

SpeakShell Tutorial

- [Introduction](#)
 - [Required Software](#)
 - [SpeakShell Overview](#)
 - [Setting up SpeakShell](#)
 - [Setting up Sphinx2](#)
 - [Setting up SpeakShell Dependencies](#)
- [Demo Program: Speech Driven Robot](#)
 - [Set Up Player and Stage](#)
 - [Set Up Laser Obstacle Avoid Demo](#)
 - [Examination of Demo](#)
 - [Running the Demo](#)
- [Developing A language Model](#)
 - [Helpful Links](#)
 - [Setting up SphinxTrain](#)
 - [Using Sphinx Train](#)
 - [Designing your language](#)
 - [Using Sphinx Knowledge Base Tools](#)
 - [Using SphinxTrain](#)
 - [Decoder Testing and language Verification](#)

Introduction

This is a tutorial for the SpeakShell class. SpeakShell uses the [Sphinx Project](#) a complex HMM based Speech Recognition System from CMU to complete simple speech recognition task. It can be thought of as a sort of wrapper program that enables a program to run in parallel to a speech recognizer that sends commands back to the executing program.

An HMM-based system, like all other speech recognition systems, functions by first learning the characteristics (or parameters) of a set of sound units, and then using what it has learned about the units to find the most probable sequence of sound units for a given speech signal. The process of learning about the sound units is called **training**. The process of using the knowledge acquired to deduce the most probable sequence of units in a given signal is called **decoding**, or simply recognition¹.

Software Required Before You Start

You will need [Perl](#) to run some required Sphinx scripts and a working C, such as [gcc](#) compiler to compile source code.

[The Sphinx Project](#) provides files supporting compilation using Microsoft's Visual C++, i.e., the solution (.sln) and project (.vcproj) files needed to compile code in native Windows format.

SpeakShell Overview

SpeakShell is designed to take simple vocal commands and have change the behavior of a program. Essentially, it is a class that allows for speech input into an open ended program through a continuous or semi-continuous Sphinx language decoder. The basic premise is that this command can be reduced to something, boolean (either true or false) about the nature of the program. Telling your computer to open your web browser, for example, can be achieved by telling setting a series of boolean values, in this case Open=true and Browser=true, would communicate to a handling program what it is to do.

The power of SpeakShell is that it neither defines these flags, nor their effects apriori, both can be set by the user by modifying files. In short, SpeakShell lends the ability to set flags for decision making by vocal input from the user to any program.

At the highest level, an instance of SpeakShell is created, then a pointer to this class is passed to both the listening class and the program being modified via threading. Thus when the listening program detects a command, it modifies data that both threads point to and changes the way the executing program is running by setting booleans specified by the user. This is accomplished through a series of classes and structures as follows:

The flag class (flag.cc)

```
This class creates an instance of a flag for each name in the flag-name file
these flags represent changes made to the executing program in boolean form.
Example: the WAIT flag
when WAIT is false a program runs normally but when WAIT is true the program waits
using loops, these flags act as a method of program control. - This is not predefined,
just a logical way of thinking about WAIT as a boolean.
```

The flag class is invoked by the command class (command.cc)

```
Each command represents a verbal command the speech recognizer recognizes.
For example: Stop, Wait, Halt are all ways of telling a program to do the same thing -
stop what it is doing.
By using more than one phrase to map to the same action,
the program can feel more natural and an operator needs less training to use the program
effectively.
The person using the SpeakShell tool can likewise, define any commands or flags as they
wish and
use whatever feels natural or logical to them.
```

Each command points to the flag it acts upon and has a specific boolean value. For example: Start and Stop both operates on the WAIT flag Start sets it to false while Stop would set it to true. These relationships are defined in the **COMMANDMAP** file specified in speakshell.h.

The SpeakShell class (speakshell.cc)

```
The SpeakShell takes a list of flags and a map of commands upon these flags
as an argument and builds a structure to hold and access this data.
It likewise handles accessing the Sphinx decoder specified by the Sphinx-nick
bash script which it runs by means of the Linux popen command.
Its main component consists of a loop that runs this decoder
which outputs what the user says when a phrase matches the provided language model
which it deciphers by means of the provided HMM.
In addition to providing methods to access vectors for each flag and command set
(vector < Flag * > and vector < Command * > respectively).

SpeakShell also keeps track of the flag which the last command affected
(called Active Flag), whether or not the flag was changed by the command,
and whether or not it has new data (the has_changed variable is set to true when
something changes).
```

Speakshell's initializing function is kept separate (speakshell_run.cc) in order to allow multiple programs access to its functions (no confusion caused by multiple definitions of main). This is so the function can be easily invoked using execvp from the threader program (threader.cpp) in the demo provided.

```
This organization allows for a pointer to a speakshell to be created by threader,
then passed off to two threads.

The first thread handles the listening functions of
the speakshell and acts upon the flags as the user speaks by means of passing the
initialized
speakshell into the active_shell() functions in speakshell.cc.
```

```
The second thread, meanwhile, uses the speakshell pointer to check those same flags as a method of making decisions.
```

TODO: DO: add a picture of the described hierarchy

The demo provided uses Sphinx Shell to add voice Start, Stop, and Halt commands to the laser obstacle avoid program from the [Player/Stage](#) tool set.

```
START - the WAIT flag is set to false and the program runs normally
STOP - the WAIT flag is set to true and the program sleeps until told otherwise
HALT - the KILL flag is set to true causing the LOA program to exit its loop and quit,
the listening program also quits
The main program detects the end of both threads and terminates returning to the command line.
```

Setting up SpeakShell Recognizer

In order to set up the SpeakShell Recognizer you will need to install a Spinx Decoder. Sphinx2, PocketSphinx (for mobile devices), Sphinx3, and Sphinx4 (Java) all have the decoder you need. SpeakShell was designed for and tested with Sphinx2, it should work this this version and PocketSphinx (there is a great deal of overlap between the two). Since SpeakShell is a wrapper it should work just as well with the later versions of Sphinx but most of the tools developed as part of this project are meant for a simple language so the continuous mapping of Sphinx3 and 4 is unnecessary.

Before we begin setting up the SpeakShell Program please make sure you have the following files (Note: these are only the files needed to run SpeakShell, not threader or the complete demo):

```
command.cc      command.h      flag.cc
flag.h          Makefile    speakshell_run.cc
speakshell.cc   speakshell.h  sphinx2-nick
```

You will also need a bin directory in the same location which contains the following files:

```
command_map  flag_names (these filenames are specified in speakshell.h as FLAGFILE and
COMMANDMAP)
```

You can test that you have all these components by making speakshell:

```
make speakshell
```

Once this is done you'll need to install a Sphinx Decoder.

Installation of Sphinx2 decoder

First download the sphinx2 [tarball](#), if this does not work or you would like to install a different sphinx decoder please use the directions on the [Sphinx](#) webpage.

you should have a sphinx2.nightly.tar file, you might want to move it into a sepearte folder to keep a local copy. Extract the sphinx2 set up files:

```
gunzip -c sphinx2.nightly.tar.gz | tar xf -
```

to install sphinx2 use the following commands

```
./configure --prefix='absolute path to directory where you want to install sphinx2'
make
make install
```

The default installation directory for sphinx2 is /usr/local/share/sphinx2/ - it will install here if you don't specify a directory with prefix.

Set up SpeakShell Dependencies

The last thing you'll need to do in this step is locate your sphinx2_continuous file. This file is the script that handles the continuous decoding of spoken input from the user, it can be thought of as the very core of sphinx.

```
cd sphinx2
find -filename sphinx2_continuous
```

if you installed correctly this file should be at

```
'pwd'/bin/sphinx2_continuous
```

open the sphinx2-nick script and change the value of **S2CONTINUOUS** to reflect your new installation. for example on my home machine

```
S2CONTINUOUS="/home/overfiel/Sphinx/sphinx2/build/bin/sphinx2_continuous" #configure --
prefix='/home/overfiel/Sphinx/sphinx2/build'
```

Once this is done, the sphinx2 decoder can be executed by the sphinx-nick bash script. If you open up the sphinx2-nick script you will see that several parameters are sent to the decoder, these tell the program where the Hidden Markov Model and language Model files are located. **SPHINXMODELPATH** specifies the absolute real path location of these files, you will likewise need to change this line for simplicity all files have been moved to the "model" directory located with the SpeakShell files.

```
DOMAIN=turtle # this line is the name of language model to be used
HMMNAME="6k" # this line is the name of the HMM to be used
The demo provided uses a language "myam" developed for robot control
DOMAIN=myam
HMMNAME="myam"
SPHINXMODELPATH="/home/overfiel/research/test/model" # this needs to be set to the path of
your model directory
```

The remainder of the components control how the sphinx2 decoder acts on the language a short description has been given with each component and they've been organized into variables that make them easily changed. The defaults for sphinx2-nick have been optimized for use with the sphinx2 decoder on a simple language - simple being less than 60 words and direct word to word mapping rather than trying to recognize phonemes.

Demo: Player Stage Robot Control

This demo consists of a recreated version of the player stage file laserobstacleavoid.cc located in the /examples/libplayerc++/ directory of any current player installation and the files needed by player and stage to run the demo.

Player Stage Set Up

To install Player and Stage simple go to their [website](#) to download the tools you need.

If you are working for a computer on the UT network, player and stage have already been installed.

```
use the command
robot-player world/speech.cfg from the SpeakShell directory to run the simulation for this
demo
```

note that in order for this demo to work, the normal version of laser obstacle avoid needs to function please test this before continuing to set up your program. The demo will work in any enviroment in which the original laserobstacleavoid functioned.

Setting up the Demo

To set up this demo you will need a working *spekshell* exicutable and the following files

```
args.h          bin/          command.cc
command.h       flag.h       Makefile
model/myam/     new_laserobstacleavoid.cc
spekshell_run.cc      spekshell.cc
spekshell.h     sphinx2-nick*  threader.cpp
threader.h      world/
```

the bin directory contains the flag and command information just as in spekshell's set up
the model/myam directory contains the language model and the HMM files for the speech
recognition task
the world directory contains all the files stage needs to build the simulated robot world
args.h is likewise used to specify robot parameters.

Be sure that the sphinx2-nick file is properly set up to use the myam model

```
#!/bin/sh
#filepath to continuous sphinx2-script
S2CONTINUOUS="'directory where you installed sphinx2'/bin/sphinx2_continuous"

#Domain is the filename of the domain with a class-based language model
DOMAIN=myam

#Name of the HMM File
HMMNAME="myam"

#Path to language model - domain files are here
SPHINXMODELPATH="/home/overfiel/research/test/model"

# HMM model
HMM="$SPHINXMODELPATH/hmm/$HMMNAME"

#language model location
TASK="$SPHINXMODELPATH/lm/$DOMAIN"

#language model
LM="$DOMAIN.lm"

#Dictionary Location
DICT="$SPHINXMODELPATH/lm/$DOMAIN/$DOMAIN.dic"
```

Additionally check to make sure each of these files exist: The *model/hmm/myam* directory should have the .chmm files for every word in the language Normally HHM blocks are made for every phoneme but since this language is so small one was created for every word using SphinxTrain.

```
ABORT.chmm      BATTERY.chmm      BEGIN.chmm      COMMANDS.chmm
DIAGNOSTIC.chmm DOCK.chmm          END.chmm         FOLLOW.chmm
GO.chmm         HALT.chmm          HELP.chmm        HOME.chmm
KEY.chmm        LIFE.chmm          LIST.chmm        ME.chmm
NO.chmm         OPERATION.chmm     PAUSE.chmm       REPORT.chmm
RESUME.chmm     RETURN.chmm        SIL.chmm         START.chmm
STATUS.chmm     STOP.chmm          YES.chmm
```

it should also contain these phonetic files

phone map	cep.256.var noisedict	cep.256.vec p3cep.256.var	d2cep.256.var p3cep.256.vec	d2cep.256.vec sendump
--------------	--------------------------	------------------------------	--------------------------------	--------------------------

the language model is in the directory **model/lm/myam** and should consists of the following files:

feat.params	myam.handdict	myam.sent.arpabo	myam_train.transcription
myam.corpus	myam.hdict	myam_test.fileids	myam.dic
myam.lm	myam_test.transcription	myam.vocab	
myam.extra	myam.phone	myam.token	sphinx_decode.cfg
myam.filler	myam.sent	myam_train.fileids	sphinx_train.cfg

To use this demo simply make the threader and the laser programs

```
make thread
make laser
or make all (will make thread, laser, and speakshell)
```

A Closer Look at how the Demo Works

Take a moment to examine the first few lines of **new_laserobstacleavoid.cc**.

Make special note of the three global variables.

```
bool run;
bool using_speech;
bool active;// global variables for laser control

void sphinx_update(Speak_Shell *SS)
{
    //new stuff - check to see if its been told to stop
    if(using_speech)
    {
        // if program is using speech
        //cout << "laser: is using speech" <check_for_change()
        // while the active flag has not changed
        sleep(2);
    }
    SS->set_change(false);

    //cout << "laser: knows ss has changed" <check_has_active()
    // still has an active flag
    if(SS->get_active_flag_name()=="WAIT")// active tells it to run normal LOA
    {
        //cout << "laser knows active flag is wait" <get_active_flag_val()
        {
            set_active(false);
            cout << "Stop robot" << endl;
            //cout << "active should be false, active:" << active << endl;
        }
        else
        {
            set_active(true);
            cout << "start robot" << endl;
        }
    }
    else if(SS->get_active_flag_name()=="KILL")// kill command stops server
    {
        run=(!SS->get_active_flag_val());// if kill is true, run is false
        if(!run)// if run is false active need be as well to break the loop
        {
            set_active(false);
            //cout <<" run and active should be set to false" << endl;
            //cout <<"run: " << run << " active: " << active << endl;
        }
    }
}
```

```

    }
    } //end of if working normally
else
{
    std::cout << "SS is null " << endl;
    exit(1);
} // somethings is really broke
} // end of if using speech, otherwise does nothing
}

```

This function is relatively primitive it does two things, first it makes sure that LOA is only affected by the WAIT and KILL flags, this lets the program use a rather robust language (such as myam) without needing to account for all the flags. The use of a few global booleans and their accessors functions was also added to facilitate the use of a single update void function (sphinx_update) and change the original laser code as little as possible. This allows the code to **run in its original manor**. It is very important that the speakshell not become necessary from the standpoint that it won't always work reliably. It is a good idea to maintain any codes ability to work without speech when modifying existing work to use speakshell.

```

My demo program ./laser works just like laserobstacle avoid does when compiled using
the original code from the Player example.
The separate new_laserobstacleavoid_run.cc code enables invokes the run_laser
in such a way that it is identical to the original code.
Only when invoked by an instance of threader and passes a pointer to a speakshell does
any of
the speakshell behavior of new_laserobstacleavoid function at all.

```

This is simply good manners from a teamwork point of view, this way new code does not force others to use speakshell to use a program, or mangle someone else's code on accident. This practice is also advisable from a task planning point of view because a user will not always be able to use speech reliably. If you take nothing away else away from these instructions and demo it should be the following.

Speech does not always work

Too many factors: bad training data, outside noise, or even dead batteries in your microphone can all cause speech recognition to fail. Plan for this to happen in your code.

The second reason for looking at this code from the laser demo was to point out that only two flags are active. If you look at the command_map file (bin/command_map):

```

## all of the form command key : flag : status
## everything must be CAPITALIZED and each section must be colon (:) delimited
## example - ABORT : KILL : ON - the abort command sets the kill flag to on
## Commands will be verified from flags - MAKE SURE ALL FLAGS ARE CORRECT BEFORE
PROCEEDING
## Commands will be mapped to flags created from bin/flag_names
## lines beginning with # are ignored
ABORT : KILL : ON
BATTERY LIFE : BATTERY : ON
BEGIN OPEARTION : EXECUTE : ON
DIAGNOSTIC : DIAGNOSTIC FLAG : ON
DIAGNOSTIC REPORT : DIAGNOSTIC : ON
DOCK : RETURN : ON
END OPERATION : EXECUTE : ON
FOLLOW : FOLLOW : ON
FOLLOW ME : FOLLOW : ON
GO : WAIT : OFF
HALT : KILL : ON
HELP : HELP : ON
HELP ME : HELP : ON
KEY : KEY : ON
LIST COMMANDS : LIST : ON
NO : NO : ON
PAUSE : WAIT : ON
RESUME : WAIT : OFF
RETURN HOME : RETURN : ON

```

```
START : WAIT : OFF
STATUS : STATUS : ON
STATUS REPORT : STATUS : ON
STOP : WAIT : ON
YES : YES : ON
```

You can see that HALT and ABORT both turn the kill flag on

while GO PAUSE RESUME START and STOP all effect the wait flag

This means these are the only commands that the demo will effect your program because of how the laser program is written. Despite the recognizer being able to recognize 28 words, only seven get used. Notice how the language are all commands you might want to tell a robot doing a generic task. It was designed with that in mind. This will be covered more in depth in the next section Designing and Building a language Model.

Running the Demo

The demo program is designed to run on a Pioneer robot, if you lack access to one of these robots you will need to settle for running the program in simulation. To run the robot laser simulation, first you need to invoke an instance of the virtual world using the speech.cfg included with this package.

```
On a UT computer this is done using the following command
from the speakshell directory:

robot-player world/speech.cfg
```

If everything is working correctly you should see a single red Pioneer robot in a simulated environment that looks like this

TODO: ADD SCREEN CAPTURE

once this is done simply run the threader program with no arguments and its default parameters

In a separate command prompt from where you set up the environment type:

```
cd speakshell
make
./thread
```

This will launch an instance of the demo that listens for commands with the following effects

```
HALT and ABORT
end the program, stopping both threads and returning to the command line

GO RESUME and START
set the wait flag to false
the robot moves forward semi-randomly using its laser to avoid obstacles

STOP and PAUSE
set the wait flat to true
the robot stops moving and waits to be told what to do
```

Developing a Language Model

Developing a language model is a long and arduous task. It takes several hours worth of wave fails to create a language model that is even usable and several more still for said model to begin to be accurate even with a simplified language such as myam. The primary goal of speakshell was to help alleviate this - that is why you

can change the mapping of commands to suit your task and why the language includes so many phrases that fit a particular field. It is highly recommended you try to adapt one of the created language models, be it [AN4](#) or [RM1](#) from the [Sphinx Project](#) or my own simple myam language to your task.

You should also realize what you are trying to do with your language driven task. If you are looking for a tele-typing or similar task other programs may exist that suit your needs better. If you find your task requires a specialized language model, these notes should be of some assistance.

Helpful Links for Developing a Language Model

[Robust Group Tutorial](#)

This is the basic walk through of how to train a phonetic language
The basics of setting up and running SphinxTrain and each decoder are here
The tutorial also walks through a few more advanced topics such as phone-merging and state tying
as well as giving examples of how to run the initial encode and decodes of a language.

[Sphinx Train Documentation](#)

This is the big book of more than you would ever want
to know about Acoustic Language Model creation.
Sphinx Train is the program you will use to construct your
LM and HMM files and create your language model.
At a minimum you need to read and understand the first section "Before You Train".

Also Note:

When you have a very small closed vocabulary (50-60 words) - like the myam example

If you have only about 50-60 words in your vocabulary,
and if your entire test data vocabulary is covered by the training data,
then you are probably better off training word models rather than phone models.
To do this, simply define the phone set as your set of words themselves and
have a dictionary that maps each word to itself and train.
Also, use a lesser number of fillers, and if you do need to train phone models
make sure that each of your tied states
has enough counts (at least 5 or 10 instances of each)^1.

This incite was particularly helpful to me since my speech recognition task
consisted of recognizing only 28 very specific phrases.
I was able to construct a much smaller and thus faster HMM
while the direct word-only mapping allows my model to be much more accurate.
The AN4 language model had difficulty distinguishing between
forward and four because both words share phonemes at the start of the words.
This is completely avoided by mapping to whole words and having
each word in the language be as phonetically unique as possible.

[Sphinx Knowledge Base Tool](#)

This is a huge time saver.
The program located here takes a corpus file (a list of all the sentences of the training
data)
and generates most of the language model files.

It should be noted that the transcription files it creates have an error in them.

The transcription takes the phrases in the corpus file,
capitalizes all the letters and adds the silence characters.

```
< s > line from corpus file </s> (no spaces around s character)
the transcriptions need to read
< s > line from corpus file </s> (filename)
otherwise using Sphinx Train will result in an error.
```

For example it should read like this:
< s > ABORT </s> (ABORT1)

Additionally the dictionary created will be phonetic
so if you plan to map based on whole words you will need to rewrite this file.
TODO: Make a script that does this

[Bakuzen Blog](#)

While the example of Acoustic Model creation and adaptation by Bakuzen is for Sphinx3 and then adapted for Sphinx4 all the basic commands incites, and information holds for all versions of sphinx. It is also a great deal more accessible than the Robust Group Tutorial.

The blog also provides a script that extracts phones from a dictionary file, effectively creating a piece of the acoustic model that was missing from the Knowledge Base Tool.

This, however, like the knowledge base tool is designed for use with phonetic matching so if you plan to use a model that matches words you need to make your own. This is as simple as writing each word in the dictionary once in a file each on a separate line in all caps.

TODO: Write a script that does this too

Installation of Sphinx Train

First download the SphinxTrain [tarball](#), if this does not work or you would like to install a different sphinx decoder please use the directions on the [Sphinx](#) webpage.

You should have a SphinxTrain.nightly.tar.gz file, you might want to move it into a separate folder to keep a local copy.

Extract the SphinxTrain set up files:

```
gunzip -c SphinxTrain.nightly.tar.gz | tar xf -
```

to install SphinxTrain use the following commands

```
cd SphinxTrain
./configure
make
```

Using Sphinx Train

Additionally its a good idea to go ahead and try training an existing language just to make sure everything is working alright.

You have the option of using [AN4](#) or [RM1](#) or the myam language files associated with the Robot Demo.

To train an an4 decoder do the following:

Move the an4 directory to the same place as your SphinxTrain Directory

```
cd an4
vi etc/sphinx_train.cfg (or any editor you want to use)
```

change the HMM type to semi continuous

```
#$CFG_HMM_TYPE = '.cont.'; # Sphinx III (line 79)
$CFG_HMM_TYPE = '.semi.'; # Sphinx I(I line 80)
```

save the file and go back to your an4 directory

run the perl scripts to train the LM

```
perl scripts_pl/make_feats.pl -ctl etc/an4_train.fileids (perl script will check the file-ids)
```

then

```
perl scripts_pl/RunAll.pl
```

This should take a while it will run the following scripts

```
perl scripts_pl/00.verify/verify_all.pl
perl scripts_pl/10.vector_quantize/slave.VQ.pl
perl scripts_pl/20.ci_hmm/slave_convq.pl
perl scripts_pl/30.cd_hmm_untied/slave_convq.pl
perl scripts_pl/40.builtrees/slave.treebuilder.pl
perl scripts_pl/45.prunetree/slave-state-tying.pl
perl scripts_pl/50.cd_hmm_tied/slave_convq.pl
perl scripts_pl/90.deleted_interpolation/deleted_interpolation.pl
perl scripts_pl/99.make_s2_models/make_s2_models.pl
```

You will also want to do a preliminary decoder run just to make sure everything is valid there as well. To do this make sure you have a decoder installed as above then do the following

```
cd an4
perl scripts_pl/make_feats.pl -ctl etc/an4_test.fileids
NOTICE: test.fileids not train - different files

perl scripts_pl/decode/slave.pl
```

This will test the accuracy of your installed decoder on the an4 language.

you should get output like this:

```
SENTENCE ERROR: 56.154% (73/130)  WORD ERROR RATE: 16.429% (127/773)
```

Designing Your Language

Step one, name your language, the associated file names of the LM will be language_name.filetype.

For example, following Bakuzen's blog I named my demo language myam.

First, create a folder for your language - which I will refer to as myam from this point forward for simplicity in the same directory as your SphinxTrain installation then type the following:

```
../SphinxTrain/scripts_pl/setup_SphinxTrain.pl --task myam

Notice that myam is the name of the task and is also the name of your folder.
It doesn't have to be, but it makes things easier later.
```

This will make a lot of files and directories that Sphinx will use to build your LM and HMM.

Of immediate note is the **ect/** directory

To create a language you will need a few things:

1. A set of wave files to train your language on

```
This can be whatever you want as long as every word spoken
in the wave files is contained in your language.
```

```
You can record whole sentences
"Open Firefox and go to my email."
or just words "email" or "open" would work depending on what you want your language to
do.

It's a good idea to try to be uniform in how much you use words.
If every example you train with uses the word "Fluffy"
your decoder will mistake a lot of words for "Fluffy"
or worse, not work when you don't say "Fluffy" and perform very poorly.

There is also a danger with words that sound similar
The sphinx example AN4 has a difficult time distinguishing FOUR and FORWARD
because both have the same sound at the beginning.

When I created the myam language since I was setting out to make 28 simple commands
I simply recorded each command as its own file. Each file is just the words in the file
name.
My wave files look like this:
ABORT1.wav
ABORT2.wav
ABORT3.wav
...

put these files into etc/wav/.

I also organized each word in my language into folders,
organization is never a bad thing.
ect/wav/ABORT/ABORT1.wav
```

There are a few ways to go about making these wav files, you can use generative programs such as [Festival](#) to make all your wav files or get a recording program such as [Audacity](#).

```
If you use Audacity or a similar program to record the wav files you will be using for
training a language model
you will need to change the file types associated with the trainer.

To do this modify the sphinx_train.cfg located in your projects etc/ directory
in the case of Audacity the program created pure wav files in their raw format so these
lines needed to be changed
in sphinx_train.cfg for the trainer to work.

$CFG_WAVFILE_EXTENSION = 'sph';
changed to
$CFG_WAVFILE_EXTENSION = 'wav';

$CFG_WAVFILE_TYPE = 'nist'; # one of nist, mswav, raw
changed to
$CFG_WAVFILE_TYPE = 'raw'; # one of nist, mswav, raw
```

It is highly recommend you use Festival as you will need a great deal of wav files. In my first attempt at creating myam I had 25 unique wave files for each command in the language and had no where near enough data to have an accurately trained language despite its simplicity and the fact that I was training the decoder to solely my voice. I needed about 50 wav files per command in the language to have enough deviation between words for the decoder to function properly, even then it is not as accurate as I would like - in short training takes an immense amount of data.

2. The corpus - a transcription of the contents of these files each line should contain exactly what is spoken in each wave file, all lower case this will be called your corpus file

For example myam.corpus :

```
abort
abort
abort
abort

bättery life
battery life
battery life
battery life
```

```
...
```

3. myam_train.fileids

A list of the file names (without the .wav extension) for the wav files you want to use to train since I used directories myam_train.fileids looks like this:

```
ABORT/ABORT1
ABORT/ABORT2
ABORT/ABORT3
ABORT/ABORT4
ABORT/ABORT5
ABORT/ABORT6
ABORT/ABORT7
ABORT/ABORT8
....
BATTERY_LIFE/BATTERLY_LIFE1
....
```

The corpus and train_fileids should have the exact same number of lines.

Each line in the fileid should match the corpus line exactly, if you mismatch the files to the corpus the language will train incorrectly.

No files should have any whitespace beyond the separation of parts of a phrase in the corpus. Extra spaces can cause errors, its very important that you check this.

In vi use

```
:set list
```

to see newline, space, and invisible characters.

4. Filler Dictionary - silence, coughing, and anything else not in the language

For a simple filler dictionary just copy this into myam.filler:

```
without spaces around brackets
< s > SIL
< sil > SIL
< /s > SIL
```

Now you've got everything you need to train a language. You still need a few more files to use SphinxTrain but those are provided by the [Sphinx Knowledge Base Tool](#).

Using the Knowledge Base Tool

The program at [this website](#) takes a corpus file as input and generates most of the language model files.

Upload your myam.corpus scroll down to the bottom of the page and press the button labeled

Compile Knowledge Base

You will need all the generated files so click on the link:

For your convenience there is also a gzip'd tar file version of the above set.

Download the tar file, an extract it, you should have the following files, all of which have a timestamp in place of the filename.

You will need to rename these files so that they have your task name instead of the timestamp:

```
myam.corpus  myam.extra  myam.hdict  myam.sent  myam.token
myam.dic     myam.handdict myam.lm     myam.sent.arpabo myam.vocab
```

Copy these files into your etc/ directory.

Examine myam.dic:

```
ABORT  AH B AO R T
BATTERY B AE T ER IY
BEGIN  B IH G IH N
...
```

Notice how it phonetically maps the files. If you wish to add possible pronunciations, this is where you will do so. Additionally, if you wish to create a word-mapped language (such as the myam example) you will need to edit this file so that each word in the dictionary maps to itself rather than a pronunciation.

myam.dic (phonetic) looks like this:

```
ABORT  ABORT
BATTERY BATTERY
BEGIN  BEGIN
...
```

Next look at myam.sent

```
< s > ABORT < /s >
< s > ABORT < /s >
< s > ABORT < /s >
....
```

this is equivalent to what should be in your myam_train.transcription file save a copy with that name now.

As stated above this file also has an error in it, the file name needs to be included on each line with the phrase uttered.

The transcriptions need to read
< s > line from corpus file </s> (filename) - without the space around the s otherwise using SphinxTrain will result in an error.

For example it should read like this:
< s > ABORT </s> (ABORT1)

The final file you need to provide SphinxTrain is your phone file (myam.phone).

The phone file contains one and only one instance of every phone to occur within a language. [Bakuzen Blog](#) provides a tool that will pull each phone from a dictionary and place it in alphabetical order (though it does repeat some phones and will need to be manually edited).

If you are creating a word-mapped language you simply need to include every word as a phone.

Regardless of the method you use **you must include the silence phone (SIL).**

myam.phone looks like this:

```
SIL
ABORT
```

```

BATTERY
LIFE
BEGIN
OPERATION
DIAGNOSTIC
REPORT
DOCK
END
FOLLOW
ME
GO
HALT
HELP
KEY
LIST
COMMANDS
NO
PAUSE
RETURN
HOME
RESUME
START
STATUS
STOP
YES

```

Move the .phone file to your etc/ directory.

Using SphinxTrain

Make sure you are in your task directory.

```
cd myam
```

Check to make sure you have all the files you need.

```

myam.filler      myam.phone      myam.sent      myam.token
myam.corpus      myam.handdict   myam.vocab     myam.dic
myam.hdict       myam.sent.arpabo myam_train.fileids
sphinx_train.cfg myam.lm         myam_train.transcription

```

Now check to make sure your sphinx_train.cfg file is set up correctly.

```

$CFG_DB_NAME = "myam" or whatever you set your task name to be.
$CFG_BASE_DIR = "/path/to/myam" or directory where your task folder exists
$CFG_SPHINXTRAIN_DIR = relative path where your SphinxTrain folder is.

for myam these files were set as follows:
$CFG_DB_NAME = "myam";
$CFG_BASE_DIR = "/home/overfiel/Sphinx/myam";
$CFG_SPHINXTRAIN_DIR = "../SphinxTrain";

also double check that your wav file information is in the correct form (since I used
Audacity):
# Audio waveform and feature file information
$CFG_WAVFILES_DIR = "$CFG_BASE_DIR/wav";
$CFG_WAVFILE_EXTENSION = 'sph';
$CFG_WAVFILE_EXTENSION = 'wav';
$CFG_WAVFILE_TYPE = 'nist'; # one of nist, mswav, raw
$CFG_WAVFILE_TYPE = 'raw'; # one of nist, mswav, raw
$CFG_FEATFILES_DIR = "$CFG_BASE_DIR/feat";
$CFG_FEATFILE_EXTENSION = 'mfc';
$CFG_VECTOR_LENGTH = 13;

```

Once this is done you are ready to train.

Move the task directory to the same place as your SphinxTrain Directory

Enter your task directory and run the setup training perl script on your train.fileids to train your language model.

```
perl scripts_pl/make_feats.pl -ctl etc/myam_train.fileids
```

then run the training scripts either one at a time or using the RunAll.pl script.

```
perl scripts_pl/RunAll.pl
```

This should take a while. It will run the following scripts

```
perl scripts_pl/00.verify/verify_all.pl
perl scripts_pl/10.vector_quantize/slave.VQ.pl
perl scripts_pl/20.ci_hmm/slave_convg.pl
perl scripts_pl/30.cd_hmm_untied/slave_convg.pl
perl scripts_pl/40.builtrees/slave.treebuilder.pl
perl scripts_pl/45.prunetree/slave-state-tying.pl
perl scripts_pl/50.cd_hmm_tied/slave_convg.pl
perl scripts_pl/90.deleted_interpolation/deleted_interpolation.pl
perl scripts_pl/99.make_s2_models/make_s2_models.pl
```

Note the following:

```
Stages 90.deleted-interpolation and 99.make_s2_models are meaningful
only if you are training models for SPHINX-2.
Deleted interpolation smooths the HMMs, which are then converted to the format used by
SPHINX-2.
At the end of any stage you may use the models for recognition.
Remember that you may decode even while the training is in progress,
provided you are certain that you have crossed the stage which generates the models you
want to decode with.^2
```

This will create all the .chmm files the decoder needs to operate and place them in your task directory.

Decoder Testing and Language Verification

In order to run the Sphinx2 (or other version) decoder you will need to modify a few of the other files.

First, you will need to transfer the files you need from sphinx2 to decode into your task folder.

```
cd myam
perl ../sphinx2/scripts/setup_sphinx2.pl -force -langmod myam.lm -task myam
```

This will put the files you need including **sphinx_decode.cfg** into your etc directory and add the other files where you need them in your task directory.

You will need to make a few modifications to your sphinx_decode.cfg file.

```
$DEC_CFG_DICTIONARY = the dictionary used by the decoder
$DEC_CFG_FILLERDICT = the filler dictionary
$DEC_CFG_GAUSSIANS = the number of densities in the model used by the decoder.
You'll need to experiment to find the right number for you, the number of Gaussians
affects accuracy.
It could be 1, 2, 4, 8, or higher depending on the complexity of your language.
$DEC_CFG_MODEL_NAME = the model name, in the an4 case this was "turtle" for myam I simple
left it "myam"
It is the prefix of the language model files (the ones created by the knowledge base
tool).
```

```
$DEC_CFG_LANGUAGEWEIGHT the language weight. A value between 6 and 13 is recommended.
See the Robust Group Tutorial for more information.

$DEC_CFG_ALIGN = The word alignment program you're going to use, its built in, you can
leave it as is.
```

the myam sphinx_decode.cfg file looks like this:

```
# These are filled in at configuration time
$DEC_CFG_DB_NAME = 'myam';
$DEC_CFG_BASE_DIR = '/home/overfiel/Sphinx/myam';
$DEC_CFG_SPHINXDECODER_DIR = '/home/overfiel/Sphinx/sphinx2';
$DEC_CFG_SPHINXTRAIN_CFG = "$DEC_CFG_BASE_DIR/etc/sphinx_train.cfg";

$DEC_CFG_LANGUAGEMODEL_DIR = "$DEC_CFG_BASE_DIR/etc";
$DEC_CFG_LANGUAGEMODEL = "$DEC_CFG_LANGUAGEMODEL_DIR/myam.ug.lm";
$DEC_CFG_LANGUAGEWEIGHT = "10";
$DEC_CFG_BEAMWIDTH = "1e-80";
$DEC_CFG_WORDBEAM = "1e-48";
$DEC_CFG_ALIGN = "builtin";
#*****variables used in characterizing models*****
$DEC_CFG_HMM_TYPE = $CFG_HMM_TYPE;
...
```

Lastly you will need a list of file_ids to test the decoder with. These can use the same wav files you used for SphinTrain, but its a better test if you use some new files.

If you make a second set of files to test your decoder, be sure to make a file_id file just as with SphinxTrain.

The decoder test scripts can be ran with the following command (where test.fileids is the list of wav files to test against.

```
perl scripts_pl/make_feats.pl -ctl etc/myam_test.fileids # this will create the needed
feature files

perl scripts_pl/decode/slave.pl # runs the decoder
```

Once this is done the decoder will produce an accuracy report from which you might want to adjust your training or decoding weights.

```
SENTENCE ERROR: 38.833% (233/600)   WORD ERROR RATE: 7.640% (434/5681)
```

References

1. "[Sphinx Train Documentation](#)". Sphinx Group CMU Berkley. 22 Nov. 2000
< <http://www.speech.cs.cmu.edu/sphinxman/scriptman1.html> > .
2. Gouvea, Evandro. "[Robust Group Tutorial](#)." Sphinx Group CMU Berkley. 12 Sept 11:06:36 EDT 2008
< <http://www.speech.cs.cmu.edu/sphinx/tutorial.html> > .

This page was created by [Nick Overfield](#), adapted from a page created by Evandro Gouvea. For comments, suggestions, or questions, contact the author.

Last modified: 28 April 2009