Mathifier – Speech Recognition of Math Equations

Salim N. Batlouni, Hala S. Karaki, Fadi A. Zaraket, Fadi N. Karameh

American University of Beirut

***Abstract*—Speech recognition has become widely used across many applications. Telephone systems can route a phone call based on what the caller says, control systems can respond to actions said by the controller, and mobile phones can recognize the speech of a contact's name and call the respective contact directly. However, speech recognition has found little use in recognition of textual material due to the large dictionary and hence large word error rates. Mathifier constricts the speech recognition to math equations; it takes as input math formulas presented in the form of user speech and produces the equations in digital mathematical form. The smaller dictionary and the specific grammar structure of the math equations help restrict the problem of the recognition process. The program has room for smartly guessing words based on the grammar structure and thus resulting in a lower error rate and better recognition. Mathifier uses Sphinx, a modular speech recognition tool from CMU, and adapts it to recognize math equations and convert them into latex form in real time.**

I. Introduction

EVEN though speech recognition has infiltrated many applications, its use is still limited as a replacement to typing text. People have to speak slowly and clearly and still expect errors when using speech recognition programs. That keeps the keyboard as the first option to text data entry. However, when it comes to math equations, people speak math equations much faster than they could type them or even write them by hand. For this reason, a speech recognizer may serve as a realistic and preferable option to typing math equations. Also, a speech recognizer that is focused on recognizing only math equations has a very specific dictionary to be followed, with very exact grammatical sentences thus allowing for lower processing time and better results. In other words, the recognizer has room for smartly guessing what the user is saying, hence greatly improving the accuracy of recognition. Having a very specific dictionary means much faster processing, since the system spends less time searching between available options (or, as we'll see later, searching the graph).

The market already offers MathTalk [1], a speech recognition tool for math equations. Mathifier, however, offers several advantages over the commercial alternative: Mathifier is free, open-source, and allows for natural mathematical speaking, as shown in Section III-E. MathTalk, on the other hand, is sold starting at $300. Also, the user needs to learn special commands to be able to use the program.

To convert speech into software-understandable features, two methods can be used: the Hidden Markov Model (HMM), and the lesser known time-warping method. The HMM method is known to be the de-facto standard due to its high accuracy and low computational needs. Both methods are discussed in more detail in Section II-A. Sphinx, which Mathifier uses to convert speech to features, uses the HMM method. This is more suitable in our case, since we need to recognize speech in real time.

In this paper we make the following contributions: (1) We use a novel lazy grammar approach to automatically translate mathematical speech into digital form in real time, and (2) we provide a user friendly tool to aid in the process of math speech recognition and allow editing and correcting the produced digital formulae.

II. Background

A. Models for Speech Recognition

Several models for speech recognition are present. The two most promising ones are Dynamic time warping and Hidden Marcov Models. Dynamic time warping (DTW) is an algorithm for measuring similarity between two sequences which may vary in time or speed [2]. It has found widespread use in voice recognition due to its capability to solve major problems faced when comparing the feature vectors of the stored speech signal with incoming speech signal. Although DTW was found to be of much use in speech recognition, it is computationally expensive, which limits its use in applications. The second method which has become predominant in the last several years is Hidden Markov Models[3]. Speech recognition was one of the first and most popular applications of HMMs, principally due to the ease of implementation of the overall recognition system, in addition the ease, efficiency, and availability of training algorithms for estimating the parameters of the model from finite training sets of speech data.

B. Speech Recognition System: CMU Sphinx IV

For this project we decided to use CMU Sphinx. It is an open source speech recognition tool developed at CMU. The design of Sphinx 4 is modular; this allows us to modify certain modules within the program without affecting the rest of the system. The general architecture can be seen on Figure 1 The original application contains the user interface on one side, and the interface with the speech recognizer (Sphinx 4, that is) on the other. Sphinx 4 has a front end that receives speech waveforms and converts them to features. However, before Sphinx starts to decode, or "recognize", the speech, it has to load a knowledge base, which is all the information it needs to know in order to recognize incoming speech. Naturally, the knowledge base is specific to a certain language, and sometimes even a dialect. The features as well as the knowledge base are supplied to the decoder, which in turn uses both components in order to recognize speech and convert it to text.

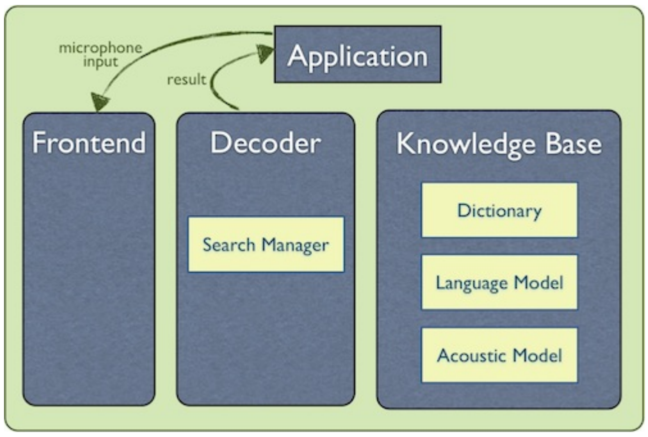


Fig. 1. Sphinx Architecture

1) Front End: The Front End is used to transform an input audio signal into a sequence of output features, which will be used for processing. This block extracts the cepstral features from the input audio signal in real time and feeds these features to the Sphinx decoder.

2) Knowledge Base: The Knowledge Base provides needed information to the decoder in order to do its job. It consists of three components: a dictionary, a language model, and an acoustic model. The dictionary is a list of words along with their phoneme sequence. The language model is the “grammar” to be used which dictates the sequence of allowed words. For instance, once the word ’plus’ is recognized, the language model knows that ’divided’ will not follow, and hence the recognizer will not consider ’divided’ as a possible option. To implement language models correctly, Sphinx uses a standard established by Java, known as the Java Speech Grammar Format (JSGF) [4]. The acoustic model has acoustic data (HMM) for every phoneme in the dictionary. Sphinx supports the use of triphones, where the phoneme has several models based on the context it is in. For example, the phoneme AH will be different when preceded by a silence than when preceded by M.

3) Search Manager and Search Graph: The Search Graph is a directed acyclic search graph, constructed from the language model, dictionary, and acoustic model. The language model or grammar defines the transitions allowed between words and their probabilities. Each word points to a list of possible pronunciations of that word and each respective pronunciation points to its list of units which are the phonemes that make up the word. Sphinx IV allows for a dynamic construction of the Search Graph during decoding. This is done through the use of the Search Manager and aids in proper real time recognition [5].

C. Application

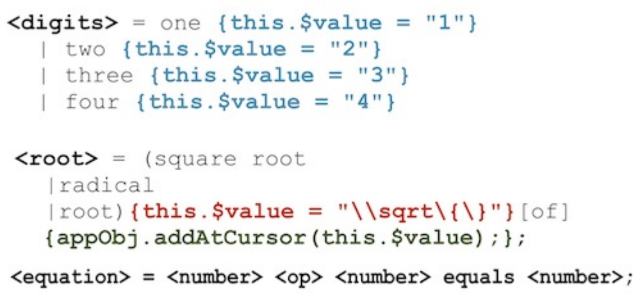
Sphinx can be thought of as the inner layer (recognizer), and the application as the outer layer (interface) with which a user will be interacting. The application layer provides the input to the recognizer and the output to the user. In our project, the application layer provides the input audio signal to Sphinx and outputs the compiled Latex to the user.

III. Mathifier Components

Mathifier provides the user with the visual feedback of what they’re saying in real-time, i.e. the display of equations in mathematical form and not the Latex code. Section III-A1 presents how in Mathifier the utterance a user says journeys through the Grammar Layer and Application Layer to to come out as compilable latex code. This is then followed by sections that present the progression of the grammar model which yielded the best results. Section Section III-A2 explains the steps taken to train the system followed by Section III-A3 which demonstrates how the results gets to be displayed in the final visual math form to the user. Section III-A4 mentions some program functionalities which were implemented based on user intuition.

A. Grammar Language Model

1) Relationship between Grammar layer and Application Layer: As previously noted in Section II-B3, the grammar language model dictates the allowed transitions between words. In Mathifier, we specified a mathematical language model which only allows transitions between words which can be mathematically sound. For example the transition from ’plus’ to ’divided by’ is not an allowed one in math grammar. To implement this math grammar, we used the Java Speech Grammar Format (JSGF) which sphinx utilizes to know the possible word transitions in real-time. JSGF allows grammar nodes to be specified as in the example of the node ’<equation>’ in Figure 2. In grammar nodes, the allowed word transitions are specified, and a word in a grammar node may be itself another grammar node such as <number> and <op> within the node <equation>. A key feature provided by the JSGF is tagging. Each node or word can be assigned a specific string value. Accordingly, values can be assigned to their Latex equivalent such as ’\sqrt’ for square root in Figure 2. Moreover, Java functions can be called through these tags. This means that whenever a certain grammar node or word is reached, its respective Java function will be called. This actually sets the basis of interaction between the grammar layer and application layer.



```
<digits> = one {this.$value = "1"}
  | two {this.$value = "2"}
  | three {this.$value = "3"}
  | four {this.$value = "4"}

 <root> = (square root
   |radical
   |root){this.$value = "\\sqrt\{\}"}[of]
   {appObj.addAtCursor(this.$value);};

<equation> = <number> <op> <number> equals <number>;
```

Fig. 2. JSGF Example

The application layer keeps track of the total equation in a string and a cursor position of the equation. The grammar layer is responsible for the recognition of individual speech segments. It specifies within one continuous segment what is allowed to be said with no knowledge of what has been previously said. Once the decoder and grammar layer have recognized the speech into words, the result is passed on to the application layer by triggering the respective Java functions of these words. The application layer employs it's previous knowledge of what was said and intelligently places the decoded result in the total equation making sure the entire equation string remains latex compilable.

2) Comprehensive Grammar: Our initial approach for the math grammar was to specify a comprehensive grammar. The grammar search graph defined allowed for long math equations with math restrictions such as: integral from a to b open x squared plus y squared close d x. Whilst this is a valid sentence, most users will not actually say such a long sentence at once. Furthermore, a well defined set of mathematical rules could cause the search graph to grow unnecessarily large in size and complexity if we were to allow every possible sentence in math to be said. This had its toll on recognition time which increased to several seconds, as well as on precision which went below acceptable rates.

3) Loose Grammar: An alternative simplistic approach was to have a loose grammar where the list of allowed statements are short and limited to the succession of few words. For example, '<number> over <number>' and '<number> plus <number>' were placed in two completely separate grammar nodes each calling its respective different Java function. Whilst the separation makes it easier for the Java function implementations and latex value assignments, the fact that both nodes could begin with the same word allowed for confusion. Assume the user started out by saying 'one' followed by some other words. Both grammar nodes would have to be loaded and traversed before figuring out to which grammar node of the two the word 'one' belongs. Having to check all possibilities before deciding on an answer was redundant and wasted time. Recognition time also increased unacceptably in this situation to several seconds.

4) Lazy Grammar: As a middle way solution, what we called the Lazy Grammar was defined in such a way that all entry points in the grammar nodes are distinct. Once the first word in a speech segment is identified it will have only one possible tree to traverse hence avoiding the confusion. Upon adopting this approach, both the precision and processing time for recognition improved significantly. Remarkably, the processing delay went down from few seconds long to real-time.

B. Training the System

As with any speech recognition system, creating a knowledge database which produces reliable learning, the so called training phase, requires considerable time and patience. Utterances have to be recorded several times, under different situations (different background noise), and using different speakers (different tones, pitches, accents). To aid and streamline the training process, we developed and added several tools to Mathifier. One important tool is a Java program that takes a text file as an argument and displays each line sequentially. The trainer would speak the line and hit enter, in which case the program automatically creates the corresponding .wav file and displays the next line in the file. This makes it much easier to train the file in a shorter amount of time.

C. Real-Time Recognition

Users will use this system in order to get rid of writing equations on paper or typing them on screen. This means that the user will speak equations that are in their mind, and not on paper. Hence, it is necessary to show the user the result of their speech while they are speaking, and not only when the sentence is finished. After all, a user cannot keep a whole complicated equation in mind and be able to say it all in one shot. In the original Sphinx, however, this is not the case. CMU Sphinx recognizes a sentence, which is an utterance bounded by two moments of silence. Each time a complete sentence is uttered, Sphinx prints out the best path. We overcame this limitation by implementing a real-time recognition feature, where the best path of the active list is printed every time a feature is processed. While the implementation worked well, the overhead is large since the best path is computed at every feature. We intend to optimize real-time recognition by computing the best path only when a completely new word is recognized.

D. User Interface

Two threads run simultaneously in our system. A user interface thread runs in the foreground, displaying controls and results by using the Java Swing library in a pleasant environment. The Sphinx speech recognizer runs in a background thread, listening to voice input and decoding the speech signals to text. It was shown in that the grammar language can convert english words to Latex code on-the-fly. Still, it is not pleasant to the user to show plain Latex code of what was recognized. Instead, compiled Latex code should be shown, where the equation is in easily understandable mathematical form. A first take on this problem was to use system calls to invoke the Latex compiler. The overhead in this case is considerably large since the OS had to perform several context switches, in addition to exchanging information using files. This overhead amounted to more than 200ms for every compilation of a single line of Latex code. A better solution that we will implement is to use an open-source Latex Java API called JLatexMath. By using the API, system calls are replaced by lighter interactions between Latex and our system. Also, data would be exchanged using buffers instead of the filesystem.

E. Program functionalities

Program functionalities were developed in accordance with what's user friendly and intuitive when speaking math. It utilizes pauses between words to predict what the user meant. For example assume the user says "a over b", pauses, and then says "squared". This power will be interpreted as belonging to the entire fraction. If the user says directly "a over b squared" then the power will be interpreted as belonging to "b" strictly. Useful commands have also been implemented to ease the formation of the equation such as "close everything". If the user has opened several brackets in the equation, this command will correctly count all unclosed brackets and close them all.

IV. Results & Future Work

After considerable training, we performed tests to gain knowledge about the error rate in our program. Two tests were performed: one that finds the error rate for speakers who trained the system, and another for speakers who did not train the system. Three persons trained Mathifier, so the same three were used in the first test. For the second test, we chose three different persons who did not train Mathifier. For speakers who trained the system, the WER (word error rate) was 16.3%. In fact, the system missed 1571 out of 9620 words. For speakers who did not train the system, the WER was slightly higher, at 21.0%. The system missed 2194 words out of 10455. The results are summarized in Figure 3. The testing process showed good results, especially since the word error rate for speakers who did not train the system was not considerably higher than those who did train the system.

	WER	**Words Correct**	**Words Incorrect**
Trained	16.3%	8049	1571
Untrained	21%	8261	2194

Fig. 3. Results

The presented system was implemented and a knowledge database for training is currently being developed. Due to the large lexicon of what math users could find useful, the process of building this database requires significant time (not withstanding the user-friendly trainer that mathifier includes). Thus far, we have trained numbers between zero and one hundred. The process is time consuming because, after composing a transcript (list of sentences to train the system with) and training the system, the acoustic model created has to be tested by using it in the recognition process. For instance, we noticed that after some preliminary training, the recognizer had a high error rate in resolving utterances that are acoustically very similar. For example, high errors occured when resolving multiples of ten, exchanging thirty with thirteen, forty with fourteen, etc. In classification terms (HMM), one could consider that the knowledge database had not sampled the probability distribution sufficiently to build fine models of sound feature transitions. To remedy this issue, manual intervention is needed. Here, another transcript was devised with more stress on the words experiencing a higher error rate. The re-training of the system gave much better results. It is clear then that it is necessary to train the system by introducing small batches of new words and stressing on those with higher error rates.

As a demonstration of the utility of the lazy grammar that mathifier introduces, the system did not experience the same high error rate when the speaker said a 2-digit number that is not a multiple of ten (ie, seventy two) even with the small database currently compiled. This is due to the grammar model shown in Section III-B. The model specifies that a digit cannot follow a number between 10 and 19, but it could follow larger 2-digit numbers. Hence, when the system has a similar acoustic weight to both thirty and thirteen, and the following word is three, then the system will go with thirty.

In addition to sampling the math lexicon finely, the knowledge database should also include a rich enough variation on pronunciations from multiple speakers in different environments. That translates to lower accuracy for novel users whose pronunciation profile do not closely math those in the database.

V. Conclusion

Handwritten text is slowly losing favor to word processors, and converting speech to digital math equations goes in the same direction. The specificity of our domain, the math language, has an advantage over general speech recognizers, since it has a specific application that is not met by general recognizers. Additionally, the restriction to the math domain offers better accuracy and faster results. The applications for our product are interesting. Naturally, the product can be used to dictate math equations and have them converted in realtime. The product can also be used in lower education, where young children would know numbers verbally. By using Sphinx, these children can practice writing equations by mapping how equations sound to how they look. In the professional field, our product would be very useful as an add-on to latex. Professionals spend a lot of time meddling with different latex functions, and have to keep up with long equations in textual form. With the addition of a speech recognizer, professionals can write these equations on latex more naturally. In this project, we are using a successful, research-driven, and open-source software, and we are trying to build upon it. The scope of our project is the building of a complete product that benefits everyone. We have adhered to constraints that target the everyday user by making it intuitive and easy to use.

References

[1] http://mathtalk.com/, "Mathtalk," 2011.

[2] P. Senin, "Dynamic time warping algorithm review," December 2008.

[3] B.-H. J. Lawrence Rabiner, *Fundamentals Of Speech Recognition*. Prentice Hall, 1993.

[4] http://java.sun.com/products/java media/speech/forDevelopers/JSGF/, "Grammar format specification," 2011.

[5] P. Lamere, P. Kwok, W. Walker, E. Gouva, R. Singh, B. Raj, and P. Wolf, "Design of the cmu sphinx-4 decoder," *MERL – A Mitsubishi Electric Research Laboratory*, August 2003.

[6] B. Raj, "Bhiksha's notes," these are unofficial notes prior to the implemention of Sphinx 4.

[7] P. Lamere, P. Kwok, W. Walker, E. Gouva, R. Singh, B. Raj, and P. Wolf, "The cmu sphinx-4 speech recognition system."

[8] S. Young and N. Ressell, "Token passing: a simple conceptual model for connected speech recognition systems," *Cambridge University Engineering Department*, 1989.

[9] M.-Y. Hwang and X. Huang, "Subphonetic modeling with markov states - senone," *Camegie Mellon University*, 1992.

[10] http://cmusphinx.sourceforge.net/wiki/tutorialam, "Training specifications," 2011.

[11] http://www.adobe.com/products/acrobat/adobepdf.html, "Adobe portable document format," 2011.

[12] The cmu pronouncing dictionary. [Online]. Available: http://www.speech.cs.cmu.edu/cgi-bin/cmudict