructing a `us` voice you must have the following installed in your version of festival

```
festvox_kallpc16k
festlex_POSLEX
festlex_CMU
```

And for a UK voice you need

```
festvox_rablpc16k
festlex_POSLEX
festlex_OALD
```

At run-time the two appropriate festlex packages (POSLEX + dialect specific lexicon) will be required but not the existing kal/rab voices.

To generate the nonsense word list

```
festival -b festvox/diphlist.scm festvox/us_schema.scm \
   '(diphone-gen-schema "us" "etc/usdiph.list")'
```

We use a synthesized voice to build waveforms of the prompts, both for actual prompting and for alignment. If you want to change the prompt voice (e.g. to a female) edit `festvox/us_schema.scm`. Near the end of the file is the function `Diphone_Prompt_Setup`. By default (for US English) the voice `(voice_kal_diphone)` is called. Change that, and the F0 value in the following line, if appropriate, to the voice use wish to follow.

Then to synthesize the prompts

```
festival -b festvox/diphlist.scm festvox/us_schema.scm \
   '(diphone-gen-waves "prompt-wav" "prompt-lab" "etc/usdiph.list")'
```

Now record the prompts. Care should be taken to set up the recording environment so it is best. Note all power levels so that if more than one session is required you can continue and still get the same recording quality. Given the length of the US English list, its unlikely a person can say allow of these in one sitting without taking breaks at least, so ensuring the environment can be duplicated is important, even if it's only after a good stretch and a drink of water.

```
bin/prompt_them etc/usdiph.list
```

Note a third argument can be given to state which nonse word to begin prompting from. This if you have already recorded the first 100 you can continue with

```
bin/prompt_them etc/usdiph.list 101
```

See the Section called *US phoneset* in Chapter 30 for notes on pronunciation (or the Section called *UK phoneset* in Chapter 30 for the UK version).

The recorded prompts can the be labeled by

```
bin/make_labs prompt-wav/*.wav
```

Its is always worthwhile correcting the autolabeling. Use

```
emulabel etc/emu_lab
```

and select `FILE OPEN` from the top menu bar and the place the other dialog box and clink inside it and hit return. A list of all label files will be given. Double-click on each of these to see the labels, spectragram and waveform. (** reference to "How to correct labels" required **).

Once the diphone labels have been corrected, the diphone index may be built by

```
bin/make_diph_index etc/usdiph.list dic/awbdiph.est
```

If no EGG signal has been collected you can extract the pitchmarks by (though read the Section called *Extracting pitchmarks from waveforms* in Chapter 4 to ensure you are getting the best exteraction).

```
bin/make_pm_wave wav/*.wav
```

If you do have an EGG signal then use the following instead

```
bin/make_pm lar/*.lar
```

A program to move the predicted pitchmarks to the nearest peak in the waveform is also provided. This is almost always a good idea, even for EGG extracted pitch marks

```
bin/make_pm_fix pm/*.pm
```

Getting good pitchmarks is important to the quality of the synthesis, see the Section called *Extracting pitchmarks from waveforms* in Chapter 4 for more discussion.

Because there is often a power mismatch through a set of diphone we provided a simple method for finding what general power difference exist between files. This finds the mean power for each vowel in each file and calculates a factor with respect to the overall mean vowel power. A table of power modifiers for each file can be calculated by

```
bin/find_powerfactors lab/*.lab
```

The factors calculated by this are saved in `etc/powfacts`.

Then build the pitch-synchronous LPC coefficients, which use the power factors if they've been calculated.

```
bin/make_lpc wav/*.wav
```

Now the database is ready for its initial tests.

```
festival festvox/cmu_us_awb_diphone.scm '(voice_cmu_us_awb_diphone)'
```

When there has been no hand correction of the labels this stage may fail with diphones not having proper start, mid and end values. This happens when the automatic labeled has position two labels at the same point. For each diphone that has a problem find out which file it comes from (grep for it in `dic/awbdiph.est` and use `emulabel` to change the labeling to as its correct. For example suppose "ah-m" is wrong you'll find is comes from `us_0314`. Thus type

```
emulabel etc/emu_lab us_0314
```

After correcting labels you must re-run the `make_diph_index` command. You should also re-run the `find_powerfacts` stage and `make_lpc` stages as these too depend on the labels, but this takes longer to run and perhaps that need only be done when you've corrected many labels.

To test the voice's basic functionality with

```
festival> (SayPhones '(pau hh ax l ow pau))

festival> (intro)
```

As the autolabeling is unlikely to work completely you should listen to a number of examples to find out what diphones have gone wrong.

Finally, once you have corrected the errors (did we mention you need to check and correct the errors?), you can build a final voice suitable for distribution. First you need to create a group file which contains only the subparts of spoken words which contain the diphones.

```
festival festvox/cmu_us_awb_diphone.scm '(voice_cmu_us_awb_diphone)'
...
festival (us_make_group_file "group/awblpc.group" nil)
...
```

The `us_` in this function name confusingly stands for `UniSyn` (the unit concatenation subsystem in Festival) and nothing to do with US English.

To test this edit `festvox/cmu_us_awb_diphone.scm` and change the choice of databases used from separate to grouped. This is done by commenting out the line (around line 81)

```
(set! cmu_us_awb_db_name (us_diphone_init cmu_us_awb_lpc_sep))
```

and uncommented the line (around line 84)

```
(set! cmu_us_awb_db_name (us_diphone_init cmu_us_awb_lpc_group))
```

The next stage is to integrate this new voice so that festival can find it automatically. To do this, you should add a symbolic link from the voice directory of Festival's English voices to the directory containing the new voice. First cd to festival's voice directory (this will vary depending on where you installed festival)

```
cd /home/awb/projects/1.4.1/festival/lib/voices/english/
```

add a symbolic link back to where your voice was built

```
ln -s /home/awb/data/cmu_us_awb_diphone
```

Now this new voice will be available for anyone runing that version festival (started from any directory)

```
festival
...
festival> (voice_cmu_us_awb_diphone)
...
festival> (intro)
...
```

The final stage is to generate a distribution file so the voice may be installed on other's festival installations. Before you do this you must add a file `COPYING` to the directory you built the diphone database in. This should state the terms and conditions in which people may use, distribute and modify the voice.

Generate the distribution tarfile in the directory above the festival installation (the one where `festival/` and `speech_tools/` directory is).

```
cd /home/awb/projects/1.4.1/
tar zcvf festvox_cmu_us_awb_lpc.tar.gz \
  festival/lib/voices/english/cmu_us_awb_diphone/festvox/*.scm \
  festival/lib/voices/english/cmu_us_awb_diphone/COPYING \
  festival/lib/voices/english/cmu_us_awb_diphone/group/awblpc.group
```

The complete files from building an example US voice based on the KAL recordings is available at http://festvox.org/examples/cmu_us_kal_diphone/.

Notes

1. http://festvox.org/examples/cmu_us_kal_diphone/

Chapter 21. ldom full example

domain analysis, prompt designing build/walkthrough and debugging. weather or stocks or such like.

Chapter 22. Non-english ldom example

Cross linguistic build with minimal local language support.

Chapter 23. Concluding remarks and future

Where will it all lead to.

Chapter 24. Festival Details

This chapter offers descriptions of various Festival internals including APIs.

Chapter 25. Festival's Scheme Programming Language

This chapter acts as a reference guide for the particular dialect of the Scheme programming language used in the Festival Speech Synthesis systems. The Scheme programming language is a dialect of Lisp designed to be more consistent. It was chosen for the basic scripting language in Festival because:

- it is a very easy language for machines to parse and interpret, thus the foot print for the interpreter proper is very small
- it offers garbage collection making managing objects safe and easy.
- it offers a general consistent datastructure for representing parameters, rules etc.
- it was familiar to the authors
- its is suitable for use as an embedded system

Having a scripting language in Festival is actually one of the fundamental properties that makes Festival a useful system. The fact that new voices and languages in many cases can be added without changing the underlying C++ code makes the system mouch more powerful and accessible than a more monolithic system that requires recompilation for any parameter changes. As there is sometimes confusion we should make it clear that Festival contains its own Scheme interpreter as part of the system. Festival can be view as a Scheme interpreter that has had basic addition to its function to include modules that can do speech synthesis, no external Scheme interperter is required to use Festival.

The actual interpreter used in Festival is based on George Carret's SIOD, "Scheme in one Defun". But this has been substantially enhanced from its small elegant beginnings into something that might be better called "Scheme in one directory". Although there is a standard for Scheme the version in Festival does not fully follow it, for both good and bad reasons. Thus finding in order for people to be able to program in Festival's Scheme we provide this chapter to list the core type, functions, etc and some examples. We do not pretend to be teaching programming here but as we know many people who are interested in building voices are not primarily programmers, some guidance on the language and its usage will make the simple programming that is required in building voices, more accessible.

For reference the Scheme Revised Revised Revised report describes the standard definition [srrrr90]. For a good introduction to programming in general that happens to use Scheme as its example language we recommend [abelson85]. Also for those who are unfamiliar with the use of Lisp-like scripting languages we recommend a close look as GNU Emacs which uses Lisp as its underlying scripting language, knowledge of the internals of Emacs did to some extent influence the scripting language design of Festival.

Overview

"Lots of brackets" is what comes to most people's minds when considering Lisp and its various derivatives such as Scheme. At the start this can seem daunting and it is true that parenthesis errors can cuase problems. But with an editor that does proper bracket matching, brackets can actually be helpful in code structure rather than a hindrance.

The fundamental structure is the *s-expression*. It consists of an atom, or a list of s-expressions. This simply defined recursive structure allows complex structures to easily be specified. For example

```
3
(1 2 3)
(a (b c) d)
```

```
((a b) (d e))
```

Unlike other programming languages Scheme's data and code are in the same format, s-expressions. Thus s-expression are evaluated, recursively.

Symbols:

are treated as variables and evaluated return their currently set value.

Strings and numbers:

evalutate to themselves.

Lists:

The each member of the list is evaluated and the first item in the list is treated as a function and applied using the remainer of the list as arguments to the function.

Thus the s-expression

```
(+ 1 2)
```

when evaluated will return `3` as the symbol `+` is bound to a function that adds it arguments.

Variables may be set using the `set!` function which takes a variable name and a value as arguments

```
(set! a 3)
```

The `set!` function is unusual in that it does not evaluate its first argument. If it did you have to explcitly quote it or set some other variable to have a value of `a` to get the desired effect.

quoting, define

Data Types

There a number of basic data types in this Scheme, new ones may also be added but only through C++ functions. This basic types are

Symbols:

symbols atoms starting with an alphabetic character. Unlike numbers and strings, they may be used as variables. Examples are

```
a bcd f6 myfunc
plus cond
```

Symbols may be created from strings by using the function `intern`

Numbers:

In this version of scheme all numbers are doubles, there is no distinction between floats, doubles and ints. Examples are

```
1
1.4
3.14
345
3456756.4345476
```

Numbers evaluate to themselves, that is the value of the atom 2 is the number 2.

Strings:

Strings are bounded by the double quote characters ". For example

```
"a"
"abc"
"This is a string"
```

Strings evaluate to themselves. They may be converted to symbols with the function `intern`. If they are strings of characaters that represent numbers you can convert a string to a number with the function `parse-number`. For example

```
(intern "abc") => abc
(parse-number "3.14") => 3.14
```

Although you can make symbols from numbers you should not do that.

Double quotes may be specified within a string by escaping it with a backslash. Backslashes therefore also require an escape backslash. That is, `"ab\"c"` contains four characters, `a`, `b`, `"` and `c`. `"ab\\c"` also contains four characters, `a`, `b`, `\` and `c`. And `"ab\\\"c"` contains five characters `a`, `b`, `\`, `"` and `c`.

Lists or Cons

Lists start with a left parenthesis and end with a right parenthesis with zero or more s-expression between them. For example

```
(a b c)
()
(b (b d) e)
((the boy) saw (the girl (in (the park))))
```

Lists can be made by various functions most notably `cons` and `list`. `cons` returns a list whose first item is the first item in the list, standardly called its `car`, and whose remainder, standardly called its `cdr`, is the second argument of `cons`.

```
(cons 'a '(b c)) => (a b c)
(cons '(a b) '(c d)) => ((a b) c d)
```

Functions:

Functions may be applied explicity bu the function `apply` or more normally as when the appear as the first item in a list to be evaluated. The normal way to define function is using the `define` function. For example

```
(define (ftoc temp)
  (/ (* (- temp 32) 5) 9))
```

This binds the function to the variable `ftoc`. Functions can also be defined anonymously which sometimes is convinient.

```
(lambda (temp)
  (/ (* (- temp 32) 5) 9))
```

returns a function.

Others:

other internal types are support by Festival's scheme including some inportant object types use for synthesis such as utterances, waveforms, items etc. The are normally printed as in the form

```
#<Utterance 6234>
#%lt;Wave 1294>
```

The rpint form is a convinience form only. Enter that string of characters will not allow a reference to that object. The number is unique to that object instance

(it is actually the internal address of the object), and can be used visually to note if objects are the same or not.

Functions

This section lists the basic functions in Festival's Scheme. It doesn't list them all (see the Festival manual for that) but does highlight the key functions that you should normally use.

Core functions

These functions are the basic functions used in Scheme. These include the structural functions for setting variables, conditionals. loops, etc.

`(set! SYMBOL VALUE)`

Sets `SYMBOL` to `VALUE`. `SYMBOL` is not evaluated, while `VALUE` is. Example

```
(set! a 3)
(set! pi 3.14)
(set! fruit '(apples pears bananas))
(set! fruit2 fruit)
```

`(define (FUNCNAME ARG0 ARG1 ...) . BODY)`

define a function called `FUNCNAME` with specified arguments and body.

```
(define (myadd a b) (+ a b))
(define (factorial a)
 (cond
  ((< a 2) 1)
  (t (* a (factorial (- a 1))))))
```

`(if TEST TRUECASE [FALSECASE] )`

If the value of `TEST` is non-nil, evaluate `TRUECASE` and return value else if present evaluate `FALSECASE` if present and return value, else return `nil`.

```
(if (string-equal v "apples")
   (format t "It's an apple\n")
   (format t "It's not an apple\n"))
(if (member v '(apples pears bananas))
   (begin
      (format t "It's a fruit (%s)\n" v)
      'fruit)
  'notfruit)
```

`(cond (TEST0 . BODY) (TEST1 . BODY) ...)`

A multiple if statement. Evaluates each `TEST` until a non-nil test is found then evalues each of the expressions in that `BODY` return the value of the last one.

```
(cond
  ((string-equal v "apple")
   'ringo)
  ((string-equal v "plum")
   'ume)
```

```
((string-equal v "peach")
 'momo)
(t
 'kudamono)
```

`(begin . BODY )`

This evaluates each s-expression in `BODY` and returns the value of the last s-expression in the list. This is useful for case where only one s-expression is expected but you need to call a number of functions, notably the `if` function.

```
(if (string-equal v "pear")
   (begin
     (format t "assuming it's a asian pear\n")
     'nashi)
   'kudamono)
```

`(or . DISJ)`

evalutate each disjunct until one is non-nil and return that value.

```
(or (string-equal v "tortoise")
    (string-equal v "turtle"))
(or (string-equal v "pear")
    (string-equal v "apple")
    (< num_fruits 6))
```

`(and . CONJ)`

evalutate each conjunct until one is nil and return that value or return the value of the last conjunct.

```
(and (< num_fruits 10)
     (> num_fruits 3))
(and (string-equal v "pear")
     (< num_fruits 6)
     (or (string-equal day "Tuesday")
         (string-equal day "Wednesday")))
```

List functions

`(car EXPR)`

returns the "car" of `EXPR`, for a list this is the first item, for an atom or the empty list this is defined to be `nil`.

```
(car '(a b)) => a
(car '((a b) c d)) => (a b)
(car '(a (b c) d)) => a
(car nil) => nil
(car 'a) => nil
```

`(cdr EXPR)`

returns the "cdr" of `EXPR`, for a list this is the rest of the list, for an atom or the empty list this is defined to be `nil`.

```
(cdr '(a b)) => (b)
(cdr '((a b) c d)) => (c d)
(cdr '(a)) => nil
(cdr '(a (b c))) => ((b c))
(cdr nil) => nil
(cdr 'a) => nil
```

`(cons EXPR0 EXPR2)`

build a new list whose "car" is `EXPR0` and whose "cdr" is `EXPR1`.

```
(cons 'a '(b c)) => (a b c)
(cons 'a ()) => (a)
(cons '(a b) '(c d) => '((a b) c d))
(cons () '(a) => '(nil a))
(cons 'a 'b => (a . b))
(cons nil nil) => (nil)
```

`(list . BODY)`

Form a list from each of the arguments

```
(list 'a 'b 'c) => (a b c)
(list '(a b) 'c 'd) => ((a b) c d)
(list nil '(a b) '(a b)) => (nil (a b) (a b))
```

`(append . BODY)`

Join each of the arguments (lists) into a single list

```
(append '(a b) '(c d)) => (a b c d)
(append '(a b) '((c d)) '(e f)) => (a b (c d) e f)
(append nil nil) => nil
(append '(a b)) => (a b))
(append 'a 'b) => error
```

`(nth N LIST)`

Return Nth member of list, the first item is the 0th member.

```
(nth 0 '(a b c)) => a
(nth 2 '(a b c)) => c
(nth 3 '(a b c)) => nil
```

`(nth_cdr N LIST)`

Return Nth cdr list, the first cdr is the 0th member, which is the list itself.

```
(nth 0 '(a b c)) => (a b c)
(nth 2 '(a b c)) => (c)
(nth 1 '(a b c)) => (b c)
(nth 3 '(a b c)) => nil
```

`(last LIST)`

The last cdr of a list, traditionally this function has always been called `last` rather `last_cdr`

```
(last '(a b c)) => (c)
(last '(a b (c d))) => ((c d))
```

`(reverse LIST)`

Return the list in reverse order

```
(reverse '(a b c)) => (c b a)
(reverse '(a)) => (a)
(reverse '(a b (c d))) => ((c d) b a)
```

`(member ITEM LIST)`

Returns the cdr in LIST whose car is ITEM or nil if it found

```
(member 'b '(a b c)) => (b c)
(member 'c '(a b c)) => (c)
(member 'd '(a b c)) => nil
(member 'b '(a b c b)) => (b c b)
```

Note that `member` uses `eq` to test equality, hence this does not work for strings. You should use `member_string` if the list contains strings.

`(assoc ITEM ALIST)`

a-list are a standard list format for representing feature value pairs. An a-list is basically a list of pairs of name and value, although the name may be any lisp item it is usually an symbol. A typlical a-list is

```
((name AH)
 (duration 0.095)
 (vowel +)
 (occurs ("file01" "file04" "file07" "file24"))
)
```

`assoc` is a function that allows you to look up values in an a-list

```
(assoc 'name '((name AH) (duration 0.95))) => (name AH)
(assoc 'duration '((name AH) (duration 0.95))) => (duration 0.95)
(assoc 'vowel '((name AH) (duration 0.95))) => nil
```

Note that `assoc` uses `eq` to test equality, hence this does not work names that are strings. You should use `assoc_string` if the a-list uses strings for names.

Arithmetic functions

+ - * / exp log sqrt < > <= >= =

I/O functions

File names in Festival use the Unix convention of using "/" as the directory separator. However under other operating systems, such as Windows, the "/" will be appropriately mapped into backslash as required. For most cases you do not need to worry about this and if you use forward slash all the time ti will work.

`(format FD FORMATSTRING . ARGS)`

The `format` function is a little unusually in Lisp. It basically follows the `printf` command in C, or more closely follows the `format` function in Emacs lisp. It is desgined to print out infomation that is not necessarily to be read in by Lisp (unlike `pprint`, `print` and `printfp`). `FD` is a file descriptor as created by `fopen`, and the result is printed to that. Also two special values are allows there. `t` causes the output to be sent to standard out (which is usually the terminal). `nil` causes the output to be written to a string and returned by the function. Also the variable `stderr` is set to a file descriptor for standard error output.

The format string closely follows the format used in C's printf functions. It is actually interpreted by those functions in its implementation. `format` supports the following directives

%d

Print as integer

%d

Print as integer in hexadecimal

%f

Print as float

%s

Convert item to string

%%

A percent character

%g

Print as double

%c

Print number as character

%l

Print as Lisp object

In addition directive sizes are supported, including (zero or space) padding, and widths. Explicitly specified sizes as arguments as in `%*s` are not supported, nor is `%p` for pointers.

The `%s` directive will try to convert the corresponding lisp argument to a string before passing it to the low level print function. Thus list will be printed to strings, and numbers also coverted. This form will loose the distinction between lisp symbols and lisp strings as the quote will not be present in the `%s` form. In general `%s` should be used for getting nice human output and not for machine readable output as it is a lossy print form.

In contrast `%l` is designed to reserve the Lisp forms so they can be more easily read, quotes will appear and escapes for embedded quote will be treated properly.

```
(format t "duration %0.3f\n" 0.12345) => duration 0.123
(format t "num %d\n" 23) => num 23
(format t "num %04d\n" 23) => num 0023
```

(pprintf SEXP [FD])

Pretty print give expression to standard out (or FD if specified). Pretty printing is a technique that inserts newlines in the printout and indentation to make the lisp expression easier to read.

(fopen FILENAME MODE)

This creates a file description, which can be used in the various I/O functions. It closely follows C stdio `fopen` function. The mode may be

"r"

to open the file for reading

"w"

to open the file for writing

"a"

to open the file at the end for writing (so-called, append).

"b"

File I/O in binary (for OS's that make the distinction),

Or any combination of these.

(fclose FD)

Close a file descriptor as created by `fopen`.

(read)

Read next s-expression from standard in

(readfp FD)

Read next s-expression from given file descriptor FD. On end of file it returns an sexpression eq to the value returned by the function `(eof_val)`. A typical example use of these functions is

```
(let ((ifd (fopen infile "r"))
      (ofd (fopen outfile "w"))
      (word))
  (while (not (equal? (set! word (readfp ifd)) (eof-val)))
    (format ofd "%l\n" (lex.lookup word nil)))
  (fclose ifd)
  (fclose ofd)))
```

(load FILENAME [NOEVAL])

Load in the s-expressions in `FILENAME`. If `NOEVAL` is unspecified the s-expressions are evaluated as they are read. If `NOEVAL` is specified and non-nil, `load` will return all s-expressions in the file un-evaluated in a single list.

String functions

As in many other languages, Scheme has a distinction between `strings` and `symbols`. String evaluate to themselves and cannot be assigned other values, symbols of the print name are `equal?` while strings of teh same name aren't necessarily.

In Festival's Scheme, strings are eight bit clean and designed to hold strings of text and characters in what ever language is being synthesized. Strings are always treats as string of 8 bit characters even though some language may interpret these are 16-bit characters. Symbols, in general, should not contain 8bit characters.

`(string-equal STR1 STR2)`

Finds the string of `STR1` and `STR2` and returns `t` if these are equal, and `nil` otherwise. Symbol names and numbers are mapped to string, though you should be aware that the mapping of a number to a string may not always produce what you hope for. A number `0` may or may not be mapped to `"0"` or maybe to `"0.0"` such that you should not dependent on the mapping. You can use `format` to map a number ot a string in an explicit manner. It is however safe to pass symbol names to `string-equal`. In most cases `string-equal` is the right function to use rather than `equal?` which is must stricter about its definition of equality.

```
(string-equal "hello" "hello") => t
(string-equal "hello" "Hello") => false
(string-equal "hello" 'hello) => t
```

`(string-append . ARGS)`

For each argument coerce it to a string, and return the concatenation of all arguments.

```
(string-append "abc" "def") => "abcdef"
(string-append "/usr/local/" "bin/" "festival") => "/usr/local/bin/festival"
(string-append "/usr/local/" t 'hello) => "/usr/local/thello"
(string-append "abc") => "abc"
(string-append ) => ""
```

`(member_string STR LIST)`

returns `nil` if no member of `LIST` is `string-equal` to `STR`, otherwise it returns `t`. Again, this is often the safe way to check membership of a list as this will work properly if `STR` or the members of `LIST` are symbols or strings.

```
(member_string "a" '("b" "a" "c")) => t
(member_string "d" '("b" "a" "c")) => nil
(member_string "d" '(a b c d)) => t
(member_string 'a '("b" "a" "c")) => t
```

`(string-before STR SUBSTR)`

Returns the initial prefix of `STR` up to the first occurrence of `SUBSTR` in `STR`. If `SUBSTR` doesn't exist within `STR` the empty string is returned.

```
(string-before "abcd" "c") => "ab"
(string-before "bin/make_labs" "/") => "bin"
(string-before "usr/local/bin/make_labs" "/") => "usr"
(string-before "make_labs" "/") => ""
```

`(string-after STR SUBSTR)`

Returns the longest suffix of `STR` after the first occurrence of `SUBSTR` in `STR`. If `SUBSTR` doesn't exist within `STR` the empty string is returned.

```
(string-after "abcd" "c") => "d"
(string-after "bin/make_labs" "/") => "make_labs"
```

```
(string-after "usr/bin/make_labs" "/") => "bin/make_labs"
(string-after "make_labs" "/") => ""
```

`(length STR)`

Returns the lengh of given string (or list). Length does not coerce its argument into a string, hence given a symbol as argument is an error.

```
(length "") => 0
(length "abc") => 3
(length 'abc) -> SIOD ERROR
(length '(a b c)) -> 3
```

`(symbolexplode SYMBOL)`

returns a list of single character strings for each character in `SYMBOL}'` print name. This will also work on strings.

```
(symbolexplode 'abc) => ("a" "b" "c")
(symbolexplode 'hello) => ("h" "e" "l" "l" "o")
```

`(intern STR)`

Convert a string into a symbol with the same print name.

`(string-matches STR REGEX)`

Returns `t` if `STR` matches `REGEX` regular expression. Regular expressions are described more fully below.

```
(string-matches "abc" "a.*") => t
(string-matches "hello" "[Hh]ello") => t
```

System functions

In order to interact more easily with the underlying operating system, Festival Scheme includes a number of basic function that allow Scheme programs to make use of the operating system functions.

`(system COMMAND)`

Evaluates the command with the Unix shell (or equivalent). Its not clear how this should (or does0 work on other operating systems so it should be used sparingly if the code is to be portable.

```
(system "ls") => lists files in current directory.
(system (format nil "cat %s" filename))
```

`(get_url URL OFILE)`

Copies contents of `URL` into `OFILE`. It support `file:` and `http:` prefixes, but current does not support the `ftp:` protocol.

```
(get_url "http://www.festvox.org/index.html" "festvox.html")
```

`(setenv NAME VALUE)`

Set environment variable `NAME` to `VALUE` which should be strings

```
(setenv "DISPLAY" "nara.mt.cs.cmu.edu:0.0")
```

`(getenv NAME)`

Get value of environment variable `NAME`.

```
(getenv "DISPLAY")
```

`(getpid)`

The process id, as a number. This is useful when creating files that need to be unique for the festival instance.

```
(set! bbbfile (format nil "/tmp/stuff.%05d" (getpid)))
```

`(cd DIRECTORY)`

Change directory.

```
(cd "/tmp")
```

`(pwd)`

return a string which is a pathname to the current working directory.

Utterance Functions

%%%%% Utterance construction and access functions

Synthesis Functions

%%%%% Synthesis specific functions

Debugging and Help

%%%%% backtrace, debugging, advise etc.

Adding new C++ functions to Scheme

Brief decsription of C++ interface.

Regular Expressions

Regular expressions are fundamentally useful in any text processing language. This is also true in Festival's Scheme. The function `string-matches` and a number of other places (notably CART trees) allow th eunse of regular expressions to matche strings.

We will not go into the formal aspects of regular expressions but just give enough discussion to help you use them here. See [regexbook] for probablay more information than you'll ever need.

Each implementation of regex's may be slightly different hence here we will lay out the full syntaxt and semantics of the our regex patterns. This is not an arbitrary selection, when Festival was first developed we use the GNU libg++ `Regex` class but for portability to non-GNU systems we had replace that with our own impelementation based on Henry Spencer regex code (which is at the core of many regex libraries).

In general all character match themselves except for the following which (can) have special interpretations

```
. * + ? [ ] - ( ) | ^ $ \
```

If these are preceded by a backslash then they no longer will have special interpretation.

.

Matches any character.

```
(string-matches "abc" "a.c") => t
(string-matches "acc" "a.c") => t
```

*

Matches zero or more occurrences of the preceding item in the regex

```
(string-matches "aaaac" "a*c") => t
(string-matches "c" "a*c") => t
(string-matches "anythingc" ".*c") => t
(string-matches "canythingatallc" "c.*c") => t
```

+

Matches one or more occurrences of the preceding item in the regex

```
(string-matches "aaaac" "a+c") => t
(string-matches "c" "a*c") => nil
(string-matches "anythingc" ".+c") => t
(string-matches "c" ".+c") => nil
(string-matches "canythingatallc" "c.+c") => t
(string-matches "cc" "c.+c") => nil
```

?

Matches zero or one occurrences of the preceding item. This is it makes the preceding item optional.

```
(string-matches "abc" "ab?c") => t
(string-matches "ac" "ab?c") => t
```

`[ ]`

can defined a set of characters. This can also be used to defined a range. For example `[aeiou]` is and lower case vowel, `[a-z]` is an lower case letter from a thru z. `[a-zA-Z]` is any character upper or lower case.

If the `^` is specifed first it negates the class, thus `[^a-z]` matches anything but a lower case character.

`\( \)`

Allow sections to be formed to allow other operators to affect them. For example the `*` applies to the previous item thus to match zero more occurrences of somethign longer than a single character

```
(string-matches "helloworld" "hello\\(there\\)*world") => t
(string-matches "hellothereworld" "hello\\(there\\)*world") => t
(string-matches "hellotherethereworld" "hello\\(there\\)*world") => t
```

Note that you need two backslashes, one to escape the other backslashes

`\|`

Or operator. Allows choice of two alternatives

```
(string-matches "hellofishworld" "hello\\(fish\\|chips\\)world") => t
(string-matches "hellochipsworld" "hello\\(fish\\|chips\\)world") => t
```

Note that you need two backslashes, one to escape the other backslashes

Some Examples

%%%%% some typical example code usage

Chapter 26. Edinburgh Speech Tools

Details of wagon, ch_wave, ngram etc stuff.

Edinburgh Speech Tools[1]

Notes

1. http://festvox.org/docs/speech_tools-1.2.0/book1.htm

Chapter 27. Machine Learning

decision trees, OLS< SVD, and pointers to Tom's book.

Chapter 28. Resources

In this chapter we will try to list some of the important resources available that you may need when building a voice in Festival. This list cannot be complete and comprehensive but we will to give references to meta-resources as well as direct references to information code, data that may be of use to you.

This document itself will be updated occasionally and it is worth checking to ensure that you have the latest copy.

Updates, new databases, new language support etc will happen intermittently, new voices will be released which may help you develop your own new voices.

http://festvox.org

has been set up as a resource center for voices in Festival offering databases, examples and repository for voice distribution. Checking that site regularly is a good thing to do.

Specifically

http://festvox.org/examples/cmu_us_kal_diphone/

Offers a complete example US English diphone databes as built using the walkthough in Chapter 20. The originally recorded diphone databases is also available as is, at http://festvox.org/databases/cmu_us_kal_diphone/.

http://festvox.org/examples/cmu_time_awb_ldom/

Offers a complete example limited domain synthesis database as build using the walkthroughs in Chapter 5.

Other databases, lexicons etc will be installed on festvox.org as they become available.

There is also a mailing-list `festvox-talk@festvox.org` for discussing aspects of building voices. See http://festvox.org/maillist.html for details of joining it and the archive of messages already sent. Also, while traffic is low, feel free to mail the authors `awb@cs.cmu.edu` or `lenzo@cs.cmu.edu` and we will try to help where we can.

Festival resources

The Festival home page http://www.cstr.ed.ac.uk/projects/festival/ It is updated regularly as new developments happen.

The Festival Speech Synthesis System code and the Edinburgh Speech Tools library and related programs are available from

ftp://ftp.cstr.ed.ac.uk/pub/festival/

or in the US at

http://www.festvox.org/festival/downloads.html

Note that precompiled versions of the system are also available from that site, though at time of writing only Linux binaries are available.

Festival comes with its own manual and html, postscript and GNU info format. It and a less comprehensive Speech Tools manual are pre-built in `festdoc-1.4.1.tar.gz`. The manuals are also available on line at

http://www.festvox.org/docs/manual-1.4.2/festival_toc.html
http://www.festvox.org/docs/speech_tools-1.2.0/book1.htm[10]

You will likely need to reference these manuals often.

It will also be useful to have access to other voices development in Festival as seeing how others solve problems may make things clearer.

In addition to Festival itself a number of other projects throughout the world use Festival and have also released resources. The "*Related Projects*" links give urls to other organizations which you may find useful.

It is worth mentioning Oregon Graduate Institute here who have done a lot of work with the system and release other voices for it (US English and Mexican Spanish). See http://cslu.cse.ogi.edu/tts/ for more details.

A second project worth mention, is the MBROLA project [dutoit96] http://tcts.fpms.ac.be/synthesis/mbrola.html, they offer a waveform synthesis technique [dutoit93] and a number of diphone database for lots of different languages. MBROLA itself doesn't offer a front end, just phone, duration and F0 target to waveform synthesis. (However the do offer a full French TTS system too.) Their diphone databases complement Festival well and a number of projects use MBROLA databases for their waveform synthesis and Festival as the front end. If you lack resources to record and build diphone databases this is a good place to check for existing diphone databases for languages. Most of their databases have some use/distribution restrictions but they usually allow any non-commercial use.

General speech resources

The network is a vast resource of information but it is not always easy to find what you are looking for.

Indexes to speech related information are available. The `comp.speech` frequently asked questions maintain by Andrew Hunt, is an excellent constantly updated list of information and resrouces available for speech recognition and synthesis. It is available in html format from

Australia: http://www.speech.su.oz.au/comp.speech/
UK: http://svr-www.eng.cam.ac.uk/comp.speech/
Japan: http://www.itl.atr.co.jp/comp.speech/
USA: http://www.speech.cs.cmu.edu/comp.speech/

The Linguistics Data Consortium (LDC), although expensive, offers many speech resources including lexicons and databases suitable for synthesis work. There web page is http://www.ldc.upenn.edu A similar organization is the European Language Resources Association http://www.icp.grenet.fr/ELRA/home.html which is based in Europe. Both these home pages have links to other potential resources.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
to be added:
recording and EGG information
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Notes

1. http://festvox.org
2. http://festvox.org/examples/cmu_us_kal_diphone/
3. http://festvox.org/databases/cmu_us_kal_diphone/
4. http://festvox.org/examples/cmu_time_awb_ldom/

5. http://festvox.org/maillist.html
6. http://www.cstr.ed.ac.uk/projects/festival/
7. ftp://ftp.cstr.ed.ac.uk/pub/festival/
8. http://www.festvox.org/festival/downloads.html
9. http://www.festvox.org/docs/manual-1.4.2/festival_toc.html
10. http://www.festvox.org/docs/speech_tools-1.2.0/book1.html
11. http://cslu.cse.ogi.edu/tts/
12. http://tcts.fpms.ac.be/synthesis/mbrola.html
13. http://www.speech.su.oz.au/comp.speech/
14. http://svr-www.eng.cam.ac.uk/comp.speech/
15. http://www.itl.atr.co.jp/comp.speech/
16. http://www.speech.cs.cmu.edu/comp.speech/
17. http://www.ldc.upenn.edu
18. http://www.icp.grenet.fr/ELRA/home.html

Chapter 29. Tools Installation

This chapter describes the installation method for Festival, the Festvox voice building tools and related system as used throughout this book. Binary distributions are included for standard systems, but full source is also provided as are instrction on installation.

These tools are included on the CD as distributed with the book, though all parts (and possible later releases of them) are also available from `http://festvox.org`.

Notes

1. http://festvox.org

Chapter 30. English phone lists

Relating phonemes to sounds is not obvious as people think. Even when one is familar with phone sets its easy to make mistakes when reading lists of phones alone. This is particularly true in reading diphone nonsense words. The table provided here are intended for both the experienced and inexperienced reader of phones, to help you decide on the pronunciation.

These tables are not supposed to be a substitute for a good phonetics course, they are intended to give people a basic idea of the pronunciation of the phone sets used in the particaulr examples in this document. Many simplifying assumptions have been made, and often aren't even mentioned. To the phoneticians out there I apologise, as much as the assumptions are wrong we are here listing atomic discrete phones which we have found useful in building practical systems, even though better sets probably exist.

US phoneset

Inspite of everyone telling you that there is one and only one US phoneset, when it comes to actually using one you quickly discover there are actually many standard one used by lots of different pieces of software, often the difference betwen them is trivial (e.g. case folding) but computers being fundamentally dumb can't take these trivial differences into account. Here we list the *radio* phoneset which is used by standard US voices in festival. The definition is in `festival/lib/radio_phones.scm`. This list was based on those phones that appear in the Boston University FM radio corpus with minor modifications. The list here is exactly those phones which appear in the diphone nonses words as used in the example explained in Chapter 20.

aa
: fAther, wAshington

ae
: fAt, bAd

ah
: bUt, hUsh

ao
: lAWn, dOOr, mAll

aw
: hOW, sOUth, brOWser

ax
: About, cAnoe

ay
: hIde, bIble

eh
: gEt, fEAther

el
: tabLE, usabLE

em
: systEM, communisM

en
: beatEN

er
: fERtile, sEARch, makER

ey
: gAte, Ate

ih
: bIt, shIp

iy
: bEAt, shEEp

ow
: lOne, nOse

oy
: tOY, OYster

uh
: fUll, wOOd

uw
: fOOl, fOOd

b
: Book, aBrupt

ch
: CHart, larCH

d
: Done, baD

dh
: THat, faTHer

f
: Fat, lauGH

g
: Good, biGGer

hh
: Hello, loopHole

jh
: diGit, Jack

k
: Camera, jaCK, Kill

l
: Late, fuLL

m
: Man, gaMe

n
: maN, New

ng
: baNG, sittiNG

p
: Pat, camPer

r
: Reason, caR,

s
: Sit, maSS

sh
: SHip, claSH

t
: Tap, baT

th
: THeatre, baTH

v
: Various, haVe

w
: Water, cobWeb

y
: Yellow, Yacht

z
: Zero, quiZ, boyS

zh
: viSion, caSual

pau
: short silence

In addition to the phone sthemselves the nonsense word generated by the diphone schema also have some other notations to denote different type of phone.

The use of - (hyphen) in the nonsense word itself is used to denot an explicit syllable boundary. Thus `pau t aa n - k aa pau` is used to state that the word should be pronounced as `tan ka` rather than `tank ah`. Where no explicit syllable boundary is given the pronunciation should be pronounce naturally without any boundary (which is probabaly too underspecified in some cases).

The use of _ (underscore) in phone names is used to denote consonant clusters. That is `t_-_r` is the /tr/ as found in `trip` not that in `cat run`.

UK phoneset

This phoneset developed at CSTR a number of years ago is for Southern UK English (RP, "received pronunciation"). Its definition is in `festival/lib/mrpa_phones.scm`.

uh
: cUp, dOne

e
: bEt, chEck

a
: cAt, mAtch

o
: cOttage, hOt

i
: bIt, shIp

u
: pUll, fOOt, bOOk

ii
: bEAt, shEEp

uu
: pOOl, bOOt

oo
: AUthor, cOURt

aa
: ARt, hEARt

@@
: sEARch, bURn

ai
: bIte, mIght, lIke

ei
: Ate, mAIl

oi
: tOY, OYster

au
: sOUth, hOW

ou
: hOle, cOAt

e@
: AIR, bARE, chAIR

i@
: EAR, bEER

u@
: sUre, jUry

@
: About, arlAs, equipmEnt

p
: Pat, camPer

t
: Tap, baT

k
: Camera, jaCK, Kill

b
: Book, aBrupt

d
: Done, baD

g
: Good, biGGer

s
: Sit, maSS

z
: Zero, quiZ, boyS

sh
: SHip, claSH

zh
: viSion, caSual

f
: Fat, lauGH

v
: Various, haVe

th
: THeatre, baTH

dh
: THat, faTHer

ch
: CHart, larCH

jh
: diGit, Jack

h
: Hello, loopHole

m
: Man, gaMe

n
: maN, New

ng
: baNG, sittiNG

l
: Late, bLack

y
: Yellow, Yacht

r
: Reason, caReer,

w
: Water, cobWeb

#
: short silence

In addition to the phone sthemselves the nonsense word generated by the diphone schema also have some other notations to denote different type of phone.

The use of - (hyphen) in the nonsense word itself is used to denot an explicit syllable boundary. Thus `pau t aa n - k aa pau` is used to state that the word should be pronounced as `tan ka` rather than `tank ah`. Where no explicit syllable boundary is given the pronunciation should be pronounce naturally without any boundary (which is probabaly too underspecified in some cases).

The use of _ (underscore) in phone names is used to denote consonant clusters. That is `t_-_r` is the /tr/ as found in `trip` not that in `cat run`.