

The CMU Sphinx-4 Speech Recognition System

Paul Lamere^[1], Philip Kwok^[1], Evandro B. Gouvêa^[2], Bhiksha Raj^[3],
Rita Singh^[2], William Walker^[1], Peter Wolf^[3]

1. Sun Microsystems Laboratories, USA
2. Carnegie Mellon University, USA
3. Mitsubishi Electric Research Laboratories, USA

Abstract

The Sphinx-4 speech recognition system is the latest addition to Carnegie Mellon University's repository of Sphinx speech recognition systems. It has been jointly designed by Carnegie Mellon University, Sun Microsystems Laboratories and Mitsubishi Electric Research Laboratories. It is differently designed from the earlier Sphinx systems in terms of modularity, flexibility and algorithmic aspects. It uses newer search strategies, is universal in its acceptance of various kinds of grammars and language models, types of acoustic models and feature streams. Algorithmic innovations included in the system design enable it to incorporate multiple information sources in an elegant manner. The system is entirely developed on the Java™ platform and is highly portable, flexible, and easier to use with multithreading. This paper describes the salient features of the Sphinx-4 decoder and includes preliminary performance measures relating to speed and accuracy.

1. Introduction

The Sphinx-4 speech recognition system has been jointly developed by Carnegie Mellon University, Sun Microsystems Laboratories, and Mitsubishi Electric Research Laboratories (MERL). It has been built entirely in the Java™ programming language. In this paper we describe the significant features of the Sphinx-4 decoder. It is highly modular and flexible, supporting all types of HMM-based acoustic models, all standard types of language models, and multiple search strategies. Algorithmic innovations in the system enable the concurrent use of multiple information streams. The Sphinx-4 system is an open source project. The code has been publicly available at SourceForge™ since its inception. The design, results, and team meeting notes are also publicly available.

Sphinx-4 has been developed in the Java programming language, which has inherent advantages from the stand-point of code maintenance. The Java platform is highly portable: once compiled, the bytecode can be used on any system that supports the Java platform. Its garbage collection feature simplifies memory management greatly, freeing the programmer from any memory leakage concerns. The Java programming language provides a standard way of writing multithreaded applications to easily take advantage of multi-processor machines. Also, the Javadoc™ tool automatically extracts information from comments in the code and creates html files that provide documentation about the software interface.

The Sphinx-4 architecture has been designed for modularity. Any module in the system can be smoothly exchanged for another without requiring any modification of the other modules. Furthermore, the particular module to be used can be

specified at run time through a command line argument, with no need to recompile the code. One can, for instance, change the language model from a statistical N-gram language model to a context free grammar (CFG) or a stochastic CFG by modifying only one component of the system, namely the *linguist* (Section 2.2.2). Similarly, it is possible to run the system using continuous, semi-continuous or discrete state output distributions by appropriate modification of the *acoustic scorer* (Section 2.2.3). The system permits the use of any level of context in the definition of the basic sound units. Information from multiple information streams can be incorporated and combined at any level, *i.e.*, state, phoneme, word or grammar. The search module can also switch between depth-first and breadth-first search strategies. One by-product of the system's modular design is that it becomes easy to implement it in hardware.

In the following sections we describe the overall architecture and salient design aspects of the system, focusing on newer algorithmic improvements that have been included in it.

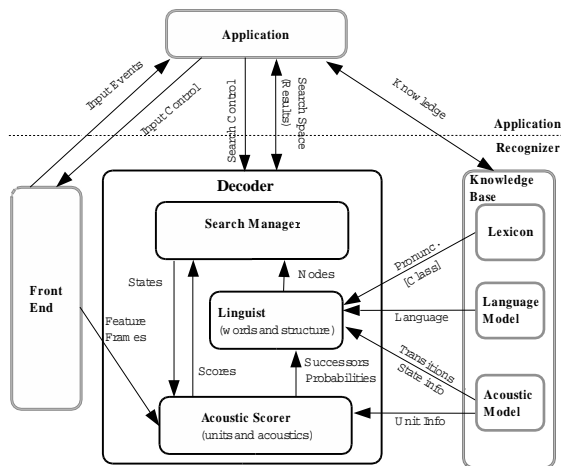


Figure 1: Sphinx-4 system architecture. The main blocks are the front end, the decoder, and the knowledge base. The decoder comprises the linguist, the search manager, and the acoustic scorer. The communication between these modules, as well as communication with an application, are depicted.

2. System architecture

Figure 1 shows the overall architecture of the Sphinx-4 decoder. The speech signal is parameterized at the front-end module, which communicates the derived features to the decoding block.

The decoding block has three components: the search manager, the linguist, and the acoustic scorer. These work in tandem to perform the decoding. In the following sections we describe each of these components in greater detail.

2.1. Front end

Figure 2 shows a detailed representation of the front-end module. The module consists of several communicating blocks, each with an input and an output. Each block has its input linked to the output of its predecessor. When a block is ready for more data, it reads data from the predecessor, and interprets it to find out if the incoming information is speech data or a control signal. The control signal might indicate beginning or end of speech – important for the decoder – or might indicate data dropped or some other problem. If the incoming data is speech, it is processed and the output is buffered, waiting for the successor block to request it. This design allows the system to be used in live or batch mode without modification, since it can handle control signals, such as start or end of speech, essential for live mode operation.

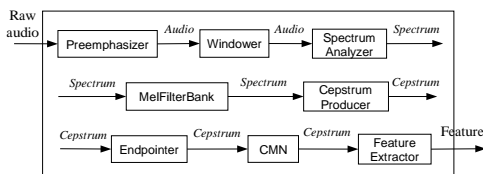


Figure 2: Sphinx-4 front end. The current implementation is compatible with state of the art standards. Each block is independent of the others, and can be easily replaced as needed.

One of the features of this design is that the output of any of the blocks can be tapped. Similarly, the actual input to the system need not be at the first block, but can be at any of the intermediate blocks. The current implementation permits us to run the system using not only speech signals, but also spectra, cepstra, etc. In addition, any of the blocks can be replaced. Thus, other kinds of features such as auditory representations can also be plugged in. Additional blocks can also be introduced between any two blocks, to permit noise cancellation or compensation on the signal, on its spectrum or on the outputs of any of the intermediate blocks. Features computed using independent information sources, such as visual features, can be directly fed into the decoder, either in parallel with the features from the speech signal, or bypassing the latter altogether.

The system is capable of running continuously from a stream of input speech. It is also capable of three other modes of operation: fully endpointing, where the system performs explicit endpointing, determining both beginning and ending endpoints of a speech segment automatically, click to talk, where the user indicates the beginning of a speech segment but the system determines when the speech ends automatically, and push to talk, where the user indicates both the beginning and the end of a speech segment.

Currently, endpoint detection is performed by a simple algorithm that compares the energy level to three threshold levels. Two of these are used to determine start of speech, and one for end of speech. The endpointer detects the start and/or end of speech from the incoming audio, as required, and discards non-speech segments, sending only speech segments to the decoder. The decoder thus does not waste time processing non-speech segments.

2.2. Decoder

The decoder block consists of three modules: search manager, linguist, and acoustic scorer. These are described below.

2.2.1. Search Manager

The primary function of the search manager is to construct and search a tree of possibilities for the best hypothesis. The construction of the search tree is done based on information obtained from the linguist. In addition, the search manager communicates with the acoustic scorer to obtain acoustic scores for incoming data.

The search manager makes use of a token tree. Token trees have been utilized in other speech recognition systems [1]. The token tree consists of a set of tokens that contain information about the search and provides a complete history of all active paths in the search. Each token contains the overall acoustic and language scores of the path at a given point, a *SentenceHMM* reference (see Section 2.2.2), an input feature frame identification, and a reference to the previous token, thus allowing back-tracing. The *SentenceHMM* reference allows the search manager to fully categorize a token to its senone, context-dependent phonetic unit, pronunciation, word, and grammar state.

The search algorithm maintains a set of active tokens in an *active list*. The active list represents the tips of the active search branches. During the search, each input feature frame is scored against the acoustic models associated with each token in the active list, and low scoring branches are pruned. The search manager updates the active list based on the successor *SentenceHMM* states of the tokens remaining after pruning.

Sphinx-4 provides a beam pruner that constrains the scores to a configurable minimum relative to the best score, while also keeping the total number of active tokens to a configurable maximum. New implementations can easily be created that provide alternative methods of storing and pruning the active list. The garbage collector automatically reclaims unused tokens, thus simplifying the implementation of the pruner by merely allowing a token to be removed from the set of active tokens.

The active list is available as part of the final recognition result. Applications can use the active list to inspect the highest scoring paths. This can be used, for instance, to construct an N-Best list.

Search through the token tree and the sentence HMM is performed in two ways: *depth-first* or *breadth-first*. Depth-first search is similar to conventional stack decoding. The most promising tokens are expanded in time sequentially. Thus, paths from the root of the token tree to currently active tokens can be of varying lengths. In breadth-first search, on the other hand, all active tokens are expanded synchronously, making the paths from the root of the tree to the currently active tokens equally long.

In Sphinx-4, breadth-first search is performed using the standard Viterbi algorithm as well as a new algorithm called *Bush-derby*. For details about this algorithm please refer to [2].

2.2.2. Linguist

The linguist translates linguistic constraints provided to the system into an internal data structure, the *grammar*, which is usable by the search manager (Section 2.2.1). Linguistic constraints are typically provided in the form of context free grammars, N-gram language models, finite state machines etc. The grammar is a directed graph where each node represents a set of words that may be spoken at a particular time. The nodes are

connected by arcs which have associated language probabilities that are used to predict the likelihood of transitioning from one node to another. Sphinx-4 currently provides grammar loaders for three external grammar formats: a word list grammar loader, which generates a flat-unigram grammar from a simple list of words; an N-Gram grammar loader, which loads statistical N-gram models represented in the well-known ARPA-standard format; and a finite state transducer (FST) grammar loader, that creates a grammar based upon an externally defined finite state transducer. The pluggable nature of Sphinx-4 allows new grammar loaders to be easily added to the system.

The grammar is further compiled into a SentenceHMM, which is a directed state graph where each state in the graph represents a unit of speech. This separation of the grammar graph and the SentenceHMM graph allows the definition and loading of linguistic constraints to be completely decoupled from the internal representation of the SentenceHMM and from the operation of the decoder. Support for new grammar formats can be easily added to Sphinx-4 without knowledge of the internal representation of the search space.

Grammar nodes are decomposed into a series of word states, one for each word represented by the node. Word states are further decomposed into pronunciations states, based on pronunciations extracted from a dictionary maintained by the linguist. Alternate pronunciations are allowed. Each pronunciation state is then decomposed into a series of unit states, where units may represent phonemes, diphones, syllables, etc. and can be specific to contexts of arbitrary length. Each unit is then further decomposed to its sequence of HMM states. The SentenceHMM thus comprises all of these states. States are connected by arcs that have language, acoustic and insertion probabilities associated with them.

Although the contents of a SentenceHMM are well defined by the linguist, there are a number of strategies that can be used in constructing the SentenceHMM that affect the search. By altering the topology of the SentenceHMM, the memory footprint, perplexity, speed and recognition accuracy can be affected. The modularized design of Sphinx-4 allows different SentenceHMM compilation strategies to be used without changing other aspects of the search. Linguists can be broadly classified as either static or dynamic. A static linguist creates the entire SentenceHMM statically, prior to recognition, while the dynamic linguist only creates portions of the SentenceHMM that are relevant to the current set of hypotheses being evaluated. Notwithstanding the logistic constraints, the actual SentenceHMM structure itself is independent of whether the linguist is static or dynamic. The choice of whether to employ one or the other depends mainly on the vocabulary size, language model complexity and desired memory footprint of the system.

To clarify how Sphinx-4 handles true n-grams, *i.e.*, phonetic units of arbitrary context size, we describe here in more detail the operation of a particular static linguist, the *SimpleLinguist*. One of the primary difficulties in building the SentenceHMM graph is in managing cross word contexts. The SimpleLinguist makes the simplifying assumption that each grammar node contains at most one word. Since Sphinx-4 allows unit contexts to be of arbitrary size, the decomposition of a pronunciation state into a series of unit states needs to consider all possible left and right contexts. This can be particularly complex at word boundaries, especially when a context can extend into more than one subsequent word, as can happen when a right context size is greater than one. To manage this complexity, the SimpleLinguist explicitly tracks the set of all possible entry (left) and exit (right) contexts for each word.

The SimpleLinguist begins with the previously described grammar graph. The SimpleLinguist first creates an object, a *GState*, for each grammar node. The *GState* is used to manage information for each grammar node. The SimpleLinguist then visits each grammar node and constructs a list of the starting contexts for each node based upon the word at that node. Similarly, a list of ending contexts are generated for each node. The list of starting and ending contexts are maintained in the *GState* object associated with each node. Next, the SimpleLinguist collects the set of left contexts for each node by aggregating all of the previously gathered ending contexts for each upstream grammar node. In a similar fashion, the set of right contexts are collected by aggregating the starting contexts for each downstream node. Subsequently, the linguist generates the set of entry points into each word by generating a list of context pairs consisting of the product of the set of left contexts and the set of entry points for the word. Each context pair represents a single left context entering the word, matched with each possible beginning context. The units and associated HMM states for the word are attached to each entry point using the left context of the entry point context pair as the initial left context. Similarly, the collected set of right contexts are used to fill out the right contexts for units as they are attached as necessary. As units are added, the linguist searches for common units with identical left and right contexts at the same position within a particular pronunciation and shares these units among multiple paths. Finally, the tail units of a word are added to a list of exit points for the word.

At this point, each *GState* contains a fully expanded word with all contexts in proper form along with a set of entry and exit points for the word. The only remaining task is to connect the words together. The linguist creates an arc between each exit point of a word and the appropriate entry point of the next word as indicated by the grammar. The linguist ensures that connections are only made between units that have compatible left and right contexts. Once the connections have been made, we have a fully populated SentenceHMM. All *GStates* are discarded, since the information they contain is no longer needed.

2.2.3. Acoustic Scorer

The task of the acoustic scorer is to compute state output probability or density values for the various states, for any given input vector. The acoustic scorer provides these scores on demand to the search module. In order to compute these scores, the scorer must communicate with the front-end module to obtain the features for which the scores must be computed. When parallel input features are used, the acoustic scorer must score the incoming features against the proper set of HMMs.

The scorer retains all information pertaining to the state output densities. Thus, the search module is ignorant of whether the scoring is done with continuous, semi-continuous or discrete HMMs. Any heuristic algorithms incorporated into the scoring procedure for speeding it up can be performed locally within the search module. Where such heuristics can benefit from additional information derived from the search module, however, this information can be obtained on demand. The modular nature of the design permits us to include conventional Gaussian scoring procedures, as well as several other algorithms.

3. Experimental Evaluation

In this section we report results on some benchmark tests of increasing complexity. As a comparator, we report performance

Table 1: Benchmark tests comparing Sphinx-3 (S3) and Sphinx-4 (S4). Experiments were run on a Sun Blade™ 1000 workstation with dual 750 MHz UltraSPARC® III processors.

Task name	Speed (xRT)		WER (%)	
	S3	S4	S3	S4
Isolated Digits	0.4	0.1	0.7	1.4
Connected Digits	0.2	0.2	1.1	0.3
Small Vocab.: words				
–flat unigram	6.8	2.5	9.8	7.7
–trigram	6.8	3.0	1.2	2.6
–FST	N/A	1.2	N/A	1.1
Small Vocab.: full				
–flat unigram	6.9	4.1	21.0	21.7
–trigram	6.6	6.0	15.0	13.1
–FST	N/A	2.2	N/A	8.2
Medium Vocabulary				
–flat unigram		13.7	18.7	17.3
–unigram		15.3	13.2	14.5
–bigram		19.0	1.8	3.3

on the CMU Sphinx-3 [3] speech recognition system on the same tasks. For all experiments acoustic models were trained with the training module of the CMU Sphinx-3 system. The following benchmark tests have been conducted so far:

Isolated digits: This was conducted on the TI46 isolated digits database [4]. The acoustic models were trained on speech from the adult speakers in the TIDigits database [5].

Connected Digits: This test was conducted on a subset of the TIDigits database containing adult speakers. The acoustic models used were the same as those for isolated digits.

Small Vocabulary: This was conducted on CMU’s Census database (AN4) [6]. The entire database (including components conventionally marked as “training” and “test”) was used for recognition. Acoustic models were trained using the Wall Street Journal (WSJ) database [7]. We ran several tests, including tests on the control words (words) component of AN4 as well as on the entire database (full). Performances with a flat unigram language model (LM) and a trigram LM were evaluated, as well as an FST created from the trigram LM.

Medium Vocabulary: This test used the speaker independent portion of the Resource Management database (RM1) [8]. Acoustic models were trained using all the training data pertaining to the task. We created several statistical LMs, which include a flat unigram, a unigram, and a bigram. The LMs were created from a training set provided with the database.

Table 1 shows the results of the benchmark tests. Both word error rate (WER) and recognition speed in times real time are shown in the table.

The results in Table 1 show that the Sphinx-4 is clearly comparable to the Sphinx-3 in recognition performance and significantly superior in terms of speed, on the tasks evaluated. Sphinx-4’s ability to handle FSTs, for example, gives it a clear advantage. The Sphinx-4 performance has not been optimized on the bigram and trigram tasks at the time of this submission. As such, the evaluation of medium vocabulary tasks is ongoing, and large vocabulary tasks will be approached shortly. The optimized trigram tasks and the completed medium and large vocabulary evaluations will be completed by the time the paper is presented. They will also be reported on SourceForge as and when they are completed.

4. Summary

In this paper we have described the salient features of the architecture of CMU’s latest Sphinx-4 speech recognition system. We have described the advantages of the system’s modular design, its flexibility in the usage of various kinds of acoustic and language representations, and the features due to the Java platform that it has been developed on. The recognizer incorporates the conventional Viterbi search algorithm, as well as a new algorithm in its search, the Bush-derby algorithm. In addition, Sphinx-4 allows the use of parallel feature streams in a natural way. Benchmark tests show that the Sphinx-4 is comparable to Sphinx-3 in recognition performance, and is much faster in some cases. Although all benchmark tests have not yet been performed, these will be reported on SourceForge as they are completed.

5. Acknowledgments

The authors thank Prof. Richard Stern at CMU, Robert Sproull at Sun Microsystems, and Joe Marks at MERL, for making this team possible. We also thank SourceForge.net for their contribution to the open source community. The authors also thank Sun Microsystems Laboratories and the current management for their continued support and collaborative research funds. Rita Singh was sponsored by the Space and Naval Warfare Systems Center, San Diego, under Grant No. N66001-99-1-8905. The content of this publication does not necessarily reflect the position or the policy of the US Government, and no official endorsement should be inferred.

6. References

- [1] S. J. Young, N. H. Russell, and J. H. S. Russell, “Token passing: A simple conceptual model for connected speech recognition systems,” Cambridge University Engineering Dept, Tech. Rep., 1989.
- [2] B. Raj, M. Warmuth, R. Singh, and P. Lamere, “Bush-derby search: the paper,” in *Proceedings of the European Conference on Speech Communication and Technology*. ISCA, 2003.
- [3] P. Placeway, S. Chen, M. Eskenazi, U. Jain, V. Parikh, B. Raj, M. Ravishankar, R. Rosenfeld, K. Seymore, M. Siegler, R. Stern, and E. Thayer, “The 1996 hub-4 sphinx-3 system,” in *Proceedings of the DARPA Speech Recognition workshop*. DARPA, 1997.
- [4] G. R. Doddington and T. B. Schalk, “Speech recognition: Turning theory to practice,” *IEEE Spectrum*, vol. 18, no. 9, pp. 26–32, Sept. 1981.
- [5] R. G. Leonard and G. R. Doddington, “A database for speaker-independent digit recognition,” in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, vol. 3. IEEE, 1984, p. 42.11.
- [6] A. Acero, *Acoustical and environmental robustness in automatic speech recognition*. Kluwer Academic Pubs., 1993.
- [7] D. Paul and J. Baker, “The design of the wall street journal-based csr corpus,” in *Proceedings of ARPA Speech and Natural Language Workshop*. ARPA, Feb. 1992, pp. 357–362.
- [8] P. Price, W. M. Fisher, J. Bernstein, and D. S. Pallett, “The darpa 1000-word resource management database for continuous speech recognition,” in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, vol. 1. IEEE, 1988, pp. 651–654.