TR-IT-0286

Scalar Quantization of Cepstral Parameters for Low Bandwidth Client-Server Speech Recognition Systems

グルーン ライナー
Rainer Gruhn

シンガー ハラルド
Harald Singer

1998.12

We describe a simple but highly efficient approach to solve the bandwidth problem for a client-server architecture speech recognition system. Data transmission from client to server follows an approach proposed by SRI, i.e. scalar quantization of cepstral parameters is used for compression. Recognition results on ATR's Travel Arrangement task show that recognition error only increases by about 0.5 % at a bandwidth of less than 6 Kb/s. This report is accessible for ITL members via `/home/singer/tex/TR-IT-Compress/`.

目次

1 Introduction

ATR's speech recognition and translation system ATR-MATRIX [6] is a research prototype researchers all over the world would like to see. However, to demonstrate it, the complete equipment, including voluminous computers, has to be shipped, thus making presentations expensive and inconvenient. Use of laptops is no suitable solution because of their relative high prices and low computational power.

Our proposal is to use a client-server system, where the speech is recorded using a small computer system at the place of the demonstration and is then transfered to a powerful server, e.g. at ATR. Unfortunately, the communication channel between remote client and server is usually very narrow. High quality speech data is by far too big to transfer it by a modem over a standard telephone line. One second of speech (assuming 16 Bit and 16 kHz) sums up to 256 kBit, which is by far more than a normal 28.8 kBaud modem could transmit. The Internet is usually too busy and too unpredictable to use it to transfer large amount of data, especially for the purpose of real time computing.

At ICASSP 1998, SRI [1] and IBM [5] proposed approaches to divide a speech recognition system into preprocessing performed on the client computer and decoding, which is performed at the server. The preprocessed data is compressed before transmission and decompressed on arrival.

Of these two papers, the SRI proposal seems to be faster and easier to implement. The client does the feature extraction, which does not require powerful computers. The resulting cepstral vectors are compressed using scalar quantization. In total, the data that has to be transmitted is reduced to about 4-6 Kbps, depending on the codebook size.

In section 2 scalar compression, codebook training and compression and decompression steps are described. Section 3 explains experiments measuring data deterioration, parameter selection verification and recognition accuracy. In section 4 we give directions for future research, both for improving the algorithm and possible applications. The appendix contains additional information such as a manual for the implemented software, additional detailed experimental results and description of configuration files for the speech recognition experiments.

2 Theory of Scalar Quantization

Scalar quantization is a simple way to reduce bandwidth of a data stream by assigning discrete labels to continuous data points or vectors. The comprehensive and general theory can be studied in [2], this section will only describe the necessary basics for the present application.

Scalar quantization basically means to reduce continuous data values to integer codeword numbers. Compared to vector quantization, scalar quantization is easier and faster to compute. For example, given a vector with 13 floating point values, each of the 13 data items in the vector is quantized independently, not the vector as a whole. The principle of scalar quantization is simple: the data values are distributed with mean μ and variance σ. Using mean and variance, the actual value is projected into a discrete space. The data points in this discrete value space are called bins.

To calculate the index value, which is the bin corresponding to a data value x, we use the equation

$$index = \lfloor A\frac{x-\mu}{\sigma^2} + \frac{B}{2} + \frac{1}{2} \rfloor, \qquad 0 \leq index < B \tag{1}$$

with A being a scale factor and B the total number of bins.

If x is the mean value, the index will be exactly $\frac{B}{2}$. The scale factor is necessary to prevent all data values to be projected to the same bin (at $\frac{B}{2}$) and helps gaining additional accuracy. As an example, the scale factor could be set to 1000 and the number of bins to 32000. For rounding reasons, $\frac{1}{2}$ is added.

Many bin values will rarely or not occur. To store the index more efficiently, the bins are combined to groups, the C_i in Figure 1. The grouping is done by assigning an equal data amount to every group. The indices i of the groups are the codeword numbers that will be transmitted. Figure 1 shows a possible distribution of index values and an adequate merging to codewords dependent on N, the amount of data per bin.

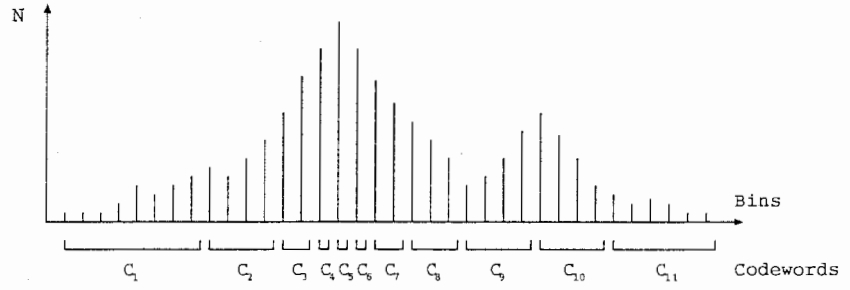

図 1: Binning of data and quantization

2.1 Codebook

To encode and decode the data, codebooks are needed. The information they contain is dependent on their purpose:

- For encoding the following information is needed:
  - scale factor A
  - number of bins B
  - mean
  - variance
  - end bin for each codeword
- For decoding, a table listing the original values corresponding to the bins, one per codeword, is sufficient.

Furthermore, the codebook size has to be given in both cases, and the list of bins and the list of values must be sorted similarly.

(2.1.1) Building a Codebook

To build a codebook, the training data has to be read twice. The first time, mean, variance and the total amount of data are calculated. The second time a bin table similar to Figure 1 is set up using equation 1 for each parameter in the data vector. The last step is to divide the amount of data by the codebook size to get divisions of equal size and find out the "border bins" where this data amount is reached. These bins (and the corresponding values) are saved to disk together with the other needed data as a codebook header. See 付録 A for instructions how to use the codebook training software.

(2.1.2) Codebook Size

In this project, our main goal is to reduce the necessary bandwidth. In other words, the codebook size and thus the number of bits needed to transfer a codebook number should be chosen efficiently. Some parameters in the data vector will have a higher variance than others. For optimal representation of the original data it is reasonable to use larger codebooks for high variance parameters and accept smaller codebooks for more constant ones.

Given a total number of bits available for transmitting the codewords for all parameters in a data vector, the optimal bit distribution can be calculated using the "greedy search" algorithm proposed in [2, page 225]. This algorithm assigns bits to the i quantizers by stepwise giving one bit to the quantizer with highest demand w_i. The demand w_i is a measure for the neediness of another bit for quantizer i. A simplified equation for the demand is

$$w_i = \sigma_i^2 2^{-2b_i} \tag{2}$$

with σ_i being the variance and b_i the number of bits already assigned to quantizer i.

The algorithm to distribute B bits among i quantizers used in this project is slightly different from the algorithm in [2, page 234]. It consists of four steps:

Step 0 Initialize the bit allocations $b_i = 0$ and the demands $w_i = 0$ for each quantizer i.

Step 1 Find the index i with the maximum demand w_i.

Step 2 Set $b_i = b_i + 1$ and recalculate w_k using equation 2.

Step 3 While $\sum b_i \leq B - 1$ go to Step 1. Otherwise stop.

The library function implementing this algorithm is described in Appendix B.2.

As an example, we target a transmission rate of 4.8 kbps, i.e. 6 byte or 48 bit for a 10 msec frame of speech. After subtracting 4 bits for the data header, we can distribute 44 bits to the 13 feature parameters. The algorithm calculates the bit distribution 6 5 4 4 4 3 3 3 3 3 2 2 2, allocating many bits to the energy and the first cepstral parameters. Section 3.2 shows the distortion resulting in this choice.

2.2 Data Transmission

Transmitting an integer value for every codeword would be an enormous waste of bandwidth. The transmission of the compressed data is thus done by storing the codewords in a bitvector. Figure 2 shows an example how a bitvector may look like. The first bits are used for the data header, which is needed to distinguish data from control information, like start-of-sentence (we are using ATRSPREC's FrameSync format). Following are the codewords as binary numbers, shifted to positions that optimally use the available bits. A codeword may be spread over two bytes.

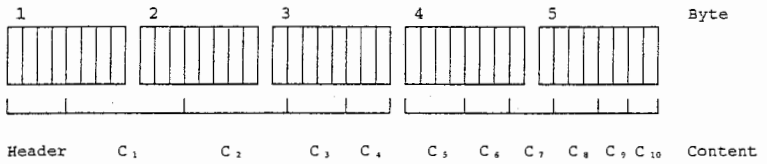

図 2: An example for the bitvector written by `CB_writeEncode_data`: the first 4 bits contain the FrameSync header, the codewords are stored in the following bits. Usually the first parameters have the highest variance and thus the largest codeword size.

3 Experiments

3.1 Database

As training data for the codebook, we use 407 conversation sides with a total of about 8 hours of speech taken from ATR's travel arrangement task described in[4]. The speech test set consists of 42 conversations sides comprising 551 utterances by 17 male and 25 female speakers, totalling 40 minutes speech from the same database.

3.2 Data Deterioration by Lossy Compression

Signal compression by scalar quantization is an efficient way of data reduction, for the price of losing precision. The distortion becomes clearly visible in a cepstrogram, i.e. a spectrogram calculated as FFT of the Mel-Fourier Cepstrum Coefficients (MFCC). Figure 3 shows cepstral parameters for an example signal, Figure 4 shows the same cepstral parameters after compression and decompression. The example signal is a female speaker saying "tsuing de onegai shimasu". The graphics show that compression does not change the overall formant structure.

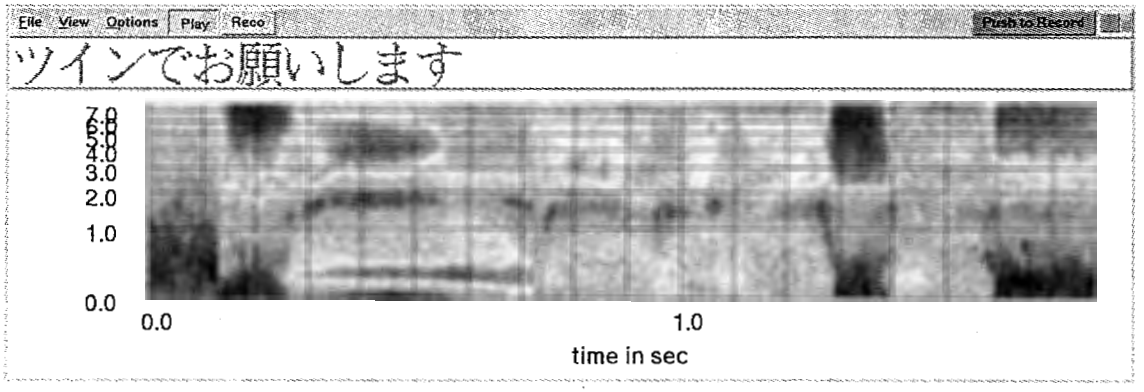

図 3: Cepstrogram (spectrogram of cepstral parameters) without compression

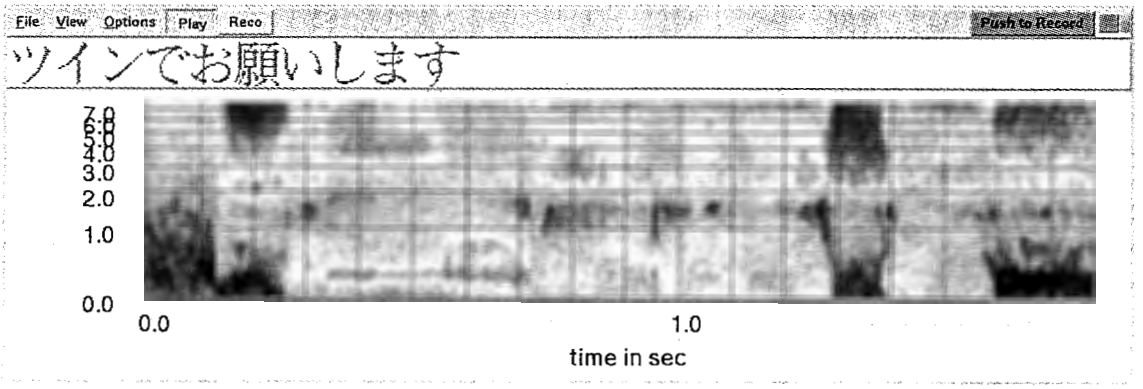

図 4: Cepstrogram after compression and decompression

The following tables show a numeric example of how much the data is deteriorated by compression. The examples are calculated using 8 hours of speech data for codebook training (407 conversation sides) and 40 minutes for error estimation (42 conversation sides). Table 1 lists mean squared error values, Table 8 shows the percental mean deviation. The bold written numbers in the tables indicate the bit distribution used in all experiments (see section (2.1.2) for details).

The mean squared error values are calculated using

$$\epsilon = \frac{1}{n}\sum_{i=1}^{n}(x_i - x_i^*)^2 \tag{3}$$

The equation for the percental mean deviation is

$$\Delta = \frac{1}{n}\sum_{i=1}^{n} \| \frac{x_i - x_i^*}{x_i} \| \tag{4}$$

with x_i being the ith data value and x_i^* the according compressed and decompressed value.

Several unexpected jumps in the percental mean deviation (Table 8) let this distortion measure appear unreliable.

表 1: Example tables for mean squared error: Upper table lists errors with the upper border bin as codevalue, bottom table lists errors if the mean between upper and lower border bin is used.

bits	energy	cep 1	cep 2	cep 3	cep 4	cep 5	cep 6	cep 7	cep 8	cep 9	cep 10	cep 11	cep 12
1	2.8930	1.3964	1.3620	1.4574	0.6552	0.4833	0.2897	0.3556	0.3095	0.4509	0.2484	0.1920	0.0980
2	0.9691	0.4477	0.3790	0.4482	0.2667	0.2355	0.1375	0.1367	0.1024	0.1277	**0.0865**	**0.0733**	**0.0413**
3	0.3166	0.1641	0.1241	0.1662	0.0813	**0.0903**	**0.0458**	**0.0464**	**0.0331**	**0.0378**	0.0322	0.0244	0.0169
4	0.1280	0.0555	**0.0456**	**0.0624**	**0.0243**	0.0243	0.0143	0.0131	0.0100	0.0114	0.0115	0.0076	0.0068
5	0.0670	**0.0196**	0.0169	0.0245	0.0087	0.0084	0.0051	0.0043	0.0037	0.0043	0.0050	0.0029	0.0027
6	**0.0397**	0.0061	0.0049	0.0076	0.0028	0.0021	0.0015	0.0009	0.0011	0.0010	0.0016	0.0008	0.0011
7	0.0311	0.0021	0.0018	0.0029	0.0011	0.0008	0.0006	0.0003	0.0004	0.0004	0.0007	0.0003	0.0004
8	0.0264	0.0007	0.0006	0.0010	0.0004	0.0003	0.0002	0.0001	0.0001	0.0001	0.0003	0.0001	0.0002
9	0.0240	0.0001	0.0001	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

bits	energy	cep 1	cep 2	cep 3	cep 4	cep 5	cep 6	cep 7	cep 8	cep 9	cep 10	cep 11	cep 12
1	1.3120	0.3942	0.3315	0.3119	0.1771	0.1087	0.0668	0.0835	0.0705	0.1025	0.0572	0.0445	0.0224
2	0.2649	0.1405	0.0986	0.0966	0.0795	0.0534	0.0325	0.0341	0.0250	0.0307	**0.0211**	**0.0177**	**0.0097**
3	0.1096	0.0675	0.0315	0.0350	0.0269	**0.0207**	**0.0112**	**0.0121**	**0.0085**	**0.0095**	0.0080	0.0060	0.0040
4	0.0596	0.0231	**0.0112**	**0.0131**	**0.0090**	0.0055	0.0035	0.0037	0.0027	0.0030	0.0028	0.0019	0.0016
5	0.0408	**0.0061**	0.0041	0.0052	0.0034	0.0019	0.0012	0.0013	0.0010	0.0011	0.0012	0.0007	0.0006
6	**0.0323**	0.0018	0.0012	0.0017	0.0012	0.0004	0.0004	0.0003	0.0003	0.0003	0.0003	0.0002	0.0002
7	0.0285	0.0006	0.0004	0.0006	0.0005	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0000	0.0001
8	0.0254	0.0002	0.0001	0.0002	0.0002	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
9	0.0239	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

3.3 Parameter Selection Verification

Several parameters are used in codebook training that were set based on preliminary experiments. Finding out optimal settings is a possible yet very time consuming task. Because of this, we only do a few checks to see if considerable improvements are possible.

In equation 1, a scale factor A appears. A was always assumed to be 1024. If A is too small, all values will be around the mean, so that the codebook is inefficient. If A is too big, many values will be projected to the upper or bottom border bin, so that information is lost.

The experiments were done using scale factors of 512, 2048, 4096 and 8192. The resulting mean square error tables can be found in appendix C.2. Comparison with Table 1 shows that increasing the scale factor to 2048 or 4096 significantly improves the error rates for the high cepstral coefficients while keeping the error rates for energy and the first coefficients at the same level. This suggests to set A to 4096 for further experiments. Even higher scale factors A would lead to information loss.

3.4 Bit Distribution

The algorithm described in section (2.1.1) calculates how to distribute a given number of bits among a set of quantizers. Using the 407 conversation speech data base results in the listed distributions in Table 2 The first 4 bits are always used for the FrameSync data header. 13-dimensional FrameSync format data vectors are used. Cepstral mean subtraction did not have any impact on the bit distribution.

3.5 Data Size

An important matter is the actual amount of data. Table 3 shows the achieved data reduction if quantization is done using 56 bit codebooks. All sizes include data headers if existing.

表 2: Bit distribution for 13-dimensional parameter vectors, depending on bandwidth.

bits	header	pow	cep1	cep2	cep3	cep4	cep5	cep6	cep7	cep8	cep9	cep10	cep11	cep12
40	4	5	5	3	3	3	3	3	2	2	2	2	2	1
48	4	6	5	4	4	4	3	3	3	3	3	2	2	2
56	4	6	6	5	4	4	4	4	4	3	3	3	3	3
64	4	7	6	5	5	5	5	4	4	4	4	4	4	3

表 3: Data size comparison: 100 sec. speech

	wave file	parameter vectors	compressed bitvectors
length in kB	3200	560	70
percental length	100	17.5	2.19
		100	12.5

3.6 Computation Time

Computation of the codebook is very fast, i.e. about 70 seconds CPU time on a Pentium II 300 Mhz using 8 hours of preprocessed speech. This is seems to be independent on codebook size as the major calculation effort consists in reading the preprocessed data.

To find out the additional overhead for compression, 100 seconds of wave data were preprocessed. Table 4 shows the computing times for pcx200 (Pentium II 300 Mhz, 256 MB memory, Linux 2.0.32) and atra52 (DEC Alpha500/500, 500 MB memory, Digital UNIX 4.0) in exclusive use. Computation times on atra52 were slightly lower for compressed than for uncompressed speech because fewer disk accesses are needed. This is unlikely to play a role on low-end computers (which were not available for experiments).

表 4: Preprocessing computation time comparison for 100 sec. speech

computer	compressed	user time	kernel	realtime
pcx200	N	9.100	0.600	0:11.56
	Y	11.240	0.380	0:11.80
atra52	N	6.555	1.450	0:08.07
	Y	6.780	0.742	0:07.61

3.7 Recognition Rate Deterioration by Compression

As a quick test, 75 seconds of speech data (one conversation side, 18 utterances, ID-number TAC70102) were used to estimate the recognition rate depending on various factors. Those factors include allowed bandwidth for transmission and optimal beam setting while aiming at a real time factor not too much over 100 %. Experiments showed that coding the data with bigger codewords does not necessarily improve the recognition rate, but accelerate the recognition process so that higher beam settings can be chosen. Based on the results in Table 5, 56 bit data transfer rate was chosen for ATR's OPENHOUSE 1998 demonstration.

A more meaningful experiment is to test speech recognition performance for a big corpus of compressed and uncompressed cepstral data. We used ATRSPREC (release tag r06r02b) on a DEC AlphaStation 500/500 with 1GB memory, running Digital UNIX 4.0.

We also experimented using cepstral mean subtraction (CMS) [3]. If we want to allow the client to be any computer, we are confronted with some hardware-specific differences in the recorded data. For example microphones or AD converters have different transfer characteristics. CMS provides equalization of these characteristics of the client-side microphone and hardware.

表 5: Recognition rate comparison for TAC70102 using MFCC

bit rate	beamfactor	realtime factor	word best accuracy
32	75	122	7.41
40	75	105	34.81
48	75	110	71.11
56	80	103	79.26
64	85	103	73.33
72	90	114	72.59
448 (uncompr.)	80	74	76.30

7 utterances were excluded that exceeded the available memory for some conditions, leaving 544 utterances of the original 551 utterances. Table 6 shows the word recognition rates and realtime factor.

表 6: Comparison of word accuracy in % (realtime factor) for standard Mel-Fourier Cepstrum Coefficients (MFCC) and MFCC followed by Cepstral Mean Subtraction (CMS) depending on available bandwidth for 544 utterances

bit rate	MFCC	MFCC+CMS
56	84.20 (4.79)	86.27 (3.84)
448 (uncompr.)	84.97 (3.66)	86.83 (3.33)

For normal MFCC, the mismatch between compressed features and models results in less than a 1 % drop in recognition accuracy but a 30 % increase in the realtime factor. For MFCC+CMS, the drop in recognition accuracy due to compression is slightly less, but more importantly, the realtime factor only increases by about 10 %.

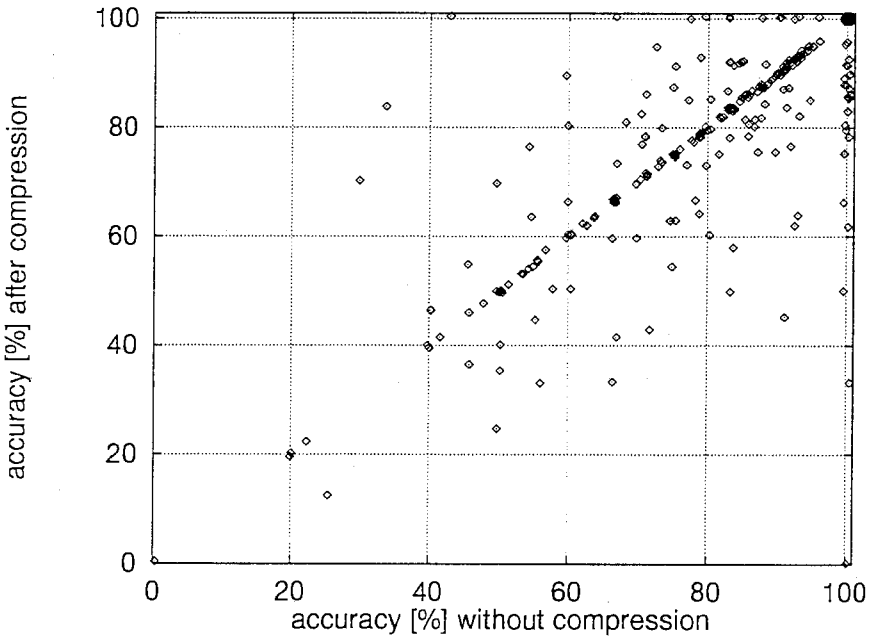

図 5: Word recognition accuracy comparison for uncompressed MFCC+CMS and compressed (56bit) MFCC+CMS features. Each point stands for one utterance.

Figure 5 shows a comparison of word accuracy of compressed and decompressed speech signals with uncompressed signals. Each point stands for one utterance. The utterances, where recognition rate was

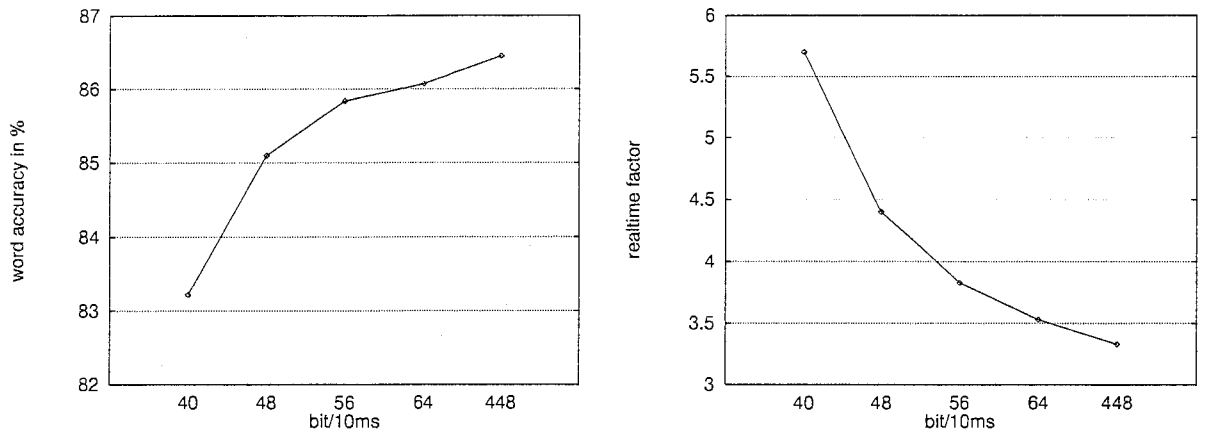

図 6: Word recognition accuracy and realtime factor depending on available bandwidth (MFCC+CMS) for 543 utterances

not changed by compression and decompression, lay on the diagonal. Points above the diagonal signify improvement by compression and decompression, points below the diagonal stand for deteriorated results. In this example, 428 utterances were unchanged, 70 deteriorated, 46 improved. The total word accuracy was 86.83 %, dropping to 86.27 % due to compression losses.

表 7: Word recognition accuracy depending on available bandwidth (MFCC+CMS) for 543 utterances

bit rate (in bit/10ms)	word acc. (%)	word corr. (%)	real time factor (%)
40	83.22	86.45	5.70
48	85.10	88.40	4.40
56	85.84	88.91	3.83
64	86.07	89.20	3.53
448 (uncompr.)	86.45	88.95	3.33

Table 7 lists recognition rates for a larger span of bit rates when using CMS. Please note, that 8 utterances could not be recognized and were therefore deleted from all comparisons leaving 543 utterances. Figure 6 shows the results of Table 7 graphically. For most realistic tasks, the drop in performance for 56 bit seems completely acceptable.

4 Future Directions

In this research, we showed that compression of cepstral data is a valuable feature for a useful speech recognition system. Down to 5.6 kbps, we only get a drop in performance of 0.5 %. Where do we go from here?

4.1 Future Improvements

Apart from applications of the compression algorithm for small bandwidth connections, it is quite intriguing that we can compress the data by 90 % without major degradation for speech recognition.

If we compressed the speech database, we would

- reduce the disk space to about about 10 %.
- if the speech recognition system is trained on compressed data, an increase in recognition rate can be expected. Currently we train on uncompressed and test on compressed data. The slight accuracy loss is probably due to this mismatch.
- correlation between succeeding frames of speech is currently not exploited. By using frame-to-frame correlation, we can expect further reduction of bandwidth without information loss.

If we train on compressed data, a possible problem might be the extreme quantization: some parameters are projected to e.g. only 2 bits, i.e. four possible values. Such quantized parameters are unlikely to comply with Gaussian modelling assumptions.

4.2 Future Applications

The feasibility of speech recognition over a narrow bandwidth channel encourages using the ATR-MATRIX speech-to-speech translation system in client-server mode.

Possible applications include:

- WWW-based implementation of ATR-MATRIX
- high-speed multi-client application

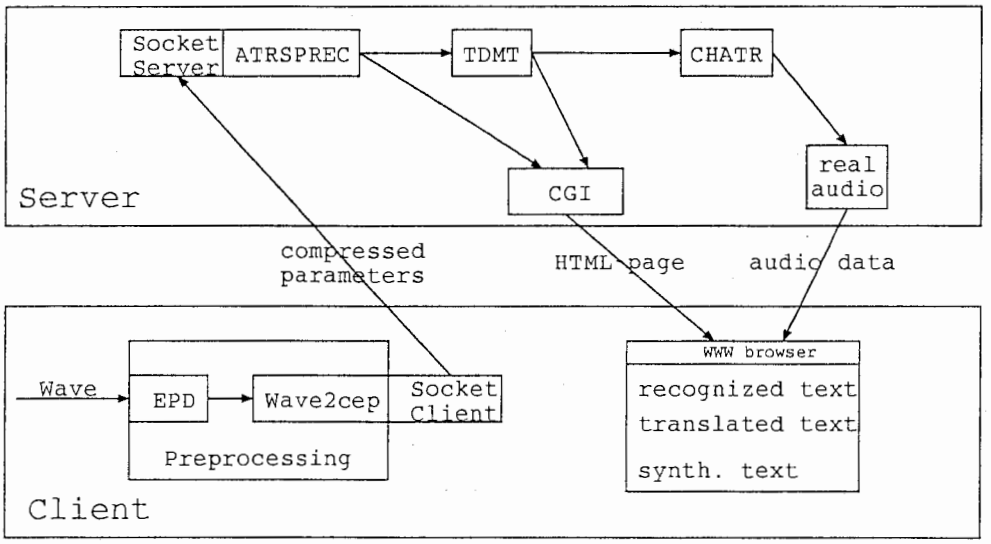

図 7: A WWW-based demonstration system for ATR-MATRIX.

An ATR-MATRIX demonstration WWW page would allow presenting the system virtually everywhere, requiring only a laptop, modem and telephone line. Preparation time and costs for a demonstration would

decrease tremendously, as no damageable computer system has to be shipped. Software preparation can be reduced to starting a browser on the laptop and starting the server script on a powerful server at ATR. Figure 7 shows the possible structure of such a WWW-based ATR-MATRIX system. This architecture would also fully fit the scenario of a Japanese traveller coming to a hotel in the USA, who wants to talk to the receptionist: both sides would communicate using the same computer.

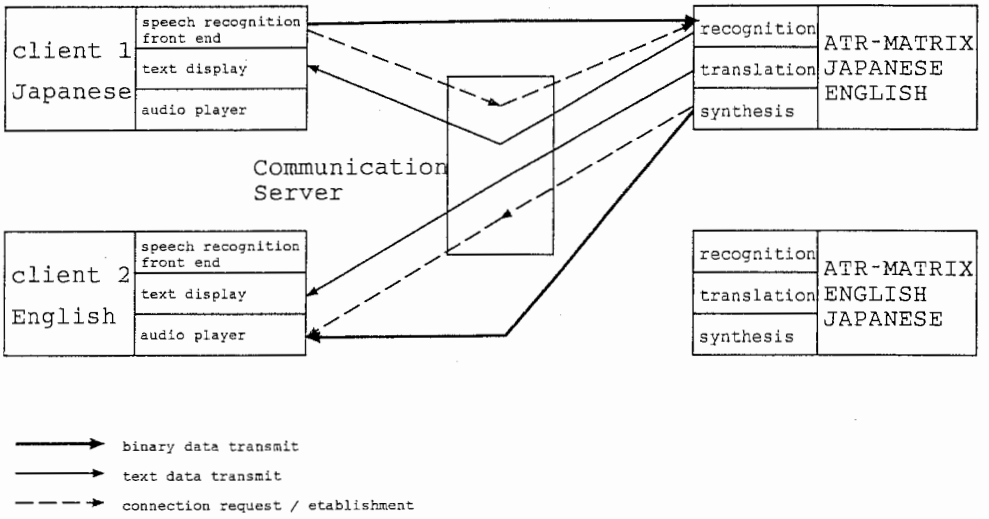

図 8: A multi-client application using narrow bandwidth connections. For clarity, connections are only shown for Japanese-to-English translation.

A more general application is a two- or multiclient scenario as shown in Figure 8. It allows two or more remote clients to communicate using narrow bandwidth connections via a communication server. This communication server would transfer text data and establish direct client-server connections for cepstral data (client → server) and speech data (server → client).

Acknowledgements

We would like to thank Dr. Seiichi Yamamoto, President of ATR-ITL, and Dr. Yoshinori Sagisaka, Department Head, for giving us a chance to do this research at ITL. We also would like to thank all members of department 1, especially Michael Schuster and Hirofumi Yamamoto, for valuable discussions.

参考文献

[1] V. Digalakis, L. Neumeyer, and M. Perakakis. Quantization of cepstral parameters for speech recognition over the world wide web. In *Proc. ICASSP*, pages II–989–992, Seattle, 1998.

[2] A. Gersho and R. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1991.

[3] M. Morishima, T. Isobe, and N. Koizumi. Normalization of the feature vector in telephone speech recognition. In *IEICE TRANSACTIONS*, pages 49–54, Kawazaki, 1997.

[4] A. Nakamura, S. Matsunaga, T. Shimizu, M. Tonomura, and Y. Sagisaka. Japanese speech database for robust speech recognition. In *Proc. ICSLP*, pages 137–140, Philadelphia, 1996.

[5] G. Ramaswamy and P. Gopalakrishnan. Compression of acoustic features for speech recognition in network environments. In *Proc. ICASSP*, pages II–977–980, Seattle, 1998.

[6] B. Reaves, A. Nishino, and T. Takezawa. ATR-MATRIX: Implementation of a speech translation system. In *Proc. Acoust. Soc. Jap.*, Spring 1998.

付録 A Program make_codebook

The program `make_codebook` reads the input data file, calculates the codebooks for en- and decoding and writes them to files.

option	default	description
`-help`	(n.a.)	Display help message.
`-config`	(none)	read parameters from file, not command line.
`-inputFd`	(none)	Name of the training data file. **Obligatory**. As the data must be read twice, `stdin` as input source is impossible.
`-inputParamType`	float	Which data type the training data is. Only float is supported.
`-inputFormat`	FrameSync	Which data format the training data is. Only FrameSync is supported.
`-inputParamSize`	13	How many parameters the data body consists of. Must be $\geq$ numberOfUsedParam.
`-numberOfUsedParam`	13	How many of the existing parameters in the data vector shall actually be used. E.g. if the training data consists of 13 cepstral coefficients and their 13 derivatives, but only the first parameters, the coefficients, shall be used, set `inputParamSize` to 26 and `numberOfUsedParam` to 13. Always the first n Parameters are used.
`-inputByteorder`	BigEndian	The training data byte order.
`-scaleFactor`	1024	See equation 1 for details.
`-numberOfBins`	32768	See equation 1 for details. It is a `short` variable, so values must not exceed 65535.
`-sumBitsOfCodewords`	48	Total size of coded vector. The higher the number the more accurate the coding will be. 4 bits will be used for header coding, the rest for the data.
`-outputEncoder`	codebook.enc	file to write the codebook for encoding.
`-outputDecoder`	codebook.dec	file to write the codebook for decoding.

Usage examples:

```
make_codebook -inputFd=/home/rgruhn/data/TAC70015.A.FSYNC -inputParamSize=26
```

Reads file `TAC70015.A.FSYNC`, which contains 26 dimensional FrameSync format data. The first 13 parameters are used for codebook training (13 is the default setting of `numberOfUsedParam`), the second 13 parameters are ignored.

```
make_codebook -config=myconfig
```

Trains codebook using configuration options in file `myconfig`.

```
make_codebook -help
```

Displays help message.

The input data for training can not be read from `stdin`, as it has to be read twice: first to calculate the mean and variance vector and the second time to calculate the index values, using equation 1. This makes the use of this program in a pipeline impossible, the data must be read from file.

付録 B Library libATRcompress

The library `libATRcompress.a` or `libATRcompress.so` resp. provides various functions for data compression and decompression as well as for codebook handling.

The functions were written to replace the write and read calls in the library `ATRevent`.

Compressed data is no own data type. It is FrameSync format. The compression algorithms are used when writing or reading to or from file or `stdin`. To make compression available without need to use the I/O-routines, a compression/decompression option was added to the `wave2cep` module, modifying the cepstra as last step of cepstral calculation.

Additional command line options are provided for I/Ocontrol configuration:

option	default	description
`-outputCompress`	OFF	set to ON to activate compression of output data.
`-inputDecompress`	OFF	set to ON to activate decompression of input data.
`-outputEncodeBookFile`	codebook.enc	codebook for compression of data.
`-inputDecodeBookFile`	codebook.dec	codebook for decompression of data, must match with previously used encoder.

Command line options were also added for ATRwave2cep:

option	default	description
`-SimCompression`	OFF	set to ON to activate compression and decompression of cepstral parameters. E.g. useful to train a recognizer on compressed data without using the I/O-routines and piping.
`-outputEncodeBookFile`	none	codebook for compression of data.
`-inputDecodeBookFile`	none	codebook for decompression of data, must match with previously used encoder.

Usage example:

```
ATRwave2cep -outputCompress=ON | ATRcep2para -inputDecompress=ON
```

Calculates cepstral parameters, compresses the output, pipes it to `ATRcep2para` where the data is used for parameter calculation.

```
ATRwave2cep -outputCompress=ON -outputEncodeBookFile=bettercodebook.enc |
ATRcep2para -inputDecompress=ON -inputDecodeBookFile=bettercodebook.dec
```

Does the same as above, but using the codebooks `bettercodebook.enc` and `bettercodebook.dec` for compression and decompression.

```
ATRall -SimCompression=ON -outputEncodeBookFile=codebook.enc -inputDecodeBookFile=codebook.dec
```

Does the same as above, without need to split the recognizer into preprocessing and recognition programs.

B.1 ATRcompress_structurelib.c

The library `ATRcompress_structurelib.c` contains some necessary functions to handle codebooks, such as memory functions and a routine to read a codebook from disk.

```
#include "codebook.h"

_MCB   *CB_malloc_MCB(int usedParam, int *bitDistribution, int flag);
void   CB_free_MCB(_MCB *codebooks);
_MCB   *CB_read_codebook (char *CB_filename, int enc_dec_flag);
```

Detailled description follows:

```
_MCB    *CB_malloc_MCB(
        int usedParam, /* number of parameters used from data vector */
        int *bitDistribution, /* size of codebooks in bits */
        int flag  /* encode/decode - flag */
        )
```

This function allocates memory for the master codebook itself and for the single codebooks and their data bodies. The size of the data bodies is allocated according to the settings in `bitDistribution`. Depending on `flag`, memory for encoding or decoding is prepared. Returns `NULL` on failure.

```
void   CB_free_MCB(_MCB *codebooks)
```

Frees the memory of a master codebook.

```
_MCB *CB_read_codebook(
        char *CB_filename, /* filename */
        int enc_dec_flag  /* flag if this is an encoder oder decoder */
        )
```

Allocates memory for a master codebook and reads it from file `CB_filename`. If `enc_dec_flag` and type of master codebook in `CB_filename` do not match or any other errors occur, this function returns `NULL`.

B.2 ATRcompress_calcOptBitDistrib.c

This file contains a function to calculate the optimal bit distribution for codebook training.

```
#include    "codebook.h"

int ATR_calc_optBitDistribution(
        int vectorDimension,
             /* how many parameters are in the vectors */
        double *varVector,
            /* variance of each parameter */
        int givenBits,
            /* how many bits are available for distribution */
        int *bitDistribution
            /* pointer to put the bit distribution vector */
        )
```

This function calculates the optimal bit distribution for quantization of parameter vectors, given variance and the number of available bits. Returns 0 on success or 1 on error.

B.3 ATRcompress_encodelib.c

This file contains an init and an write and encode function.

```
#include   "codebook.h"
#include   "InterFaceSSS.h"

int     CB_init_Encoder(char* CB_filename, char* dataType);
size_t  CB_writeEncode_data (int fd, void *datap, size_t count);
int     CB_compressWithoutWrite(void *mydatavector, unsigned char *mybitvector);
void    CB_term_Encoder();
```

Detailled description follows:

`int CB_init_Encoder(char* CB_filename, char* dataType)`

This function initializes the encoding process by allocating necessary memory, reading the codebook from file `CB_filename` and calculating all constants. `dataType` must be float or int, as other data types are not supported. Returns 0 on success or 1 on error.

`size_t CB_writeEncode_data (int fd, void *datap, size_t count)`

This function takes one FrameSync data event, compresses it, stores it into a bitvector and writes this to `fd`. This function was programmed to optimally fit in into `ATRevent_write_frame_sync` and replace all `write` calls there. In `ATRevent_write_frame_sync`, `write` is called twice, once for the header and a second time for the data body. To enhance efficiency, the header value is stored internally and later transmitted together with the data body.

The data is compressed using scalar quantization. To use only minimum bandwidth, header and codewords are transmitted in one bit vector. The FrameSync data header currently is an integer number between 0 and 12, thus can be stored using 4 bit; if these definitions are expanded, the setting

```
#define CB_TRANSMITTED_HEADER_BITS 4
```

in `codebook.h` must be changed adequately.

For reading these vectors, `CB_readDecode_data` (see section B.4) should be used.

Write is told in the input variable `count`, how many bytes of (uncompressed) data are to be written. As the really written bytes never match `count` anyway, this function returns -1 on error and `count` else. Partial write is considered a error.

`int CB_compressWithoutWrite(void *mydatavector, unsigned char *mybitvector);`

This function generates a FrameSync data event header, compresses the float vector `mydatavector` and stores both compressed header and compressed data body in the bitvector `mybitvector`.

`void CB_term_Encoder();`

This function frees allocated memory.

B.4 ATRcompress_decodelib.c

This file contains an init and a read and decode function for bitvectors sent by CB_writeEncode_data (see section B.3). The result is an event in FrameSync format.

```
#include        "codebook.h"

int     CB_init_Decoder(char* CB_filename, char* dataType);
size_t  CB_readDecode_data (int fd, void *datap, size_t count);
int     CB_decompressWithoutRead(unsigned char *mybitvector, float *mydatavector);
void    CB_term_Decoder();
```

Detailled description follows:

```
int     CB_init_Decoder(char* CB_filename, char* dataType);
```

This function initializes the decoding process by allocating necessary memory, reading the codebook from file CB_filename and calculating all constants. dataType must be float or int, as other data types are not supported. Returns 0 on success or 1 on error.

```
size_t  CB_readDecode_data (int fd, void *datap, size_t count);
```

This function reads one bit vector written by CB_writeEncode_data (see section B.3) from fd, retrieves header and codewords, decodes using the codebook and stores the decoded data, a vector of floats, to datap. If the header is not a data mark, the data body is undefined.

This function is concepted to replace the read calls in the function SSSReadFrameSync in the library ATRevent_read_frame_sync.c.

Additional information on coding and bit vectors can be found in section B.3.

It returns ≤ 0 on error or count on success. There is no direct connection between the number of bytes to be read given in count and the actual number of read bytes. This function reads into a buffer until as many bytes are read as a bitvector has. The bitvectorsize is calculated from the data in the codebook during init. This function will read until a bitvector is complete or return -1 if EOF is received.

```
int     CB_decompressWithoutRead(unsigned char *mybitvector, float *mydatavector);
```

Prepares the bit vector mybitvector for decoding by CB_decompress. The result is stored in mydatavector.

```
void    CB_term_Decoder();
```

This function frees allocated memory.

B.5 ATRcompress_compdecomplib.c

This file provides a function to compress and decompress cepstral data without using the I/O routines, i.e. without writing it. An init function is also provided.

```
#include        "codebook.h"

int CB_init_compDecomp(int id, char *mydatatype);
int CB_compDecomp(double *doubledata, int vectorsize);
```

Detailled description follows:

```
int CB_init_compDecomp(int id, char *mydatatype);
```

This function initializes both compression and decompression. Returns 0 on success or 1 on error.

```
int CB_compDecomp(double *doubledata, int vectorsize);
```

This function expects a vector of doubles and the vector size, e.g. the cepstral order plus one for the power parameter. The function converts the doubles to floats, compresses these, decompresses the bitvector, turns the floats to doubles again and stores them in `*doubledata`. `vectorsize` must match the number of parameters used in the codebooks.

付録 C Additional Experiment Results

This section contains all experiment result tables that were not included in section 3 (experiments) to keep the main part clear and readable.

C.1 Percental Mean Derivation

Please read section 3.2 for explanations.

表 8: Example tables for percental mean derivation: Upper table lists errors with the upper border bin as codevalue, bottom table lists errors if the mean between upper and lower border is used.

bits	energy	cep 1	cep 2	cep 3	cep 4	cep 5	cep 6	cep 7	cep 8	cep 9	cep 10	cep 11	cep 12
1	53.10	114.69	69.25	597.55	352.06	339.46	462.33	728.50	466.53	1539.44	528.56	959.23	481.89
2	22.56	104.82	20.16	62.11	46.70	36.81	56.72	59.23	54.07	104.30	**55.67**	**81.20**	**52.59**
3	14.98	14.37	12.27	22.94	18.88	**20.03**	**15.23**	**43.76**	**22.21**	**32.21**	22.65	23.38	16.52
4	10.42	10.92	**9.30**	**7.04**	**7.47**	5.08	11.27	19.82	9.04	27.04	9.03	18.94	12.21
5	6.81	**3.53**	1.98	5.14	2.38	2.99	3.07	9.00	3.06	12.60	2.90	7.31	5.13
6	**4.20**	2.72	1.02	1.60	1.63	2.00	2.23	3.90	2.16	5.57	1.92	1.77	1.76
7	1.85	1.04	0.56	1.07	0.47	0.66	0.59	1.43	0.83	2.10	0.59	1.29	1.25
8	1.71	0.28	0.32	0.31	0.26	0.42	0.29	0.26	0.54	0.43	0.32	1.02	0.49
9	0.59	0.18	0.20	0.12	0.15	0.12	0.14	0.16	0.22	0.30	0.15	0.35	0.15

bits	energy	cep 1	cep 2	cep 3	cep 4	cep 5	cep 6	cep 7	cep 8	cep 9	cep 10	cep 11	cep 12
1	51.48	19.25	38.39	293.06	174.12	152.79	219.54	342.50	230.95	754.42	261.86	465.92	232.98
2	21.08	43.98	36.47	26.60	22.68	20.11	18.31	10.80	26.05	38.03	**26.56**	**28.08**	**19.46**
3	14.32	5.67	10.36	7.36	8.27	**6.11**	**8.53**	**21.82**	**9.72**	**5.28**	9.68	7.69	5.32
4	10.12	**3.73**	**1.67**	**3.71**	**2.52**	4.67	3.18	9.53	3.13	13.08	2.91	7.56	5.34
5	6.65	**1.28**	3.70	1.66	1.49	1.60	2.40	4.02	1.41	5.71	1.88	1.80	1.82
6	**4.11**	1.10	1.68	1.05	0.51	0.68	0.48	1.47	0.82	2.17	0.55	1.85	0.78
7	1.80	0.29	0.80	0.33	0.49	0.43	0.93	0.26	0.30	0.45	0.62	0.52	0.50
8	0.22	0.33	0.34	0.35	0.20	0.14	0.47	0.57	0.28	0.63	0.29	0.40	0.16
9	0.56	0.14	0.13	0.19	0.08	0.17	0.26	0.31	0.10	0.22	0.12	0.09	0.18

C.2 Testing the Scale Factor A

Please read 3.3 for explanations.

表 9: Mean squared errors if scale factor of 512 is used instead of 1024.

1	1.3095	0.3939	0.3311	0.3115	0.1772	0.1087	0.0668	0.0835	0.0705	0.1092	0.0572	0.0445	0.0353
2	0.2639	0.1406	0.0983	0.0963	0.0795	0.0534	0.0326	0.0341	0.0251	0.0307	**0.0211**	**0.0177**	**0.0141**
3	0.1088	0.0677	0.0315	0.0350	0.0269	**0.0206**	**0.0112**	**0.0121**	**0.0086**	**0.0096**	0.0080	0.0060	0.0055
4	0.0597	0.0230	**0.0112**	**0.0132**	**0.0091**	0.0056	0.0036	0.0037	0.0028	0.0030	0.0028	0.0019	0.0021
5	0.0408	**0.0062**	0.0041	0.0052	0.0035	0.0019	0.0013	0.0013	0.0011	0.0012	0.0012	0.0007	0.0009
6	**0.0325**	0.0019	0.0013	0.0017	0.0013	0.0005	0.0004	0.0004	0.0003	0.0003	0.0004	0.0002	0.0003
7	0.0287	0.0007	0.0005	0.0007	0.0005	0.0002	0.0002	0.0001	0.0001	0.0001	0.0002	0.0001	0.0001
8	0.0255	0.0002	0.0002	0.0003	0.0002	0.0001	0.0001	0.0001	0.0001	0.0000	0.0001	0.0000	0.0001
9	0.0240	0.0001	0.0000	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

表 10: Mean squared errors if scale factor of 2048 is used instead of 1024.

bits	energy	cep 1	cep 2	cep 3	cep 4	cep 5	cep 6	cep 7	cep 8	cep 9	cep 10	cep 11	cep 12
1	1.3083	0.3948	0.3320	0.3122	0.1772	0.1087	0.0668	0.0772	0.0401	0.0269	0.0201	0.0154	0.0078
2	0.2656	0.1409	0.0987	0.0967	0.0796	0.0535	0.0326	0.0342	0.0178	0.0118	**0.0086**	**0.0061**	**0.0030**
3	0.1096	0.0677	0.0316	0.0351	0.0269	**0.0207**	**0.0112**	**0.0121**	**0.0067**	**0.0046**	0.0033	0.0022	0.0011
4	0.0598	0.0230	**0.0113**	**0.0132**	**0.0091**	0.0056	0.0036	0.0037	0.0024	0.0017	0.0012	0.0008	0.0005
5	0.0409	**0.0062**	0.0042	0.0053	0.0035	0.0019	0.0013	0.0013	0.0009	0.0006	0.0005	0.0003	0.0002
6	**0.0323**	0.0019	0.0013	0.0017	0.0013	0.0005	0.0004	0.0004	0.0003	0.0002	0.0002	0.0001	0.0001
7	0.0285	0.0006	0.0005	0.0007	0.0005	0.0002	0.0002	0.0001	0.0001	0.0001	0.0001	0.0000	0.0001
8	0.0254	0.0002	0.0002	0.0003	0.0002	0.0001	0.0001	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000
9	0.0239	0.0001	0.0000	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

表 11: Mean squared errors if scale factor of 4096 is used instead of 1024.

bits	energy	cep 1	cep 2	cep 3	cep 4	cep 5	cep 6	cep 7	cep 8	cep 9	cep 10	cep 11	cep 12
1	1.3089	0.3948	0.3322	0.1547	0.1173	0.0641	0.0329	0.0295	0.0172	0.0145	0.0123	0.0097	0.0064
2	0.2661	0.1408	0.0988	0.0523	0.0581	0.0289	0.0135	0.0122	0.0070	0.0060	**0.0054**	**0.0038**	**0.0026**
3	0.1099	0.0677	0.0316	0.0193	0.0224	**0.0112**	**0.0048**	**0.0045**	**0.0028**	**0.0025**	0.0023	0.0015	0.0012
4	0.0599	0.0230	**0.0113**	**0.0074**	**0.0081**	0.0037	0.0016	0.0017	0.0011	0.0010	0.0010	0.0007	0.0009
5	0.0409	**0.0062**	0.0042	0.0028	0.0031	0.0012	0.0006	0.0006	0.0005	0.0006	0.0008	0.0006	0.0009
6	**0.0323**	0.0019	0.0013	0.0011	0.0013	0.0004	0.0002	0.0003	0.0004	0.0006	0.0008	0.0006	0.0009
7	0.0285	0.0006	0.0005	0.0004	0.0005	0.0002	0.0001	0.0002	0.0004	0.0006	0.0008	0.0006	0.0008
8	0.0254	0.0002	0.0002	0.0002	0.0002	0.0001	0.0001	0.0002	0.0004	0.0006	0.0008	0.0006	0.0008
9	0.0239	0.0001	0.0000	0.0001	0.0001	0.0000	0.0001	0.0002	0.0004	0.0006	0.0008	0.0006	0.0008

表 12: Mean squared errors if scale factor of 8192 is used instead of 1024.

bits	energy	cep 1	cep 2	cep 3	cep 4	cep 5	cep 6	cep 7	cep 8	cep 9	cep 10	cep 11	cep 12
1	1.3079	0.3948	0.1258	0.0634	0.0741	0.0368	0.0264	0.0258	0.0160	0.0146	0.0128	0.0101	0.0072
2	0.2662	0.1409	0.0453	0.0249	0.0340	0.0114	0.0098	0.0102	0.0069	0.0064	**0.0060**	**0.0043**	**0.0032**
3	0.1101	0.0677	0.0147	0.0109	0.0135	**0.0035**	**0.0038**	**0.0040**	**0.0039**	**0.0041**	0.0043	0.0032	0.0029
4	0.0600	0.0230	**0.0052**	**0.0053**	**0.0055**	0.0012	0.0028	0.0036	0.0035	0.0038	0.0040	0.0032	0.0029
5	0.0409	**0.0062**	0.0019	0.0029	0.0047	0.0009	0.0026	0.0034	0.0033	0.0038	0.0039	0.0030	0.0029
6	**0.0324**	0.0019	0.0008	0.0025	0.0046	0.0009	0.0025	0.0034	0.0032	0.0037	0.0039	0.0030	0.0029
7	0.0285	0.0006	0.0004	0.0024	0.0045	0.0008	0.0024	0.0034	0.0032	0.0037	0.0039	0.0030	0.0029
8	0.0254	0.0002	0.0004	0.0024	0.0045	0.0008	0.0024	0.0034	0.0032	0.0036	0.0039	0.0030.	0.0029
9	0.0239	0.0001	0.0004	0.0023	0.0045	0.0008	0.0024	0.0034	0.0032	0.0036	0.0039	0.0030	0.0029

付録 D Details of Recognition Experiments

All recognition experiments were run with ATRSPREC release tag r06r02b. Specifically, we used the script

```
$ATRSPREC/script/atrLatt.py
```

which allows online calculation of CMS based on pause-units, i.e. speech segments in the .TRS transcription files. Configuration files are calculated on-the-fly.

D.1 Configuration File for Recognition

Here is an example of a configuration file when using 56bit compression and CMS, generated by atrLatt.py:

```
I/Ocontrol:inputByteorder=BigEndian
I/Ocontrol:inputFd=stdin
I/Ocontrol:inputFormat=NoHeader
I/Ocontrol:inputParamSize=160
I/Ocontrol:inputParamType=short
I/Ocontrol:outputByteorder=BigEndian
I/Ocontrol:outputFd=stdout
I/Ocontrol:outputFormat=NoHeader
I/Ocontrol:outputParamSize=26
I/Ocontrol:outputParamType=float

ATRwavecut:PausePeriod=NOT
ATRwavecut:SamplingFrequency=16000.0
ATRwavecut:pause_symbol=-

ATRwave2cep:AnalysisType=fft
ATRwave2cep:CepstrumOrder=12
ATRwave2cep:CutoffHighFrequency=8000
ATRwave2cep:CutoffLowFrequency=0
ATRwave2cep:DebuggingLevel=0
ATRwave2cep:FilterBankOrder=16
ATRwave2cep:FrameLength=20
ATRwave2cep:FrameShift=10
ATRwave2cep:FrequencyWarping=mel
ATRwave2cep:LagWindowFactor=0.01
ATRwave2cep:LpcOrder=16
ATRwave2cep:Preemphasis=0.98
ATRwave2cep:SamplingFrequency=16000
ATRwave2cep:SimCompression=ON
ATRwave2cep:TimeWindow=hamming
ATRwave2cep:inputDecodeBookFile=/home/singer/SPREC/sample/ATRcompress/codebook.cms56dec
ATRwave2cep:outputEncodeBookFile=/home/singer/SPREC/sample/ATRcompress/codebook.cms56enc

ATRcep2para:CepstrumOrder=12
ATRcep2para:DeltaCepstrumWindow=9
ATRcep2para:LDA=
ATRcep2para:OutputParameter=pow+cep(12)+dpow+dcep(12)
ATRcep2para:deltaCepstrumPadding=zero
ATRcep2para:rho=1.0

ATRlattice:UTT_END=6
ATRlattice:UTT_END_delay=70
ATRlattice:UTT_START=5
ATRlattice:active_model=all
ATRlattice:amname=/dept1/work1/ResearchJ/V5/amodel/AM.M.CMS.bin,/dept1/work1/ResearchJ/V5/amodel/AM.F.CMS.bin
ATRlattice:amscale=1.000000
ATRlattice:backward_frame=-1
```

```
ATRlattice:beam=110,110
ATRlattice:dimension=26
ATRlattice:frame_shift=10
ATRlattice:lexicon=/dept1/work1/ResearchJ/lmodel/19981118/LEX.W.comp.h
ATRlattice:lmscale=8,13
ATRlattice:ngram=Multi-Class-2,/dept1/work1/ResearchJ/lmodel/19981118/multicomp.700.bin
ATRlattice:pause_symbol=-
ATRlattice:phone_boundary=ON
ATRlattice:state_skip=ON,75000
ATRlattice:wdpenalty=0,0
ATRlattice:word_boundary_skip=2
ATRlattice:word_merge=all
ATRlattice:work_area=3600,150
```

D.2 Some Script Fragments Used for Recognition

The following (bash) script fragment trains encoding and decoding codebooks given a feature parameter file in FrameSync format:

```
for bit in 40 48 56 64 ; do
  $ATRSPREC/src/ATRCOMPRESS/make_codebook \\
   -inputFd=/home/pcx200/singer/407.cms.fsync -inputParamSize=13 \\
   -sumBitsOfCodewords=$bit \\
   -outputEncoder=codebook.cms"$bit"enc -outputDecoder=codebook.cms"$bit"dec
done
```

Recognition experiments for these bit rates using CMS were run with the following bash script. `latDir` stands for a directory of result lattices and `resultFile` contains detailed results on a per-utterance basis as a Python list of dictionaries.

```
for bit in 40 48 56 64 ; do
 atrLatt.py \\
  -ATRwave2cep:SimCompression=ON \\
  -ATRwave2cep:outputEncodeBookFile=$ATRSPREC/sample/ATRcompress/codebook.cms"$bit"enc \\
  -ATRwave2cep:inputDecodeBookFile=$ATRSPREC/sample/ATRcompress/codebook.cms"$bit"dec \\
  -calcCMS=CalcMeanPU \\
  -ATRlattice:amname=/dept1/work1/ResearchJ/V5/amodel/AM.M.CMS.bin,/dept1/work1/ResearchJ/V5/amodel/AM.F.CMS.bin \\
  -resultFile=result.Lattices"$bit" -latDir=Lattices"$bit" >& atrLatt.log"$bit"
done
```

Finally, a result file can be analyzed by a (python) script fragment. This can be interactively executed by using mule/emacs, marking the code as a region and using `M-x py-execute-region` if your mule is properly configured.

```
import os, pickle
os.sys.path.append(os.environ['ATRSPREC']+'/script/python/lib')
import atrscore

# read in the python object from file
f=open('/home/singer/SPREC/script/result.Lattices56');res=pickle.load(f);f.close()

# get rid of None entries, i.e. where we did not get a valid recognition result
bad = filter(lambda x: res[x]==None, res.keys())
for i in bad: del res[i]

# print summary
summary = reduce(lambda x,y,res=res: x+res[y], res.keys(),atrscore.RESULT())
os.sys.stderr.write("acc: %.2f, corr: %.2f, realtime factor: %.2f, ids: %d\n" %
                    (summary.acc(), summary.cor(), summary.realtime(), len(res.keys())))
```

Obviously, you can do much more, like plotting the results using atrgnuplot.py etc.