

# MailTalk

## Abstract

With the goal of improving users' productivity, we use the Sphinx-4 speech recognition library to enhance Mac OS X's email client, Mail, with a speech-based command interface. We find that under Sphinx-4's baseline configuration, command accuracy (CAcc) varies from 89-99% in our controlled tests. After tuning, we improve accuracy by up to 3 percentage points, to a CAcc of 92-99%. We conclude that while the productivity gains are not likely realized in this implementation, future work holds the promise of doing so. We also contribute a novel extension to the Sphinx-4 API to automate testing using WAV files as input.

## 1. Motivation

According to a 2001 Gartner survey, the average office worker spends 49 minutes per day managing email—about 10% of the workday [1]. Regarding reach, a 2009 Nielson study finds 65.1% of the world's population uses email [2]. In short, email management is pervasive and time consuming. Despite this, the way people interact with email has changed little since its mainstream rollout in the 1980s: we still use a keyboard and pointing device for input, and a display for output—sometimes along with a “new mail” sound. We ask: is there a better way? We hypothesize that adding new input and output modalities—namely speech input and synthesized speech output—will boost email management productivity. Even a small productivity improvement has the potential to, in aggregate, save millions of minutes of effort per day.

## 2. System Capabilities

The MailTalk system we devise works with Mac OS X's native Mail application. The system has no explicit user interface (UI). Instead, the user works with Mail directly. In addition to the normal modes of operation via the keyboard and mouse, the user may leverage the extra input modality provided by MailTalk: speech. By simply speaking commands, such as “delete,” the MailTalk system interprets the command and coordinates with Mail to perform the desired action. Further, MailTalk repeats the command to the user through synthesized speech, providing verbal confirmation of the action, which, of

course, is synchronized, with the visual effects of the action (e.g. the email disappearing). Thus, MailTalk’s enhancements to the email client are seamless and invisible, but audible.

Unlike other systems, such as MIT CSAIL’s City Browser research prototype [3] built using the WAMI speech recognition library [4], MailTalk extracts commands from background noise and non-relevant speech automatically. It does not require a specific user interaction to start or stop recording. This, too, provides a better user experience, though at the risk of misunderstanding the user’s true intentions. Continuous speech support is provided natively by the Sphinx-4 library through the live mode decoder, consisting of speech classifier, speech marker, and non-speech data filter components.

## 2.1. Input

Input is in the form of speech. We describe the grammar in Listing 1 below, in Java Speech Grammar Format (JSGF) format [5]. JSGF is a standard speech recognition grammar format created by Sun Microsystems (now Oracle). In the grammar, we provide flexibility for users to say a command in multiple different ways. We believe this aids usability, since the user does not need to become trained in a particular style of command—such as whether an email is called an *email*, or *mail*, or a *message*. For example, to invoke a “compose message” command in the Mail client, the user may utter any of the following phrases:

- New
- New mail
- New mail message
- New email
- New email message
- New message
- Compose
- Compose mail
- Compose mail message
- Compose email
- Compose email message
- Compose message
- Create
- Create mail
- Create mail message
- Create email
- Create email message
- Create message

Many of these variations are defined succinctly using the `<message>` definition in the grammar, which reflects the various ways a user may refer to an email message. Also notable in the grammar is the definition of `<folder>`. While the grammar itself is *closed*—meaning words not specified in the grammar are not recognized—the definition of a “folder” is defined at run time based on the actual folders MailTalk discovers in the email client. If the user has folders named *Personal*, *Work*, and *Junk*, for example, the grammar rule would update dynamically to be defined as “`personal {/account@domain/personal} | work {/account@domain/work} | junk {/account@domain/junk}.`” Per the JSGF specification the content within the curly braces of the definition indicates the semantic meaning associated with the word

or phrase; so for mail folders, the curly brace content is the unique path describing the location of the folder.

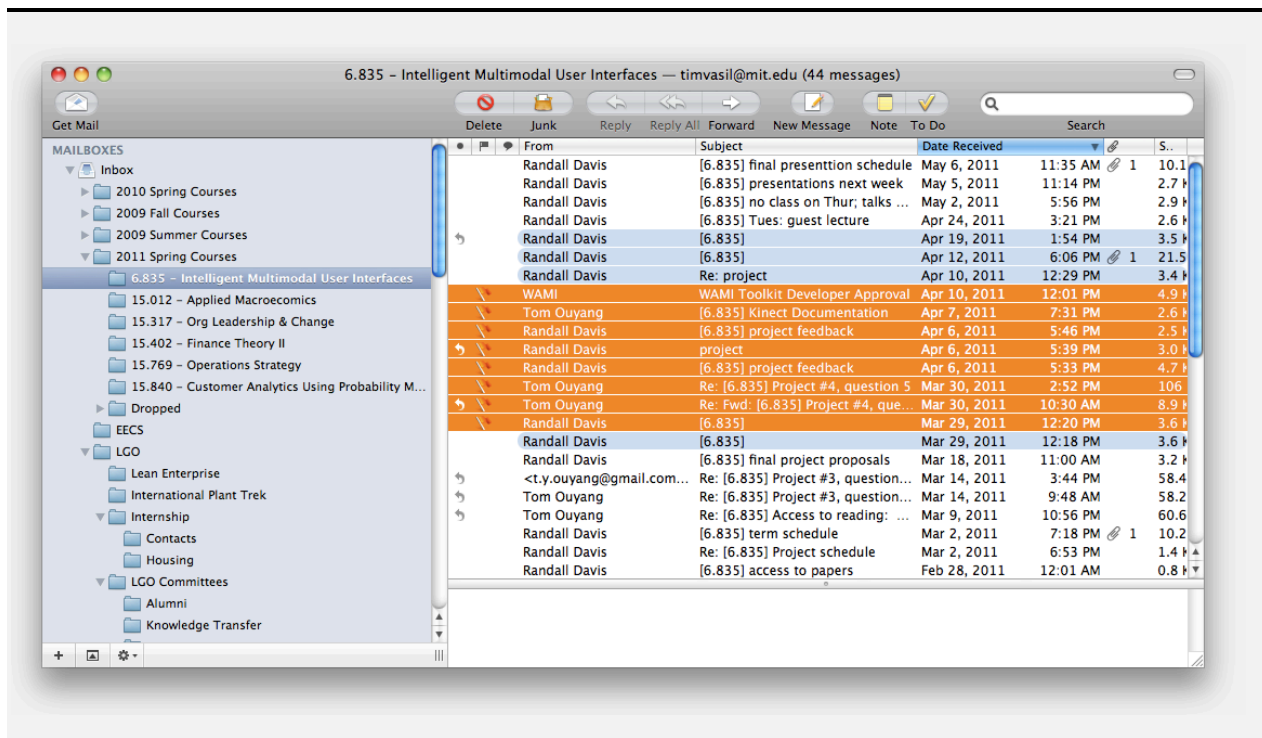
**Listing 1:** MailTalk's speech recognition grammar defining all supported commands, in JSGF format.

```
1 #JSGF V1.0;
2 /**
3  * MailTalk grammar
4  */
5
6 grammar mailtalk;
7
8 /** Reusable definitions */
9 public <folder> = l g o eleven {lgo11} | team six {team6};
10 public <message> = mail [message] | email [message] | message;
11
12 /** Commands */
13 public <new> = ((new | compose | create) [<message>]) {new};
14 public <open> = (open [<message>]) {open};
15 public <view> = view {viewFolder} <folder>;
16 public <read> = ((reed | speak) [<message>]) {read};
17 public <stop> = stop {stop};
18 public <undo> = (un do | oops) {undo};
19 public <redo> = re do {redo};
20 public <next> = next [<message>] [in thread] {next};
21 public <prev> = (previous [<message>] [in thread] | back) {prev};
22 public <mark> = <NULL> {mark} [mark [<message>] [as]]
23     (
24         red {read} | un red {unread} |
25         flag {flagged} | flagged {flagged} |
26         un flag {unflagged} | un flagged {unflagged} |
27         spam {spam} | junk {spam}
28     );
29 public <delete> = ((delete | trash | throw away | discard) [<message>]) {delete};
30 public <selectThread> = (select thread) {selectThread};
31 public <moveTo> = [always] {createMoveRule} (move [<message>] to) {moveTo} <folder>;
32 public <copyTo> = [always] {createCopyRule} (copy [<message>] to) {copyTo} <folder>;
33 public <forward> = forward {forward};
34 public <reply> = (reply [to <message>]) {reply};
35 public <replyAll> = (reply [to] (all | everyone)) {replyAll};
36 public <add> = add {add} (
37     sender {sender} |
38     recipients {recipients} |
39     (c c [recipients]) {CCs} |
40     (b c c [recipients]) {BCCs} |
41     all {all}
42 )
43 [of <message>] [to (address book | contacts)];
```

## 2.2. Output

MailTalk provides two types of output: instructions to the Mail client to take specific actions (based on the interpretation of voiced commands by the user), and synthesized speech to notify the user of its interpretations. This second channel of output confirms to the user what is happening, which particularly useful in situations where the screen does not unambiguously convey this information. For example, the delete command visually manifests itself as selected email disappearing from the active view; however, moving emails to a different folder has the same visual manifestation. Figure 1 shows an example effect of a voiced command on the Mail user interface.

**Figure 1:** Sample output of MailTalk. Here, MailTalk tells the Mail email client to flag the selected emails based on the “Mark as flagged” (or simply “Flagged” or “Flag”).



## 3. How it Works

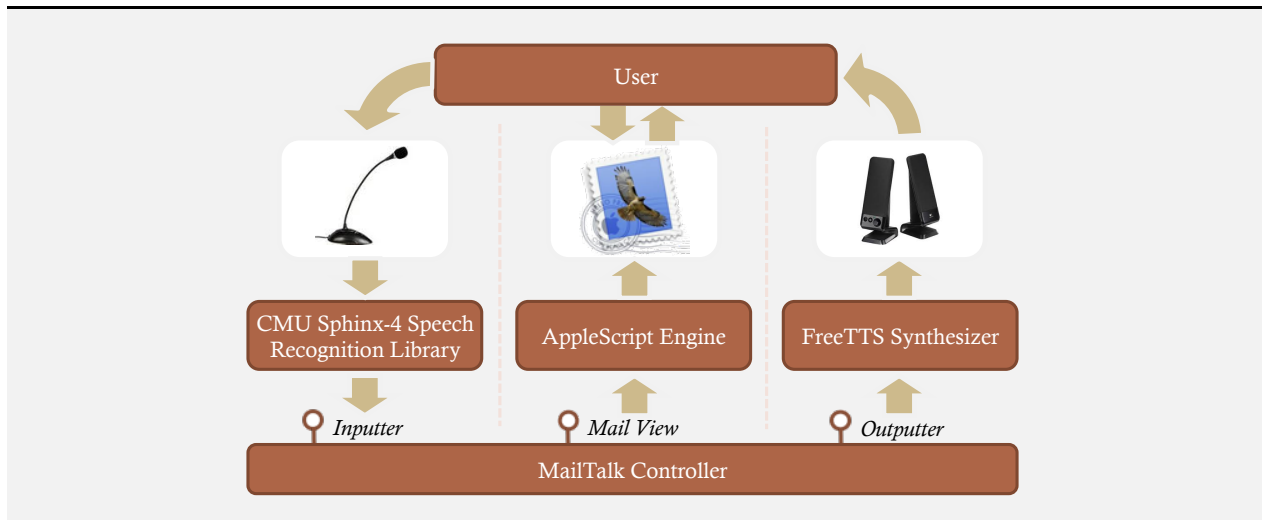
Written in a combination of Java and AppleScript, MailTalk performs speech recognition using Carnegie Mellon University’s open source Sphinx-4 speech recognition library [6] and speech synthesis using FreeTTS speech synthesizer [7]. The complete listing of technologies is detailed in Table 1.

**Table 1:** Technologies upon which MailTalk is based.

Category	Technology	Version
Programming language	Java / Java Virtual Machine (JVM)	1.6
Operating system	Mac OS X	10.6.7
Email client	Mac OS X's Mail application	4.5
Email client integration	AppleScript	2.3
Speech recognizer	CMU Sphinx-4 Library [6]	1.0 (beta 6)
Speech synthesizer	FreeTTS [7]	1.2
Command line parsing	JOpt Simple [8]	3.2

### 3.1. Architecture

**Figure 2:** Architectural overview of the MailTalk system. Arrows indicate the flow of information.



The architecture of the MailTalk system is based on a design principal of functional abstraction through generic interfaces. As shown in Figure 2, the foundation of the system is the MailTalk controller, which interacts with subsystem implementations through three interfaces:

- Inputter (`IInputter` Java interface): Provides notifications of interpreted mail events using the well-established listener design pattern; here, the Sphinx-4-based speech recognizer serves as the current implementation, swapping it with another type of recognizer or even a new input modality such as a sketch recognizer is straightforward.

- Mail View (`IMailView` Java interface): Provides a means of executing the interpreted commands. The current MailView implementation issues AppleScript commands to automate tasks in the Mail email client. As with the inputter, swapping this implementation with another—designed for, say Microsoft Outlook—is again straightforward.
- Outputter (`IOutputter` Java interface): Notifies the user of a command as it executes. MailTalk implementations include the FreeTTS-based speech synthesized output and console output (for debugging and testing).

## 3.2. Command Line

MailTalk is started via the `main()` method of the `com.timvasil.mailtalk.MailTalk` class. Three optional command line arguments are available, described in Table 2.

**Table 2:** MailTalk’s command line arguments.

Command	Short Form	Explanation
<code>--help</code>	<code>-h</code>	Provides information on available command line arguments
<code>--noeffect</code>	<code>-n</code>	Indicates the mail commands should not be executed. The Mail View implementation is replaced with a “dummy” one that takes no action.
<code>--test [path]</code>	<code>-t [path]</code>	Plays the wave (.wav) files in the specified <i>path</i> , analyzing whether each results in a correct interpretation (hit), incorrect interpretation (substitution), or no interpretation (deletion), and whether there are any spurious commands issued (insertions). Results are written to the console in real time, and after all WAV files have played, overall statistics and command accuracy (CAcc) information is also written to the console.

### 3.2.1. Interactive Mode

To run the application, we recommend a maximum heap size of at least 256 MB, as the Sphinx-4 recognition library requires more memory than allocated by default to perform recognition with our grammar. A sample command line to run MailTalk in interactive mode is shown in Listing 2. Naturally, you will need to ensure your system’s microphone is enabled and set to an volume level such that your voice is discernable from background noise and does not get distorted even when you speak loudly. To terminate MailTalk, hit `<Ctrl>+C`.

**Listing 2:** Command to run MailTalk in interactive mode.

```
1 cd MailTalk/bin
2 java -cp ../lib/* -mx256m com.timvasil.mailtalk.MailTalk
```

### 3.2.2. Test Mode

To run a test based on prerecorded sound files, use the `--test` parameter, as in the samples shown in Listing 3. Note that for a test, the WAV files must follow a naming convention so MailTalk can discern ground truth, i.e. the correct interpretation of the sound as command. To do so, either choose one of the `com.timvasil.MailTalk.MailCommand` enumeration fields as the name, or append—in parenthesis—one of these fields, along with a parameter, if necessary. An example of a valid name is “Add sender to address book (Add\_Contact Sender).wav.” Here, a description of what is spoken precedes the actual command in parentheses—the `ADD_CONTACT` command with an argument of “sender,” which is followed by the file extension. Case is insensitive but the extension is required.

**Listing 3:** Commands to run tests with 45 recorded commands spoken by Alex, and 49 recorded commands spoken by Tim, respectively.

```
1 cd MailTalk/bin
2 java -cp ../lib/* -mx256m com.timvasil.mailtalk.MailTalk --test ./TestCommands/Alex
3 java -cp ../lib/* -mx256m com.timvasil.mailtalk.MailTalk --test ./TestCommands/Tim
```

## 4. Performance

To gauge performance, we use a standardized test suite of 45 prerecorded commands from one speaker, Alex, and 49 prerecorded commands from another speaker, Tim. We perform tests across a variety of environmental conditions, namely foreground and background noise levels, and Sphinx-4 parameter settings. In evaluating the results, we use the metric of *command accuracy* (CAcc). We define CAcc as  $(N - S - \frac{1}{2}D - I)/N$ , where  $N$  is the number of commands tested,  $S$  is the number of substitutions,  $D$  is the number of deletions, and  $I$  is the number of insertions.

$$\text{CAcc (Command Accuracy)} = 1 - \text{CER}$$

$$\text{CER (Command Error Rate)} = (S + \frac{1}{2}D + I) / N \approx \text{WER}$$

Our CAcc metric is inspired by the definition of word accuracy (WAcc), though its focus is at a different level. We find *command*-level more suitable than *word*-level accuracy as our system is ultimately executing commands, not transcribing speech, so any substitution, deletion, or insertion within a command yields an incorrect (substitute) command. Unlike WAcc, we weight deletions less heavily than substitutions and insertions since it has a preferable effect. A substitution or an insertion results in an unin-

tended command being executed—such as a delete—while a deletion has no adverse side effects. We choose the weight of 0.5 based on Hunt’s own adaptation of WAcc, which weighs both deletions and insertions at 0.5 [9]. Hunt’s reason for weighting these permutations is to more accurately compare similar recognitions systems, which is like our goals of comparing similar configurations of a system.

In the subsequent sections, we describe the results of a variety of tests:

- Baseline configuration (with two different speakers),
- Environmental effects: Examining varying levels of background/foreground noise, and
- Parameter tuning: Background sensitivity, vocabulary size, out-of-grammar probability, and cutoff threshold.

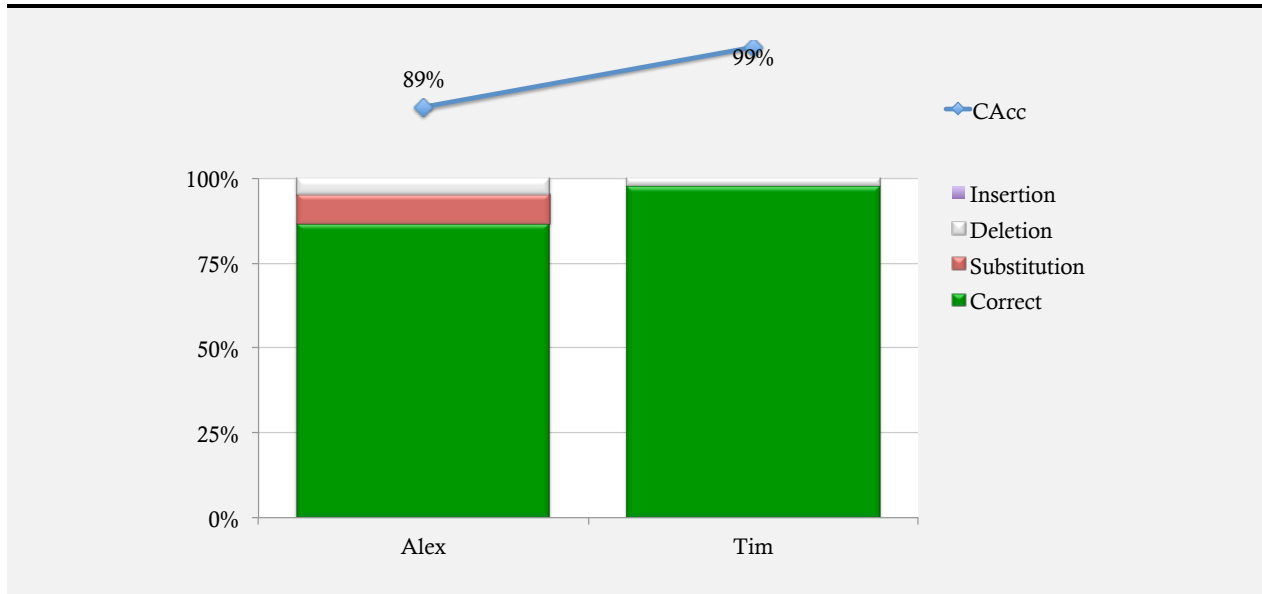
### 4.1. Baseline Configuration

For our baseline tests, we use the default configuration of Sphinx-4 as provided by Sphinx’s *helloworld* demo. We show the relevant portions of this configuration in Table 3 and the results of the tests in Figure 3. The system performs very well with Tim’s commands, yielding a 99% CAcc. With Alex’s commands, CAcc is only 89%; the lower score results primarily from several substitutions. We believe the system performs better with Tim’s commands since Tim’s commands are spoken more clearly and articulated a bit more deliberately, making the speech less ambiguous. As the programmer of the system and one who has an understanding of speech recognition algorithms, Tim may have deliberately spoken in this manner to ensure a good result. Alex, on the other hand, has no experience with speech recognition technology and no knowledge of the MailTalk implementation. We believe the CAcc seen with Alex’s commands, therefore, more accurately estimates the performance we would expect with a typical user with baseline parameter settings.

**Table 3:** Subset of the parameters used in the baseline tests. The values are taken from Sphinx’s *helloworld* demo.

Category	Baseline setting	Parameter	Value
Vocabulary size	Digits	wordInsertionProbability	1E-36
		languageWeight	8
Background sensitivity	Average	threshold	13
Out-of-grammar probability	None	outOfGrammarProbability	0
		phoneInsertionProbability	0
Cutoff threshold	None	---	---

**Figure 3:** Baseline performance.



## 4.2. Environmental Effects

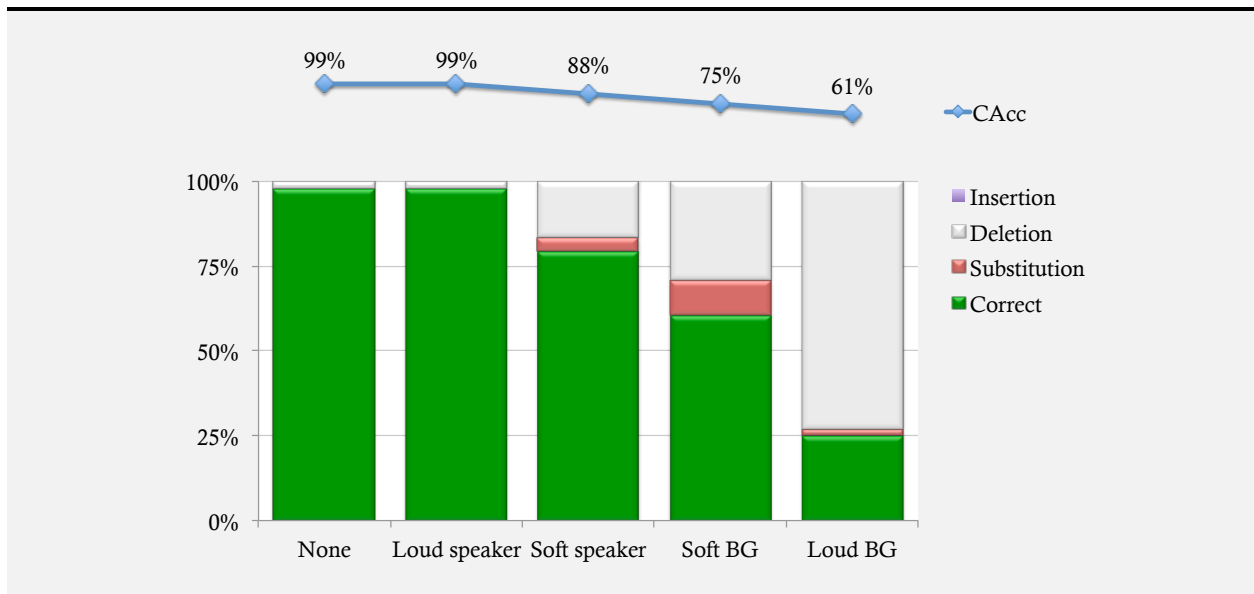
To test the robustness of the system, we varied the levels of foreground noise and background noise. The results, plotted in Figure 4, are from five scenarios:

- None: Baseline scenario, with Tim as the speaker
- Loud speaker: Recorded commands played back 33% louder.
- Soft speaker: Recorded commands played back half as loud.
- Soft BG: Recorded commands played at normal volume concurrently with low-volume dance music playing in the background (with lyrics).
- Loud BG: Recorded commands played at normal volume concurrently with the same music playing in the background, but at moderate volume.

MailTalk's performance does not degrade with the *loud speaker* scenario, since at this new volume there are not significant levels of distortion. It does degrade with the *soft speaker* scenario as the voice commands become faint enough to start being confused with background noise. And, not surprisingly, the scenarios with background noise yield further degradation proportional to the volume of the noise. Our conclusion: with the out-of-the-box configuration, the system performs very well in quiet environments; the microphone volume must be set sufficiently high to detect commands, but there is significant latitude past a threshold. Introducing background noise, such as music, degrades performance both in

substitutions and deletions. Substitutions are more prevalent at lower volumes since they confuse the speech recognizer but not enough to shake its confidence in the result. In contrast, with loud background noise most commands (75% in our test) are simply ignored, as they cannot be differentiated from the din.

**Figure 4:** Performance with varying foreground and background noise levels.



### 4.3. Parameter Tuning

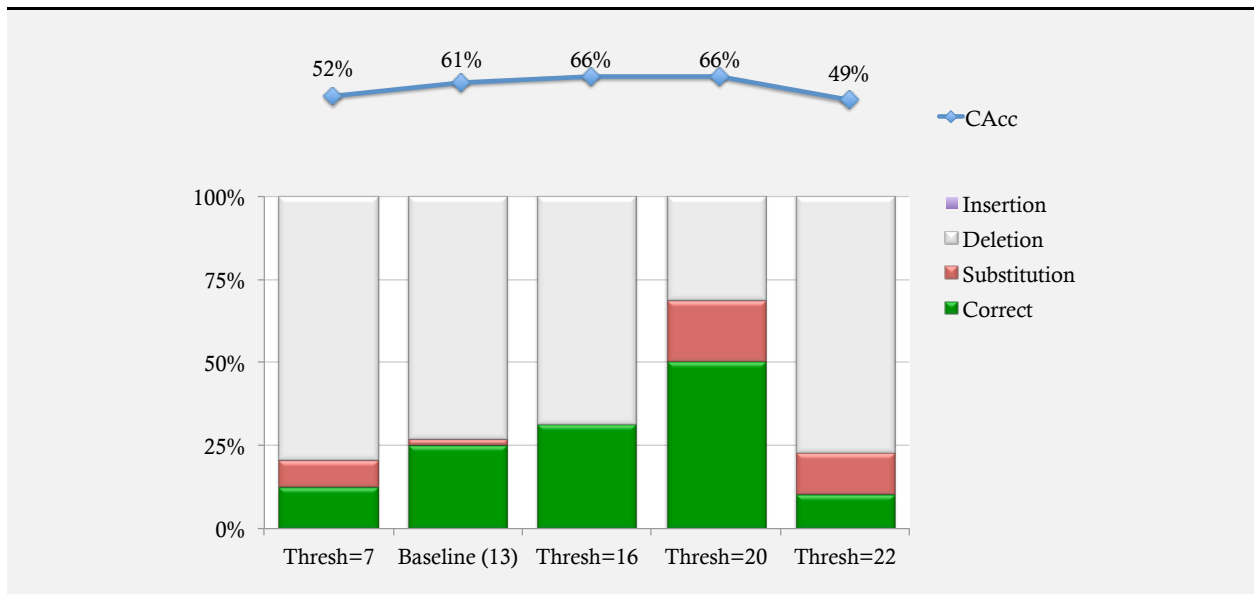
In this section we explore whether we can tune Sphinx’s default parameters to improve performance. While tuning is labor-intensive and a bit cumbersome, we see it widely adopted by the Sphinx user community, to the point where common tunable parameters are called out specifically in the stock configuration files. Some researchers, such as Vertanen, have even worked on tuning “recipes” [10]. Since our baseline performance is quite good, we look to tuning to improve CAcc in the face of background or foreground noise.

#### 4.3.1. Background Sensitivity

We begin with the *threshold* parameter, which controls the sensitivity to background noise. A lower value indicates a smaller volume difference is used to delineate foreground noise from background noise. We explored the effects of various threshold settings below and above the baseline value of 13, some of which are plotted Figure 5. Importantly, we perform these tests with loud background music is playing; this situation has poor baseline performance, so improvements are easy to detect. We find CAcc

peaks in the range of 16-20, then drops of quickly. Past this point, we believe the threshold is too great for any foreground noise to stand out against the background.

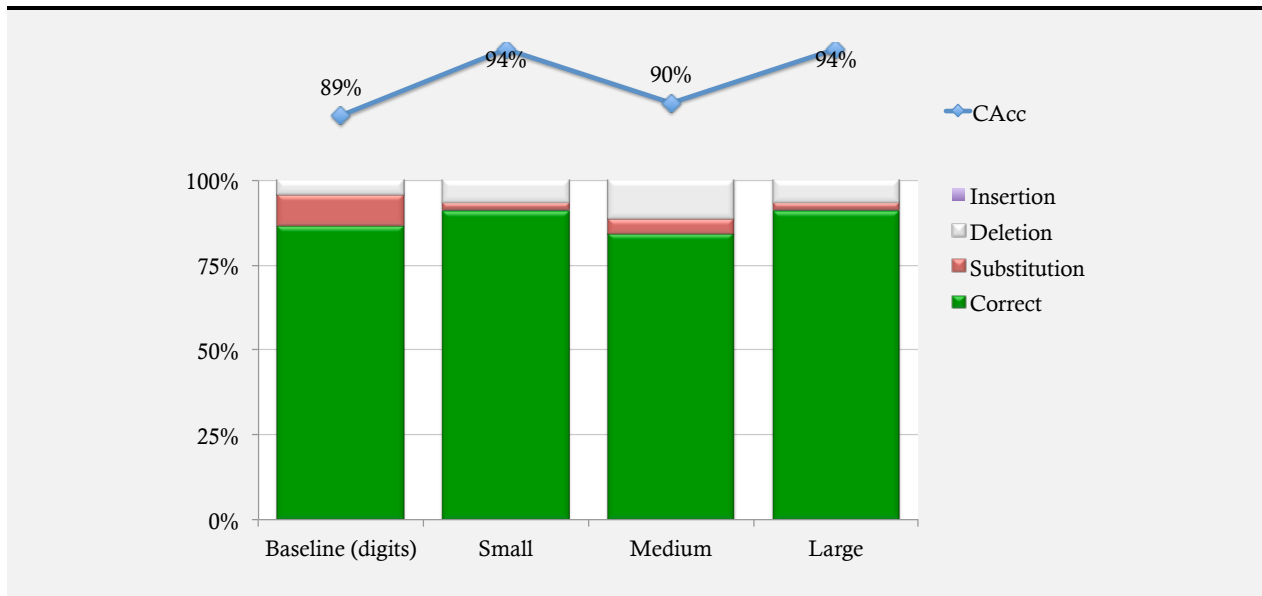
**Figure 5:** Performance with various background sensitivity settings.



#### 4.3.2. Vocabulary Size

Sphinx-4 exposes two parameters to specify the expected size of the vocabulary: *wordInsertionProbability* and *insertionWeight*. We explore permutations of these settings in logical groupings of digits, small, medium, and large vocabularies as defined in the Sphinx programmer’s guide [11]. Results are plotted in Figure 6. We do not have much insight into how these parameters tune Sphinx’s linguist, but the small and large settings perform best. We speculate they work well because the “small” setting (targeting about 80 words) matches the vocabulary size of our grammar, while the “large” setting (targeting 64,000 words) is more in tune with the size of the Wall Street Journal dictionary and acoustic model we used.

**Figure 6:** Performance under varying vocabulary size settings.

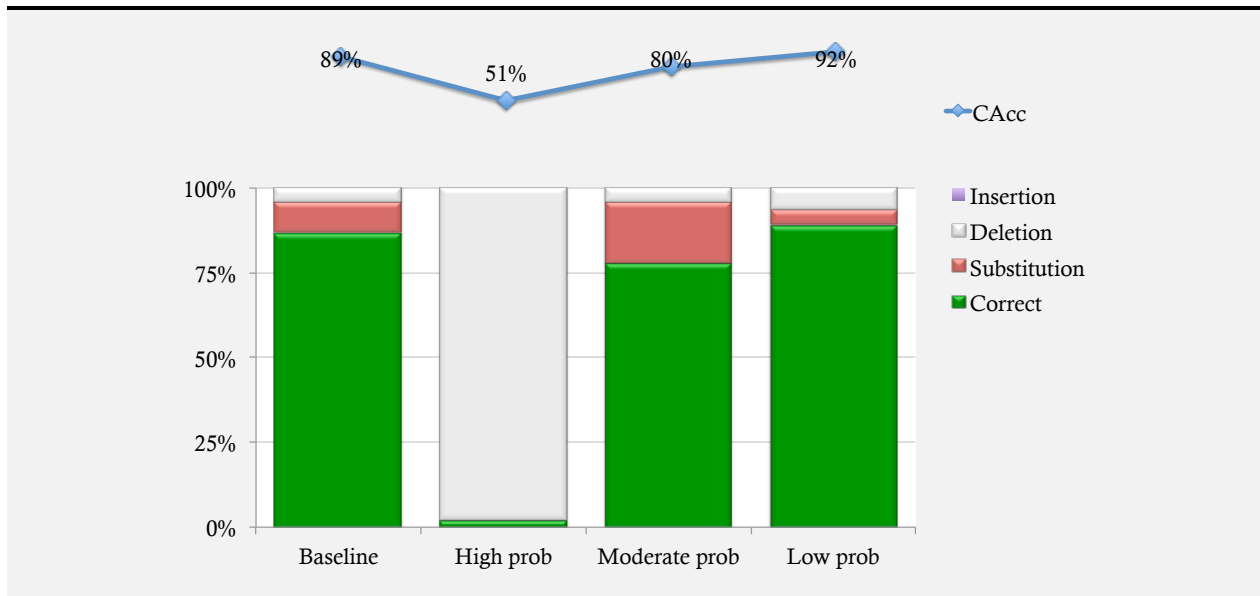


### 4.3.3. Out-of-grammar Probability

Sphinx-4 recognizes speech using a closed grammar, which is why we go through the trouble of updating the grammar with a user’s actual mail folder names. Consequently, the recognizer may attempt to force-fit out-of-grammar words (like “zebra”) into something present in the grammar (like “delete”). With two parameters, *outOfGrammarProbability* and *phoneInsertionProbability*, we can tell Sphinx-4 to expect some spoken words not to be present in the grammar, in which case MailTalk is able to simply discard the command. In our testing, we explored various values for these parameters based on the recommended ranges implied by the Sphinx-4 documentation. The results are shown in Figure 7.

We find that a high probability of out-of-grammar words (1E-6) yields a low CAcc, as the recognizer considers even valid, clearly articulated commands to be out-of grammar. At moderate probability (1E-16), we see substitution errors, mainly from missing parts of commands with multiple words. At low probability (1E-26) we improve over the baseline probability (0) as some less intelligible commands are no longer force-fitted into the grammar rather than simply being ignored. This improves CAcc, which favors deletion over substitution.

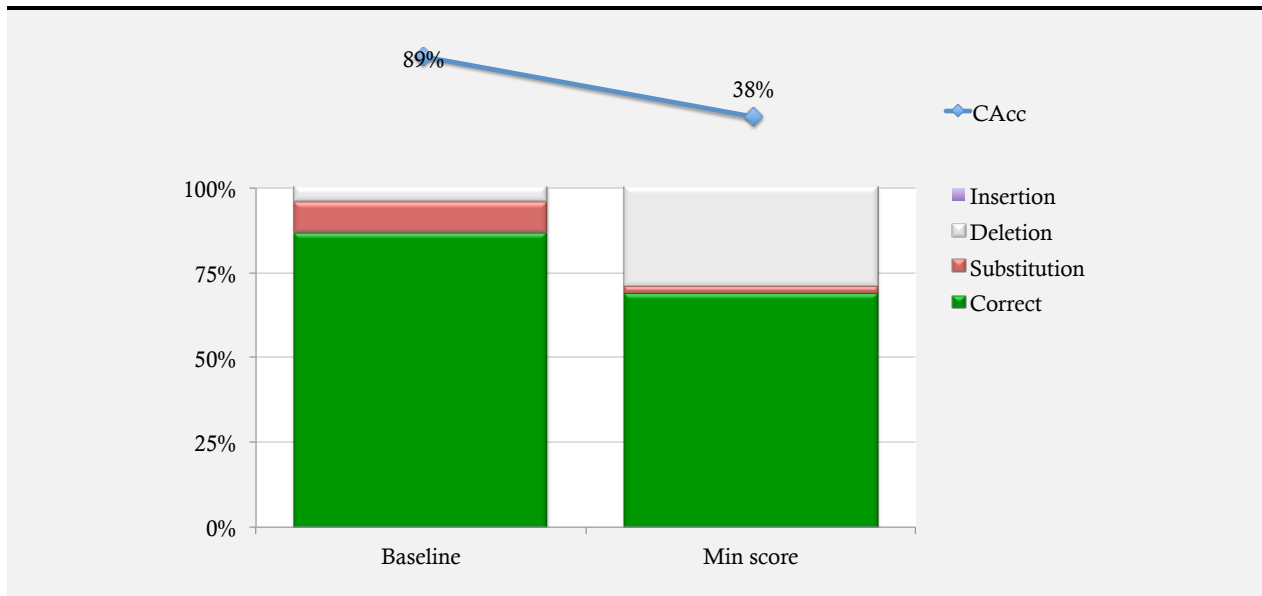
**Figure 7:** Performance under varying out-of-grammar probability settings.



#### 4.3.4. Cutoff Threshold

As a final parameter, we explore ignoring results below a particular recognition score, which is a combination of Sphinx's language and acoustic scores, as provided by the `getScore()` method of the recognized token. The results are in Figure 8, showing the threshold utterly ineffective. Even with a very low threshold of  $-2.7E7$  (which approximates the score associated with some substitution errors), many of the correct recognitions are also pruned, dropping accuracy of Alex's spoken commands from 87% to 38%. We conclude that the score is not very meaningful. Since we are using a flat linguistic model, rather than a trigram model, the recognizer has no particularly good insight into the likelihood of the phrase actually being uttered. While the trigram model would allow us to calculate a more informed confidence score, we do not believe such a model is not suitable for a closed grammar where the specific ordering of words is fixed.

**Figure 8:** Performance with and without a minimum score threshold.



#### 4.4. Optimized Results

Based on the univariate analysis of individual parameters we described in the previous section, we arrive at independently optimized settings, shown in Table 4. These settings are not necessarily globally optimal or even better than the baseline, since an interaction of two parameters may improve or degrade performance. The cutoff threshold, for example, is based on a score that may vary depending on the vocabulary size or background sensitivity specified.

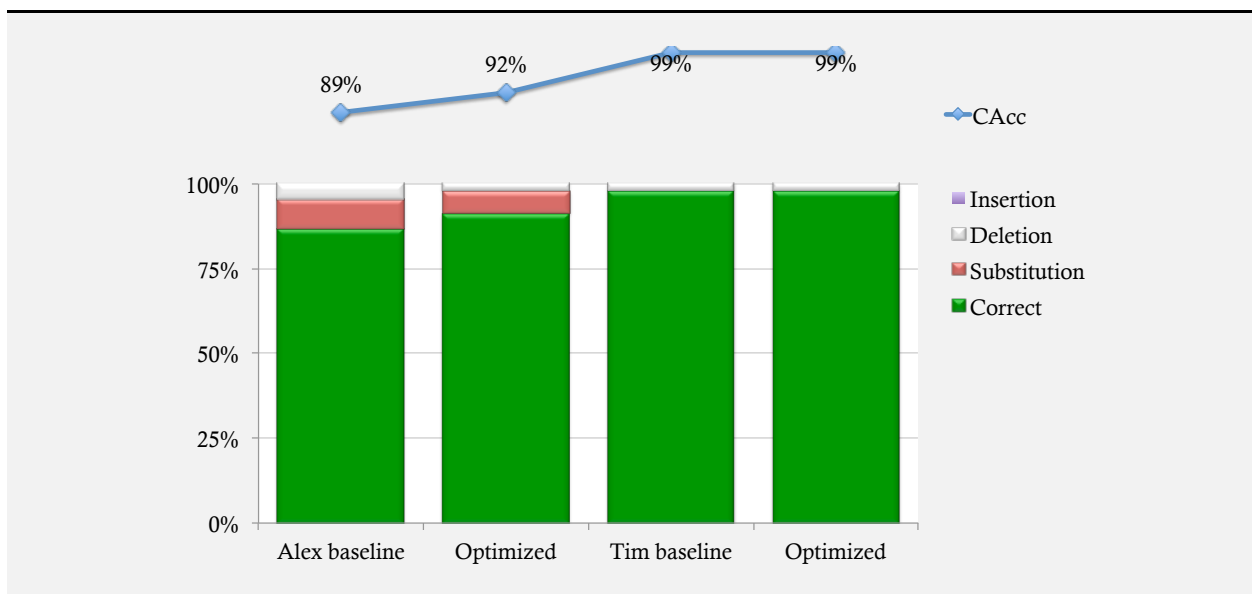
As it turns out, these settings do perform better than the baseline, as shown in

Figure 9, with the performance of Alex’s commands improved by 3 percentage points and the performance of Tim’s commands—already at 99% accuracy—unchanged. If performance had not improved with the new settings, we would be inclined to perform additional testing, perhaps using a design of experiments (DoE) approach. Whether a DoE or other technique would be able to find a “global optimum” is unclear and perhaps unlikely, as the ideal parameter values may vary based on speaker and environmental conditions anyway.

**Table 4:** Independently optimized parameter values based on our tests.

Category	Optimized setting	Parameter	Original Value	Optimized Value
Vocabulary size	Small	wordInsertionProbability	1E-36	1E-26
		languageWeight	8	7
Background sensitivity	High	threshold	13	18
Out-of-grammar probability	Low	outOfGrammarProbability	0	1E-50
		phoneInsertionProbability	0	0
Cutoff threshold	None	---	---	---

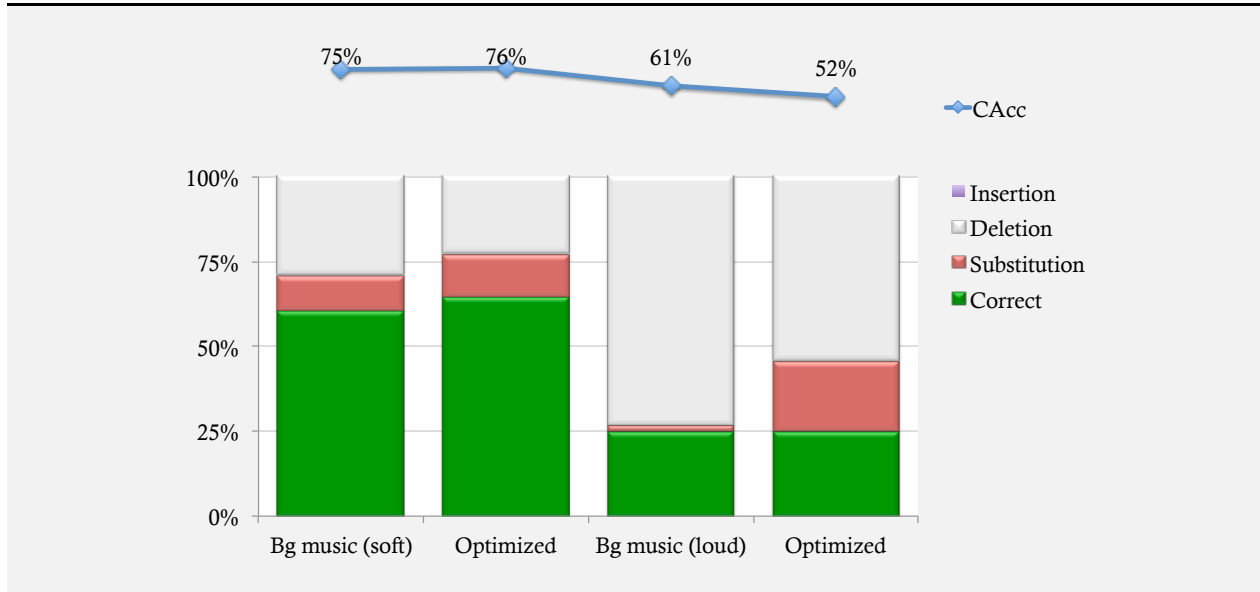
**Figure 9:** Performance of the optimized system compared to baseline in the controlled (quiet) environment.



When we examine how well the system performs under background noise conditions, we find that that, with soft background music, the optimized version improves upon the baseline by 1 percentage point. Unexpectedly, the optimized version performs 9 percentage points *worse* than the baseline for loud background music—all attributable to substitution errors. We summarize these findings in Figure 10. Loud background music is the one test case we find where the optimized settings perform worse than baseline. Since the settings individually perform better than baseline, we attribute the poor performance to an interaction among settings: perhaps the increased background sensitivity along with the increased

vocabulary size induced the recognizer to interpret noise from the music as valid syllables during spoken commands.

**Figure 10:** Performance of the optimized system with soft and loud levels of background noise.



## 5. Challenges

In developing MailTalk, most of our challenges rested in the speech recognition component of the architecture. We particularly grappled with Sphinx’s inclination to interpret any kind of continuous speech as commands, which we mitigated somewhat by enabling out-of-grammar recognition using parameter settings previously mentioned. Additionally, we discovered that speaker output interfered with the recognizer—not just synthesized speech, which initially wound our application into an infinite loop, but even seemingly innocuous beeps. We mitigated this errant behavior by clearing the microphone buffer after processing a command and immediately before listening to receive the next command.

We found the WSJ dictionary that ships with Sphinx missing some common words in email parlance such as “undo” or “inbox.” In the grammar, we worked around these problems by separating such words into parts, e.g. “un do.” Some words, like “read,” have multiple pronunciations, which we also clarified in the grammar, e.g. using “red” as the past tense of “to read” to mark an email as read.

We also found that Sphinx-4 lacked a critical API method to facilitate automated tests. With continuous recognition, there is method to interrupt a blocking `recognize()` call on a Recognizer instance. By default, then, if the recognizer fails to recognize a command (a deletion error), it will hang indefinitely, as opposed to timing out and continuing with the test. To interrupt the recognizer, we implemented a simple data processor that we inserted into the Sphinx-4 data processing stack above the microphone layer. This data processor will, after a specified wait period of three seconds, throw an exception in the appropriate thread so as to unblock the `recognize()` call and allow testing to proceed.

## 6. Conclusions

Under quiet conditions, we see MailTalk provide a tuned command accuracy (CAcc) of 92–99%, which is 3 percentage points above baseline performance. These measures are observed from automated tests using prerecorded commands from two different subjects and the internal microphone of a Mac laptop. Performance degrades with background noise, however, which makes the implementation impractical for most environments.

Based on our preliminary deployments with volunteers, we find that for more casual email users, MailTalk does not provide immediate advantages. While it does not take much time to learn—its commands are intuitive—it does not yet support enough variety of commands to satisfy these users. We found users speaking commands such as “send” and “close,” which we had not included in our grammar. For more advanced users, we believe the system offers some speedups, especially for folder lookups, though such users may derive even greater productivity gains through keyboard shortcuts, if the mail clients were to offer such functionality.

The main problem of the system is its substitutions, or false positives. We overheard one trial user say, “Oh, I didn’t know I had that folder!” Of course, a user could not have intentionally directed MailTalk to view that folder. Some false positives, such mistaking a “new” for a “delete,” are more severe than others. MailTalk in its current incarnation happily executes everything it recognizes immediately, without any type of confirmation. We believe either a confirmation step, where the user must say “yes,” may be appropriate for the more difficult-to-undo commands, or a more robust undo mechanism, or multimodal confirmation via a gesture such as a head shake, would make for a better user experience. Misinterpreted commands frustrated our trial users.

In addition to the MailTalk implementation itself, we offer a couple novel contributions to the Sphinx-4 user community. One is some infrastructure for automated audio-based unit tests, including a mechanism by which a blocking Sphinx-4 speech recognition call can be interrupted after a timeout period elapses. Another is the identification of good candidates for tunable parameters, and a simple approach by which optimal values may be discovered.

## 7. Future Work

We still believe in the potential for MailTalk or similar solutions to provide a much-needed productivity boost to the email management domain. For a system such as MailTalk to be effective, however, additional work is necessary. First on the list would be a more robust speech recognizer that limited substitutions and performed well even with background noise. If the system were simply to be aware of audio emanating from the speakers, such as music, it may be able to cancel out these sounds from the microphone to minimize these adverse effects.

Another important improvement would be advancements in the text-to-grammar processing we perform for folder names. Today, we feed the folder names into the grammar relatively unprocessed. Some words, such as “inbox” do not exist in the dictionary; nor do digits such as “11” or acronyms. A more sophisticated processor would convert these unknown characters into a valid grammar.

In our tests, we noticed the lack of authentication and authorization over the input. While it may be difficult for an adversary to hijack a user’s keyboard or mouse, he or she can without much trouble should a command such as “delete”—even from across the room—and wreak some localized havoc on the system. A nice enhancement here would involve speaker identification so that only the speaker’s voice would be recognized for the purpose of processing commands.

Finally, we leave some MailTalk commands, such as “add to address book” unimplemented, as the AppleScript coding is tedious and not terribly relevant to our core research interests. As future work, these commands could be implemented, or the entire mail view interface could be implemented in support of some other email client, such as Microsoft Outlook.

## References

- [1] Danielle Westling. (2001, April) Gartner. [Online].  
[http://www.gartner.com/5\\_about/press\\_room/pr20010419b.html](http://www.gartner.com/5_about/press_room/pr20010419b.html)
- [2] Mark Hachman. (2009, March) PCMag.com. [Online].  
<http://www.pcmag.com/article2/0,2817,2342757,00.asp>
- [3] Spoken Language Systems Group. MIT Computer Science and Artificial Intelligence Laboratory.  
[Online]. <http://web.sls.csail.mit.edu/city/>
- [4] Spoken Language Systems Group. MIT Computer Science and Artificial Intelligence Laboratory.  
[Online]. <http://wami.csail.mit.edu/>
- [5] (1998, October) Sun Developer Network. [Online]. <http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/>
- [6] Carnegie Mellon University. Sphinx-4. [Online]. <http://cmusphinx.sourceforge.net/sphinx4/>
- [7] FreeTTS. [Online]. <http://freetts.sourceforge.net/docs/index.php>
- [8] Paul Holser. (2011, May) JOpt Simple. [Online]. <http://pholser.github.com/jopt-simple/>
- [9] Melvyn J. Hunt, "Figures of merit for assessing connected-word recognisers," *Speech Communication*, vol. 9, no. 4, pp. 329-336 , August 1990.
- [10] Keith Vertanen, "Baseline WSJ Acoustic Models for HTK and Sphinx: Training Recipes and Recognition Experiments," *Technical Report, Cavendish Laboratory*, 2006.
- [11] Carnegie Mellon University. Sphinx-4. [Online].  
<http://cmusphinx.sourceforge.net/sphinx4/doc/ProgrammersGuide.html>