sprec

A speaker dependent single word speech recognition library

Author:
Martin Pitt
E-Mail: `martin@piware.de`

Supervisor:
Dr.–Ing. J. Helbig
Technical University of Dresden
Faculty Technical Acoustics

20.06.1997

Preface

Speech recognition with computers is a theme that fascinated me for quite a long time. To control a computer in the natural human way – by talking – would be a relief not only to newbies and handicapped people, but also to everybody who has to type long text all the day.

Scientist have worked to equip computers with a hearing sense for human languages for quite a long time. But today's systems are far from being practically usable and have very little in common with the understanding machines in science fiction films. Probably, perfect speech recognition can not be achieved on classic computers.

But nevertheless I wanted to have a little foretaste. Our project assignment in the 11th grade was a welcome opportunity to try it for myself. Of course I did not achieve a major breakthrough, but even for my modest single–word recognizer (up to about 50 words) there are some applications like controlling program interfaces or external devices, e. g. a FischerTechnik™ model, the famous coffee machine, the electric model railway or a universal remote control.

I dealt with different methods of signal analysis and time normalization, programmed an easy–to–use pattern management and some small demonstration programs which illustrate how to use the library.

Martin Pitt
in June 1997

Note: Originally, the library was programmed under DOS. I wrote a quite pretentious graphical shell that supported all features of the library. Now I'm using Linux, so I rewrote it to be portable. It also became much more efficient and easier. This is the reason why I only publish the library without a shell. If you are interested in the DOS version it anyway, feel free to mail me.

Contents

1 Theories and algorithms of speech recognition **3**
1.1 Used symbols . . . 3
1.2 Technical terms . . . 3
1.3 Component diagram . . . 3
1.4 A/D transformation . . . 3
1.5 Signal analysis . . . 4
1.5.1 Zero cross analysis . . . 4
1.5.2 Cepstral coefficient analysis . . . 5
1.6 Recognition of word boundaries . . . 6
1.7 Word patterns . . . 7
1.8 Comparison with reference patterns . . . 7
1.8.1 Linear time adaption . . . 7
1.8.2 Nonlinear time adaption . . . 7
1.8.3 Distance between feature vectors . . . 8

2 Interface of sprec **9**
2.1 Overview . . . 9
2.2 Global declarations . . . 9
2.3 class `Pattern` . . . 9
2.3.1 Type definitions . . . 9
2.3.2 Constructor and destructor . . . 9
2.3.3 Access to properties . . . 10
2.3.4 Feature vector management and access . . . 10
2.3.5 File I/O . . . 10
2.4 class `PatternDB` . . . 11
2.4.1 Purpose . . . 11
2.4.2 Constructors and destructor . . . 11
2.4.3 Access to properties . . . 11
2.4.4 Pattern management . . . 12
2.4.5 Word recognition . . . 12
2.4.6 Saving to a file . . . 12
2.5 class `SampleAnalyzer` . . . 12
2.5.1 Purpose . . . 12
2.5.2 Constructor . . . 13
2.5.3 Methods . . . 13
2.6 class `ZeroCrossAnalyzer` . . . 13
2.7 class `CepstrumAnalyzer` . . . 13

Chapter 1

Theories and algorithms of speech recognition

1.1 Used symbols

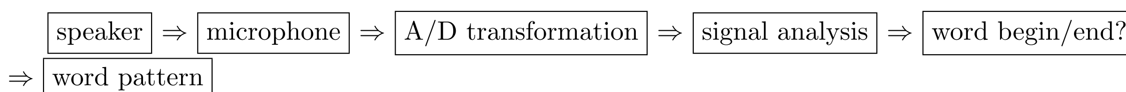
S_n, H_n, B_n: n-th value (sample) of a digital discrete speech signal

1.2 Technical terms

phoneme: spoken sound, smallest language element (vocals, plosives, nasals, ...)
formant: phoneme–typical frequency maximum in frequency spectrum
sample rate: number of samples that are processed every second during playing/recording
sample: single discrete sound volume value
pattern: succession of vectors describing the features of a speech signal
pattern class: group of patterns that describe the same word spoken with different pronunciations

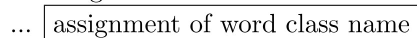
1.3 Component diagram

This diagram illustrates the steps and components of speech recognition. The detailed description of all nontrivial components follows.



recognition mode:

learning mode:



1.4 A/D transformation

The transformation of an analogous speech signal into a digital one is done by a sound card. The speech signal gets divided in small “time windows” (here 256 samples which corresponds to about 23 ms at a sample rate of 11,025 Hz) in which the signal (or precisely, its oscillation) can be supposed to be approximately constant.

1.5 Signal analysis

The purpose of signal analysis is to convert the speech signal into a discrete succession of “feature vectors”, each of them representing and describing a time window. This feature recognition is done to extract relevant information out of the speech signal and to sort out unnecessary data. A direct saving of the speech signal would waste too much memory and CPU time. On the other hand, many pieces of information of the speech signal like the development of the base frequency, volume and redundancy (about 60% in English language) are not necessary for speech recognition. Moreover, the speech signal depends on many factors like the speaker’s mood and utterance speed; so the speech signal is no usable, general representation of a word.

I dealt with two different types of signal analysis: zero cross analysis and cepstral coefficients.

1.5.1 Zero cross analysis

An easy and very fast, but quite inaccurate algorithm is the zero cross analysis. I got its principle from the documentation of an older speech recognizer (see source reference).

All zero passes over the time window (i. e. spots where the speech signal is zero) are searched. The distance between two such points indicates the wavelength and with it the frequency (see figure 1.1).

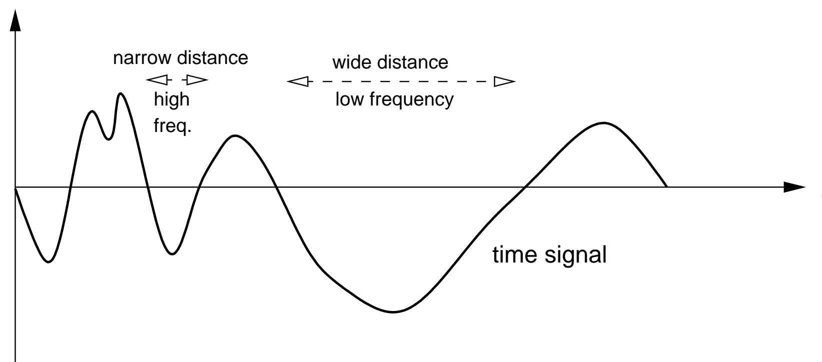


Figure 1.1: principle of zero cross analysis

However, to gain enough precision a high and low pass filter have to be applied to the speech signal before since quiet high frequencies are “covered” by loud low ones and thus not generate a zero crossing. After the high pass filter high frequencies can be registered, too. Figure 1.2 demonstrates this.

A simple high pass filter is differentiation of the speech signal. Wide wavelengths have virtually no effect on the result, however wavelengths of high frequencies remain nearly constant:

$$H = \frac{d}{dt} S \quad \Rightarrow \quad H_n = S_{n+1} - S_n \;\; \text{(for discrete values } dt = 1\text{)}$$

A band pass is used instead of a low pass because it is more precise. A bandwidth of 1 kHz around a center of 500 Hz is used.

$$B_n = a\, S_n + b\, B_{n-1} + c\, B_{n-2} \qquad \text{(for the first two coefficients: } B_k = 0 \;\; \forall\, k < 0 \;\text{)}$$

The band pass coefficients are calculated as follows:

$$a = 1 - b - c; \qquad b = 2e^{\frac{-\pi B}{F}} \cdot \cos\left(\frac{2\pi f_c}{F}\right); \qquad c = -e^{\frac{-2\pi B}{F}}$$

F is the sample rate, f_c the center frequency and B the bandwidth. I use the values $F =$ 11025 Hz, $f_c = 500$ Hz and $B = 1000$ Hz.

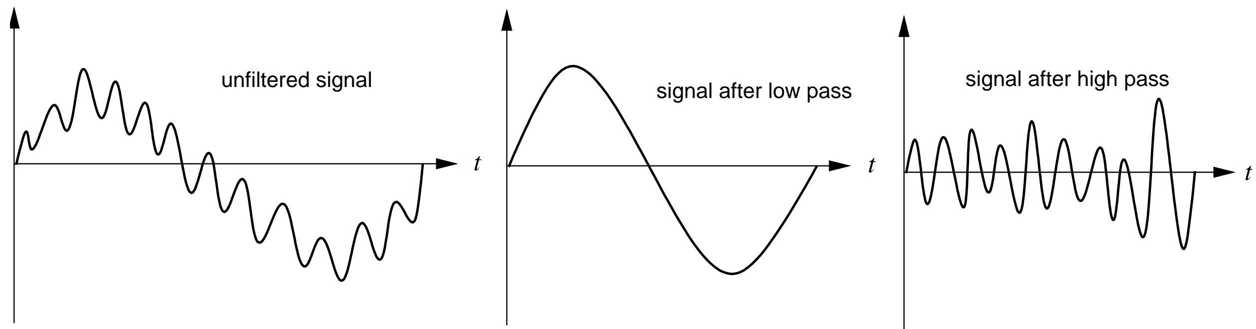


Figure 1.2: high frequencies are "covered" by low ones, so filtering is necessary

Although being quite precise, this band pass unfortunately cannot be used as high pass because it cannot damp low frequencies acceptably well. But as low pass it proved quite suitable.

That is the whole preparation. Now, all zero crossing distances of the time window are determined and assigned to one of eight frequency ranges. Thus a feature vector consists of eight numbers indicating which kind of frequencies are present in the time window. This simple frequency bank analysis can be calculated so fast that it even can be used on an ancient XT computer, but it proved successful in both my and other speech recognition systems.

I used the following frequency ranges: 0–200, 200–400, 400–700, 700–1000, 1000–1800, 1800–2600 and 2600–5000 Hz. This roughly corresponds to the areas of the human formants.

According to word length an adequate number of feature vectors are put together, resulting in a word pattern.

1.5.2 Cepstral coefficient analysis

First, a spectral analysis is applied on the time window. The resulting complex spectrum is normalized by calculating its magnitude. Due to this process, only n/2 of n coefficients are relevant since the following ones are just mirroring the first half spectrum and, since the sample consists of discrete values, frequencies higher than half of the sample frequency cannot appear anyway. They are just set to 0 to ease calculation.

The k-th coefficient of a DFT (**D**iscrete **F**ourier **T**ransformation) for a time window containing N samples is calculated as follows:

$$D_k = \frac{1}{N} \sum_{n=0}^{N-1} s_n \cdot e^{\frac{2\pi i n k}{N}} \qquad (i\text{: imaginary unit})$$

This formula is of little practical use because $e^{i\,\varphi}$ cannot be calculated directly. But since only the magnitude of the coefficient is of interest, the formula can be transformed to:

$$e^{i\,\varphi} = \cos\varphi + i \sin\varphi \quad \Rightarrow$$

$$D_k = \frac{1}{N} \left[\sum_{n=0}^{N-1} s_n \cos\left(\frac{2\pi n k}{N}\right) + i \sum_{n=0}^{N-1} s_n \sin\left(\frac{2\pi n k}{N}\right) \right]$$

$$|D_k| = \frac{1}{N} \sqrt{\left[\sum_{n=0}^{N-1} s_n \cos\left(\frac{2\pi n k}{N}\right) \right]^2 + \left[\sum_{n=0}^{N-1} s_n \sin\left(\frac{2\pi n k}{N}\right) \right]^2}$$

This formula is used to calculate the necessary N/2 spectral coefficients. Nevertheless, this algorithm is very CPU intensive (n^2 summations must be made for n coefficients). There is

a modified algorithm called FFT (**F**ast **F**ourier **T**ransformation) that allows carrying out the spectrum calculation considerably faster: the sample values are segregated into an odd and an even sequence (i. e. the first sequence contains the 1st, 3rd, 5th, ... sample and the second contains the 2nd, 4th, 6th, ... sample). The FFTs of both sequences are calculated and their results are combined into one (the actual spectral coefficient) using another formula I unfortunately do not know. So the number of summations reduces to $\frac{1}{2}n^2$. But that is just the start: the algorithm can be applied recursively, e. g. the partial sequences can be split in parts again. If the number of sample values in the time window is a power of 2, this sequence splitting can be done $\log_2 n$ times; so the number of summations can be reduced to $n \cdot \log_2 n$ which is a major saving compared to the initial n^2.

The so–called "cepstrum" (this word is formed by scrambling the word "spectrum") is now built from the spectrum by logarithmizing the spectrum and doing an inverse DFT/FFT. The formula of the inverse DFT differs from the DFT formula only by the absence of the factor $\frac{1}{N}$, so the same FFT routine can be used. The first 10 to 16 cepstral coefficients are used as feature vector. They indicate the magnitude of the highest "quefrencies" which reflect the formant structure.

1.6 Recognition of word boundaries

The spotting of word boundaries can be done manually or automatically.

The manual method is by far the simplest and also safest method. While speaking, the user has to press a key. The disadvantage of this method is, of course, that it is relatively inconvenient and the key pressing must match the actual spoken word quite precisely. Otherwise there are either long phases of silence (although they can be compensated quite well by the nonlinear time adaption), or, even worse, parts of the word get cut off.

An automatic word boundary detection is no easy task to carry out with simple means. For a really sophisticated and usable one a good microphone and quite much CPU power is needed and an advanced algorithm must be used. Nevertheless I dealt with quite an easy one, but more for experimental purposes.

A good indicator whether or not a word is spoken at the moment is the signal energy

$$E = \sum_{k=0}^{N-1} S_k^2$$

As long a the energy of the current time window exceeds a threshold a word is spoken and gets analyzed.

However, checking only the signal energy is not sufficient. Sounds like noise and friction phonemes (s, z, f, h, ...) or plosives (b, p, d, t, ...) that produce only a small signal energy would be cut off at the beginning and end of the word and within it they would be misinterpreted as word end. The latter problem cat be avoided by a pause threshold that defines how long zones of low signal energy may be.

To detect such phonemes at the word boundaries, the zero crossing rate is used:

$$R_0 = \sum_{k=0}^{N-2} \mathrm{sgn}(S_k \cdot S_{k+1}) \; ; \qquad \mathrm{sgn}(x) = \begin{cases} 1 & \text{for } x < 0 \\ 0 & \text{for } x \geq 0 \end{cases}$$

If the rate exceeds a threshold a provisional word begin is set. If the signal energy also exceeds its threshold later the word is accepted; otherwise the word begin is canceled.

The same strategy is followed at the end of the word: after the signal energy goes below its threshold, recording continues until the zero crossing rate also falls below the minimum.

Another threshold can be defined that indicates the minimal signal energy the zero crossing rate is tested at. This prevents looking upon low–energy background noise as word begin. This threshold should be very low to remain able to detect phonemes like 'f' or 's' as word start.

This procedure allows detecting word boundaries with a reasonable reliability. But it is quite susceptible for continuous background noise (that already produces quite a high zero crossing rate) and short, loud noises like keyboard strokes, knocks, etc. The latter can be often, but hardly completely, avoided by a minimal word length.

In learning mode it is strongly recommended to use the manual mode. The thresholds for the automatic mode have to be adjusted according to background noise level and even more to the used sound card and microphone.

This point requires enhancements urgently. The manual method is uncomfortable and requires some sure instinct to handle and the automatic procedure presented here is more a gamble than a reliable robust solution.

1.7 Word patterns

In recognition mode the resulting word pattern is compared against the reference patterns (see next section). In learning mode the new pattern is added to the reference pattern database (after assigning a proper name). In learning mode it is advisable to teach several samples of the same word (depending on similarity to other word classes about five to ten) to make the recognition of different pronunciation variants safer.

1.8 Comparison with reference patterns

A recently recorded word has to be matched to a word class from the reference pattern database. To do this it has to be compared to all reference patterns, i. e. the distances to all of them have to be determined. The reference pattern having the smallest distance to the recorded word is considered as the recognized one. But if even the smallest distance is still bigger than a refuse threshold, the word is rejected as not recognized.

The smaller the refuse threshold, the better gets the correct–hit rate, but the more words are refused. It has to be adjusted according to the particular application, but also to the extent of the vocabulary (i. e. the number of word classes).

To be able to compare two patterns, their lengths have to be equalized first. This procedure is called "time adaption". Two different strategies are presented below.

1.8.1 Linear time adaption

This method just stretches, respective shrinks the pattern in a linear way to a size that matches the other. This can be done very quickly but does not lead to usable results for speech recognition because single word parts may be uttered at different speeds and pauses at the word boundaries may also be different. Therefore I implemented the nonlinear time adaption (also called "dynamic programming") that does not have these shortcomings.

1.8.2 Nonlinear time adaption

The goal of this algorithm is to determine a "path" through the word pattern along which the distances of the feature vectors get minimal. Thus, two or even more time windows of the word pattern can be projected to one window of the reference pattern (which means that the word has been stretched at this point) and also the other way round. No time window must be jumped over though, and the way must not run "backwards".

For a better visualization, the patterns are displayed in a diagram with the test pattern extending along the x–axis and the reference pattern going upwards (see figure 1.3). Hence, to get from one point to the next there are three possibilities: straight upwards (which means projecting two reference pattern windows to one of the test pattern), straight to the right (which means just the opposite) or diagonally up right (which means that the speed of both patterns are equal at this point). Finding this way is not very difficult if a recursive algorithm is applied. However,

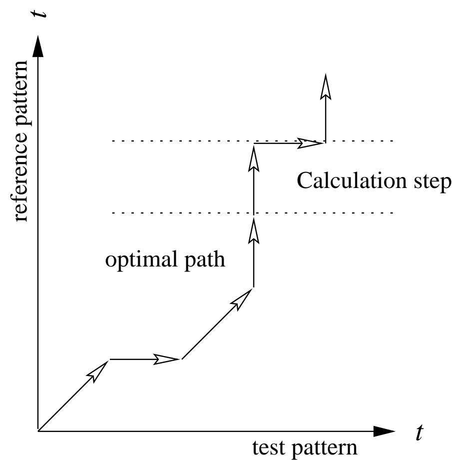


Figure 1.3: Visualisation and possible path

since there are already millions of different paths even for quite small patterns, this would last far too long.

But the precise knowledge of the path's course is actually not necessary. The only thing of interest is the (minimal) distance between the patterns, so a trick can be applied: In every step (towards the end of the reference pattern) the current minimal distance up to this point is calculated for every point of the test pattern. It is saved cumulatively in a field. To get from one step to the next, the three feature vector distances (for every possible direction) to the current reference pattern vector are calculated for every test pattern vector. The smallest of the three distances is added to the current minimal distance array. This is done until the last reference pattern vector is reached.

As a consequence you don't need to calculate every single path since the ones that cannot be the optimal one are sorted out as early as possible. Additionally, an iterative algorithm can be used. This saves a lot of calculation time.

1.8.3 Distance between feature vectors

There are several ways to define a distance between two feature vectors $\vec{f}$ and $\vec{g}$. The most obvious one may be

$$D(\vec{f},\vec{g}) = \sum_{i=1}^{n} |f_i - g_i| \; ; \qquad a_i : \text{ i-th element of an n–dimensional vector } \vec{a}$$

which means the sum of the vector element differences. This works quite well. But the modified form

$$D(\vec{f},\vec{g}) = \sum_{i=1}^{n} (f_i - g_i)^2$$

is more adequate for a purpose like speech recognition since it amplifies big distances and tolerates small ones (which appear almost every time).

Chapter 2

Interface of sprec

2.1 Overview

The library was designed to be as portable as possible. It is written in ANSI C++ and does not use any compiler or platform specific features. It just requires an ANSI C++ compliant compiler and the classes `vector` and `string` of the STL[1].

The library (its header file `sprec.h` in particular) declares classes for managing a pattern database (for handling trained reference patterns) and for signal analysis.

2.2 Global declarations

Beside the classes there are a few global declarations:

- `enum PDBIOResult` is an error code returned by the file operations of the pattern database. The meaning of its possible values are documented in `sprec.h` and in a later section of this document.
- `PATREFUSED` is a constant that may be returned when recognizing a word. It indicates that the pattern to be recognized was refused since even the distance to the closest reference pattern exceeded the refuse threshold.

2.3 class `Pattern`

2.3.1 Type definitions

`Vector` represents a feature vector. It is an array of `Element`s (which are `char`s by now).

2.3.2 Constructor and destructor

`Pattern`'s constructor has the following form:

```
Pattern (unsigned _vectorSize, const string& _name = "");
```

It constructs a new pattern object with the given vector size (depends on the used signal analysis type) and name. It is not necessary that the name is unique. It even may be desirable to have several patterns with the same name if different pronunciation variants of the same word are trained (and this indeed should be done!). The constructor does not yet allocate memory for feature vectors (initially the object contains 0 vectors). This must be done afterwards by calling `NewData()`.

[1] ANSI C++ Standard Template Library

The destructor destroys the pattern and the allocated memory for the pattern vectors (if existing).

2.3.3 Access to properties

- `string Name()` returns the name of the pattern.
- `void SetName (const string&)` changes the pattern's name.
- `unsigned VectorSize()` returns the size (dimension) of the feature vectors of this pattern. It depends on the used signal analysis type and was set in the constructor.
- `unsigned nVectors()` returns the number of vectors this pattern consists of.
- `unsigned DataLength()` returns the size of allocated memory for the pattern data. Practically it is the vector size times the number of vectors.

A pattern has two additional reserved fields of type `long int` which can be used to store any additional data an application may need. These fields are not modified by sprec but by `PatternDB::SetPatReserved ()` (see below). They can be modified and retrieved by:

- `void SetReserved1 (long)`
- `void SetReserved2 (long)`
- `long Reserved1 ()`
- `long Reserved2 ()`

2.3.4 Feature vector management and access

- `void NewData (unsigned dataLen)` has to be called every time the size (not the content) of the pattern should be changed (this is also required directly after the constructor call). It allocates pattern memory for `dataLen Element`s (not vectors!). If there is already pattern data it is deleted first.
- If a single Element of a particular vector should be read, the `()` operator can be used very comfortably. Its two parameters are the number of the desired vector and the number of the element of this vector. E. g. to read the 3rd element of the 4th vector the following code is used:

  ```
  Element e = myPattern (4, 3);
  ```

  Attention: the parameters have to be valid! The method does not check them.
- `Vector AccessVector (unsigned vector)` must be used if an Element should be modified or a complete vector (Element array) is to be accessed. The returned `Vector` can be used like an array. Attention must be payed when using this method: it does not check whether the given vector number does really exist and the returned vector is just a C pointer and therefore must be handled with care.

2.3.5 File I/O

- `PDBIOResult Load (FILE*)`
- `PDBIOResult Save (FILE*)`

These two methods load, respectively save the pattern in the given file. They return an error code indicating success (`PDB_OK`) or the reason of failure. When loading, this may be `PDB_EOF` which means that the file ended unexpectedly (there is no complete pattern in it). When saving, `PDB_DISKFULL` indicates that the pattern could not be written completely because the file cannot grow any more (the storing device got full).

Normally, these methods are not called by the user, but only by the pattern database.

2.4 class `PatternDB`

2.4.1 Purpose

The class `PatternDB` manages a list of patterns (objects of class `Pattern`) which form a reference pattern database. The list is sorted alphabetically after the names of the patterns. It provides loading and saving of the patterns and has also methods for calculating the distance between two patterns and for finding the closest match to a given test pattern (the actual word recognition).

`PatternDB` has three additional properties: a database name, a refuse threshold and an array of 16 reserved bytes which can be used freely to store any additional data an application may need.

2.4.2 Constructors and destructor

- `PatternDB (const string&)` initializes a new pattern database with the given name. Initially it contains no patterns. The recognition threshold is set to 120 (this value should be adapted to suit the application's needs and also the used signal analysis type) and the reserved field is cleared to zero.
- `PatternDB (const string& fName, PDBIOResult&)` loads a pattern database from the file with the given name. It reads the database name, the refuse threshold, the reserved fields and all patterns the database contains. An error code is returned through the `PDBIOResult` variable. When the constructor finished it can have the following values:
  - `PDB_OK`: The loading was successful.
  - `PDB_NOTFOUND`: A file with the given name does not exist.
  - `PDB_NOPDB`: The file is no pattern database or is damaged.
  - `PDB_EOF`: The file ended unexpectedly before all patterns were read. This means that the file is damaged.

`PatternDB`'s destructor destroys all patterns that are stored in it.

2.4.3 Access to properties

- `string Name()` returns the name of the pattern database.
- `void SetName (const string&)` changes the pattern database's name.
- `unsigned RecogThreshold ()` retrieves the refuse threshold.
- `void SetRecogThreshold (unsigned)` modifies the refuse threshold.
- `void* Reserved ()` provides access to the reserved field. As already stated, the returned pointer points to an array of 16 bytes!

2.4.4 Pattern management

- `unsigned Add (Pattern*)` inserts the given pattern into the database at the proper lexical position. Responsibility is taken over for the object, i. e. it gets destroyed in `PatternDB`'s destructor and must not be shared between several databases. Directly after the pattern is inserted, the protected method `SetPatReserved (Pattern*, unsigned)` is called which should be overridden in derived classes if the reserved fields of the patterns are used and are to be set after inserting. `PatternDB`'s implementation just sets them to zero. The passed parameters are a pointer to the new pattern and the assigned index. This index is also returned by `Add`.
- `void Del (const string&)`, `void Del (Pattern*)`, `void Del (unsigned)`: If a pattern should be deleted, it can be identified by three different references: its name, object pointer or its position (index) in the database. Hence there are three overloaded versions of the deletion method.
- `unsigned nPatterns ()` returns the number of patterns stored currently in the database.
- The `[]` operator is used to access the patterns (it returns a `Pattern&` reference).

2.4.5 Word recognition

- `unsigned Distance (Pattern&, unsigned idx)` calculates the distance between the given pattern and the one in the database with index idx. It performs a nonlinear time adaption (see section 1.8.2) to find the smallest distance.
- `unsigned Recog (Pattern&, unsigned *diffs = NULL)` does the actual recognition process. The given pattern is compared with all patterns in the database. The one having the smallest distance to the test pattern is considered as the recognized one and its index is returned. If, however, its distance exceeds the refuse threshold, the constant `PATREFUSED` is returned instead. If the `diffs` pointer is not `NULL`, the distances to every pattern of the database are written into it (so it must be assured that the array is big enough). This may be useful for the training process and also for adjusting thresholds.

2.4.6 Saving to a file

- `PDBIOResult Save (const string&)` writes the whole database into a file of the specified name. It returns an error code which can have the following values:
  - `PDB_OK`: The saving was successful.
  - `PDB_CREATE`: The file could not be created; the device might be read–only.
  - `PDB_DISKFULL`: The file cannot grow any more and the data was not completely written; the device might be full.

2.5 class `SampleAnalyzer`

2.5.1 Purpose

To support different methods of sample analysis with a common interface the abstract base class `SampleAnalyzer` has been designed.

2.5.2 Constructor

```
SampleAnalyzer (unsigned _vectorSize, unsigned _windowSize = 256,
                unsigned _sampleRate = 11025);
```

The first parameter, `_vectorSize`, determines the number of elements of the feature vectors this sample analysis method produces. The constructors of derived classes should set this parameter.

The "width" of the time windows (i. e. the number of sample values it contains) is controlled by `_windowSize`. It should be appropriate to the analysis method, the sample rate (which is given by the third parameter `_sampleRate`) and the desired resolution of the signal analysis (which may affect the quality of recognition).

2.5.3 Methods

- `unsigned VectorSize()`
- `unsigned WindowSize()`
- `unsigned SampleRate()`

These three property accessors return the values set by the constructor.

`static unsigned Energy (Sample, unsigned size)` calculates the energy of the given sample.

The pure–virtual method `virtual void AnalyzeWindow (Sample, Pattern::Vector)` must calculate a feature vector from the given time window. The sizes of the time window and the feature vector were specified in the constructor. This is the only method that must be implemented by derived "real" sample analyzer classes.

`Pattern* Analyze (Sample, unsigned size)` analyzes a complete sample by dividing it into time windows and applying `AnalyzeWindow()`. It creates a new `Pattern` object that is returned.

2.6 class `ZeroCrossAnalyzer`

This derivative of `SampleAnalyzer` implements the zero–cross sample analysis described in subsection 1.5.1.

Its constructor, `ZeroCrossAnalyzer (unsigned _windowSize = 256, unsigned _sampleRate = 11025)` forwards the parameters `_windowSize` and `_sampleRate` to the `SampleAnalyzer` constructor and sets the feature vector size to 8.

2.7 class `CepstrumAnalyzer`

This derivative of `SampleAnalyzer` implements the cepstrum signal analysis described in subsection 1.5.2.

Its constructor, `CepstrumAnalyzer (unsigned _windowSize = 256, unsigned _sampleRate = 11025)` forwards the parameters `_windowSize` and `_sampleRate` to the `SampleAnalyzer` constructor and sets the feature vector size to 16.