RESPITE CASA Toolkit
CTK v1.1.1
User's Guide

Jon Barker

March 8, 2001

Contents

Chapter 1

Introduction

The RESPITE project aims to extend and apply two novel technologies – missing data theory and multi-stream theory – to the problem of robust automatic speech recognition (ASR), with particular application to cellular phones and in-car environments. Both these approaches rely on the successful identification of regions of the signal spectrum containing reliable information. Within the RESPITE project several contrasting approaches to this problem will be compared. One of these approaches is Computational Auditory Scene Analysis.

The RESPITE CASA Toolkit aims to provide a flexible, extensible and consistent framework for the development and testing of CASA systems within the project. This framework should facilitate the smooth integration of CASA software components contributed by the various RESPITE partners. It is also hoped that this toolkit may prove a useful resource to the CASA community as a whole, both as a reference framework for comparing different developments, and as a way to lower the barriers to entry into this research field.

To allow the desired flexibility and extensibility the CASA Toolkit has been designed around a block processing paradigm. Much of this design has been influenced by approaches taken in other signal processing systems, notably, the Speech Training and Recognition Unified Tool (STRUT) of the TCTS lab at Faculté Polytechnique de Mons, and the Ptolemy project, Department of EECS, UC Berkeley.

The structure of the rest of the manual

Chapter 2 explains how to **download and install** the CTK package on your system.

Chapter 3 describes the block orientated processing and online data flow model underlying the toolkit.

Chapter 4 provides a number of walk through **tutorials** which illustrate how

to write CASA Toolkit scripts to run simple signal processing systems. These tutorials have been designed to be progressive and to illustrate as many of the principles of the CTK package as possible. Once you have worked through the tutorials you should be in a position to start building your own CTK systems.

Chapter 5 describes the CASA Toolkit graphical user interface (GUI). This interface has been built on top of the toolkit to aid the design of more complicated systems. The GUI allows systems to be designed in an intuitive manner by arranging widgets on a canvas. The graphical representation can then be saved as a CTK Script that can be run in the normal way. The GUI can also be used to visualise and edit existing scripts.

The toolkit has been designed to be extendable. Users with some C/C++ experience can write their own primitive blocks and compile them into the inbuilt library. Chapter 6 explains the necessary steps.

Chapter 2

CTK Installation

2.1 Unix users

1. **Download the gzip compressed CTK tar file**. This is available from:

   ```
   http://www.dcs.shef.ac.uk/research/groups/spandh/projects/respite/ctk/
   ```

2. **Unpack the archive** if you have not done so already:

   ```
   cd /usr/local
   gunzip CTKv1.1.1.tar.gz   # uncompress the archive
   tar xf CTKv1.1.1.tar      # unpack it
   ```

   This creates the directory `/usr/local/CTK` containing the files from the main archive.

   If you have downloaded the data tar file then unpack this now aswell. First copy it to /usr/local (or whereever you wish to place CTK) and then:

   ```
   cd /usr/local
   gunzip CTK.data.gz   # uncompress the archive
   tar xf CTK.data.tar      # unpack it
   ```

   The rest of these instructions assume that CTK is installed in `/usr/local/CTK`.

3. **Set some CTK environment variables** in the file `.profile` (or `.login`, depending on your shell) in your home directory. (Create the file if it is not there already.)

   - **CTKROOT** - wherever you installed CTK
   - **CTKLOCAL** - for user local CTK paths
   - **PATH** - to locate the CTK binaries
   - **MANPATH** - to access the CTK man pages

This is done like this:

In `.profile` (if your shell is bash, ksh, zsh or sh), add the following lines:

```
CTKROOT=/usr/local/CTK
CTKLOCAL=/usr/local/CTK/local
PATH=$CTKLOCAL/bin:$PATH
if [ $MANPATH ]
then
  MANPATH=$CTKROOT/man:$MANPATH
else
  MANPATH=$CTKROOT/man
fi

export CTKROOT CTKLOCAL PATH MANPATH
```

In `.login` (in case your shell is csh or tcsh), add the following lines:

```
if ( ! $?CTKROOT ) then
  setenv CTKROOT /usr/local/CTK
endif
if ( ! $?CTKLOCAL ) then
  setenv CTKLOCAL $CTKROOT/local
endif
if ( $?PATH ) then
  setenv PATH $CTKLOCAL/bin:$PATH
else
  setenv PATH $CTKLOCAL/bin
endif
if ( $?MANPATH ) then
  setenv MANPATH $CTKROOT/man:$MANPATH
else
  setenv MANPATH $CTKROOT/man
endif
```

After you have done this, you will need to login again, or re-source the profile before continuing, so that at least `$CTKROOT` and `$CTKLOCAL` are set. The installation will give an error message and not proceed otherwise.

4. **Make sure Qt is properly installed**

   The Qt widget library is needed if you want to run the CTK GUI. It should be installed on your system and the following environment variables should be set:

   - QTDIR - wherever Qt is installed on your system
   - PATH - to locate the moc program
   - LD_LIBRARY_PATH - for the shared Qt library

   If you're using GNU g++, you may also want to set these:

- LIBRARY_PATH - contains library file path
- CPLUS_INCLUDE_PATH - contains C++ include file path

This is done like this:

In .profile (if your shell is bash, ksh, zsh or sh), add the following lines:

```
QTDIR=/usr/local/qt
PATH=$QTDIR/bin:$PATH
if [ $LD_LIBRARY_PATH ]
then
        LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
else
        LD_LIBRARY_PATH=$QTDIR/lib
fi
LIBRARY_PATH=$LD_LIBRARY_PATH
if [ $CPLUS_INCLUDE_PATH ]
then
        CPLUS_INCLUDE_PATH=$QTDIR/include:$CPLUS_INCLUDE_PATH
else
        CPLUS_INCLUDE_PATH=$QTDIR/include
fi

export QTDIR PATH MANPATH LD_LIBRARY_PATH LIBRARY_PATH
export CPLUS_INCLUDE_PATH
```

In .login (in case your shell is csh or tcsh), add the following lines:

```
if ( ! $?QTDIR ) then
    setenv QTDIR /usr/local/qt
endif
if ( $?PATH ) then
    setenv PATH $QTDIR/bin:$PATH
else
    setenv PATH $QTDIR/bin
endif
if ( $?MANPATH ) then
    setenv MANPATH $QTDIR/man:$MANPATH
else
    setenv MANPATH $QTDIR/man
endif
if ( $?LD_LIBRARY_PATH ) then
    setenv LD_LIBRARY_PATH $QTDIR/lib:$LD_LIBRARY_PATH
else
    setenv LD_LIBRARY_PATH $QTDIR/lib
endif
if ( ! $?LIBRARY_PATH ) then
    setenv LIBRARY_PATH $LD_LIBRARY_PATH
endif
```

```
if ( $?CPLUS_INCLUDE_PATH ) then
    setenv CPLUS_INCLUDE_PATH $QTDIR/include:$CPLUS_INCLUDE_PATH
else
    setenv CPLUS_INCLUDE_PATH $QTDIR/include
endif
```

After you have done this, you will need to login again, or re-source the profile before continuing, so that at least $QTDIR is set. The installation will give an error message and not proceed otherwise.

5. **Compile the CTK binaries**

   You may find that pre-compiled binaries are available for download directly from the CTK web site. In this case simply copy the tar file to `$CTKROOT/..` and unpack it. If you can't find binaries for your system, then you will have to compile your own.

   The `$CTKROOT/src` directory contains a makefile which compiles the CTK binaries. The basic system - with no MATLAB or Qt support - can be built by typing:

   ```
   cd $CTKROOT/src
   make
   ```

   If this fails or if you want to install the MATLAB interface or the Qt-based GUI you will have to edit `$CTKROOT/src/Makefile`. First read the INSTALL file found in `$CTKROOT/src`.

6. **Setting up with MATLAB support**

   If you have compiled with MATLAB support you will also need to set your LD_LIBRARY_PATH to locate the MATLAB libraries.

   This is done like this:

   In .profile (if your shell is bash, ksh, zsh or sh), add the following lines:

   ```
   if [ $LD_LIBRARY_PATH ]
   then
           LD_LIBRARY_PATH=/usr/local/matlab/extern/lib/sol2:$LD_LIBRARY_PATH
   else
           LD_LIBRARY_PATH=/usr/local/matlab/extern/lib/sol2
   ```

   In .login (in case your shell is csh or tcsh), add the following lines:

   ```
   if ( $?LD_LIBRARY_PATH ) then
       setenv LD_LIBRARY_PATH /usr/local/matlab/extern/lib/sol2:$LD_LIBRARY_PATH
   else
       setenv LD_LIBRARY_PATH /usr/local/matlab/extern/lib/sol2
   endif
   ```

   The exact path may vary on your system, but it should point to the location of the files `libmat.so` and `libeng.so`.

7. **Preparing the CTK script files**

   The example CTK script files need to be editted so that the first line of each script references the CTKScript script interpreter binary. This can be done automatically by using the 'fix_scripts' script. Simply type:

   ```
   fix_scripts
   ```

8. **Setting up your editor (emacs only)**

   When viewing CTK script files emacs should be put into sh-mode. To make this happen automatically add the following lines to your `.emacs` file:

   ```
   (setq auto-mode-alist
     (append '(("\\.ctk$"  . sh-mode))
        auto-mode-alist))
   ```

Chapter 3

The CTK Design

Bottom-up CASA systems are typically built in a modular fashion from fairly standard components. The CASA Toolkit aims to ease the design of CASA systems by offering the user a library of well documented pre-defined processing blocks and a mechanism for combining them in an arbitrary fashion.

Primitive Blocks

At the heart of the CASA toolkit is a library of 'primitive blocks'. These blocks are defined in terms of the foll owing:

- **Input sockets**
- **Output sockets**
- **An algorithm** – describing how data at the inputs is to be transformed and placed on the outputs.
- **Parameters** - A set of parameters which may be set from outside the block and may control the block's operation.

Figure 3.1 illustrates several primitive blocks.

Block's may be broadly divided into 3 distinct types: **source blocks**, **processing blocks** and **sink blocks**.

A **source block** has output sockets but no input sockets. A source block will generate its own data (e.g. a pink noise generator block) or will represent a file of stored data (a WAV file reading block). The data that is read or generated is passed to the source block's output socket.

A **processing block** has both input sockets and output sockets. A processing block will take data from its inputs, perform some operation on the data and pass the result to its output sockets. For example an FFT block takes a frame

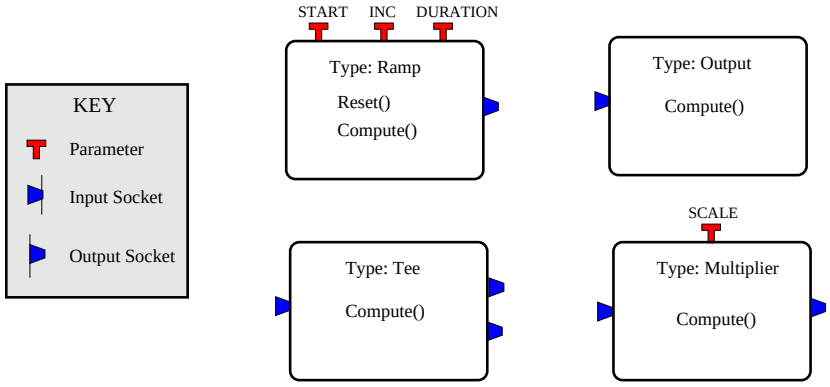

Figure 3.1: A collection of primitive blocks. Each primitive block embodies an algorithm. Block's may also have input sockets, output sockets, and user parameters

of data from its input, performs an FFT and passes the transformed frame to its output.

A **sink block** is a block that has inputs but no output sockets. Sink blocks generally represent output files or data displays.

System Sub-components

At the lowest level of description a CASA sytem is described by a set of primitive blocks, their parameters, and the connections that exist between them. However, when designing systems it may be more convenient to describe the system at a higher level as a set of sub-components which are themselves composed of primitive blocks. This may be convenient if identical sub-components are used more than once in the same system, or if the user wishes to reuse identical sub-components in other systems.

The toolkit allows the user to define a sub-component, or **intermediate block** in terms of the primitive blocks it is composed of. Once defined this sub-component may be used as though it were itself a primitive block.

This heirarchical sub-component design may be extended so that higher level intermediate blocks may themselves be composed of mixtures of primitive blocks and lower level intermediate blocks.

Describing a system

CASA systems and system sub-components (i.e. intermediate blocks) may be described using a simple script like language and stored in files with the extension '.ctk'.

A .ctk file may contain the description of one or more sub-components, or a

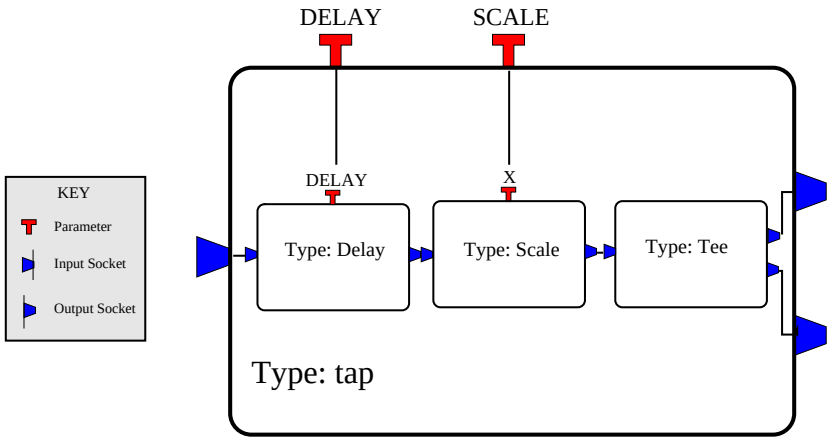

Figure 3.2: An intermediate level block defined in terms of primitive blocks.

complete system. In either case the description includes a list of all the blocks contained in the system or sub-component together with their parameter values and a description of the connections between their inputs and outputs.

Chapter 4 provides a tutorial describing in detail how to write CTK scripts.

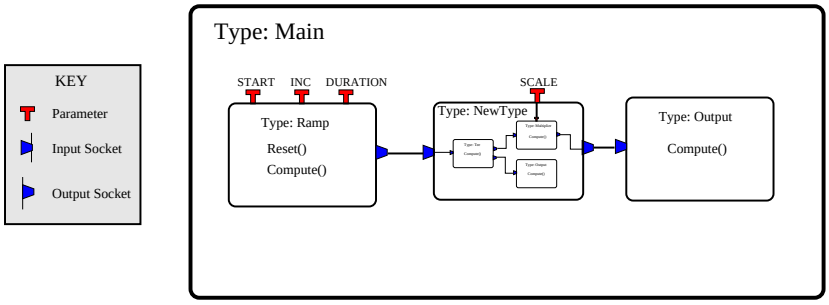

Figure 3.3: A complete system

The CTK Graphical User Interface (GUI)

Although writing scripts for small simple systems is straighforward, more complicated system can be hard to visualise from the linear script description. A graphical interface has been provided that allows the user to design systems and examine and edit existing scripts in an intuitive manner. The interface can also be used as a platform from which to interactively experiment with CTK systems.

Use of the CTK GUI is cover in Chapter 5.

Extending the toolkit

The toolkit is distributed with a library of existing primitive blocks. These primitive blocks are written in C++ and compiled into the toolkit. However, the toolkit is designed to be easily extendible. It is hoped that with the help of developer's documentation and the example of the source code for the existing inbuilt blocks, users with a little knowledge of C++ will be able to write their own inbuilt blocks for incorporation into the toolkit library.

Chapter 6 describes the necessary steps for adding a new primitive block to the system.

Chapter 4

Writing CASA Toolkit Scripts

This chapter is composed of a number of walk through tutorials which illustrate how to build simple systems using the **CASA Definition Language**.

4.1 Tutorial 1 - Displaying a spectrogram of superimposed sinusoids

In the first example a system will be built which superimposes three sinusoidal signals and displays the spectrogram of the resulting signal.

In any text editor open a new file and call it 'example1.ctk'.

At the top of the file type:

```
#!/usr/local/bin/CTKScript
```

This line tells the shell that the file is a script and should be interpreted by the `CTKScript` command in `/usr/local/bin/` . (If your CTK binaries are installed somewhere other than /usr/local/bin then make sure you use the appropriate path.)

Next start the description of a new processing block. On a new line type:

```
BLOCK main
```

This creates a new block and assigns it the name 'main'. A script file can define any number of blocks. The `CTKScript` command takes as an optional argument the name of the block to execute. By default it will search for a block called,

'main'. If the script contains only one top-level block then it makes sense to call this block 'main' so that `CTKScript` will work using the default block name argument.

The top-level block main describes the entire system and is composed of a number of interconnected lower-level blocks. These lower-level blocks can be either **intermediate-level** blocks (themselves composed of a network of lower level blocks) or one of the fixed library of lowest level **inbuilt** blocks.

For this example we must start out by adding three inbuilt sine-wave generator blocks. To do this we use the CTK script `ADD` command. Type:

```
ADD i1=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=2)
ADD i2=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=3)
ADD i3=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=4)
```

These `SineWave` blocks are **generators** as they have signal outputs (1 each) but no signal inputs. They have tunable parameters and in this case the parameters are set to produce sinusoids of 1 second duration at a sample rate of 1000 samples/second and with frequencies of 2, 3 and 4 Hz respectively.

We now need a way of superimposing these signals. This is done using the inbuilt block called `Adder` This takes an arbitrary number of inputs and sums them to produce one output. Type:

```
ADD a=Adder(NINPUTS=3)
```

By default `Adder` expects two inputs. In this case we have 3 sinusoids to add, so the `NINPUTS` (number of inputs) parameter has been set to 3.

We can now request a graphical display of the superimposed waveform. The basic mechanism for graphical output is to use the inbuilt block `Display` which generates a Qt-based output window. An alternative for users with MATLAB is to use the `MDisplay` block which uses the CTK/MATLAB interface to create a graphic in a MATLAB window. So next type either:

`ADD d1=Display` (Qt-based output)

or

`ADD d1=MDisplay` (MATLAB-based output)

As well as specifying the system's subblocks we must also describe how they are connected. By default the toolkit will try to connect the blocks in series in the same order as they occur in the script. If - as in this case - this is not what is wanted, then connections must be specified explicitly. In this example each of the sine wave generators must connect to one of the three inputs on the adder block, and the output of the added block must connect to the display block. To make these connections add the lines:

```
CONNECT i1:out1 a:in1
```

```
CONNECT i2:out1 a:in2
CONNECT i3:out1 a:in3
```

The first line connects the output socket named `out1` of the block called `i1` to the input socket named `in1` of the Adder block called `a`. The second and third lines likewise connect inputs `i2` and `i3` to the 2nd and 3rd inputs of the Adder block `a`. Note, for all blocks the input sockets will be named `in1`, `in2`, etc and the output sockets named `out1`, `out2`, etc. [1] If a block has only 1 input socket, or one output socket then the socket name can be omitted from the `CONNECT` statement without any ambiguity. So in this case we could more simply write:

```
CONNECT i1 a:in1
CONNECT i2 a:in2
CONNECT i3 a:in3
```

To connect the adder block to the display block we could add the line:

```
CONNECT a d1
```

But in this case there is no need to make this connection explicitly as it follows the default behaviour of connecting blocks in series in the order in which they are defined in the script. So this line can be omitted.

Finally add the line:

```
ENDBLOCK
```

This marks the end of a block definition.

We have now defined a simple system that will add three sinusoids and display the resulting waveform. This system is illustrated in Figure 4.1. We can now 'execute' the script and see the result. From the editor first save the script, and then in the Unix shell window make the script executable by using the Unix chmod command:

```
chmod u+x example1.ctk
```

Now simply execute it by typing:

```
example1.ctk
```

[1] At present in the case of multiple input/output sockets it may be necessary to refer to the block's documentation to see how to correctly connect up the block. In future versions nicknames may be associated with these raw input and output names to make their meaning more transparent.

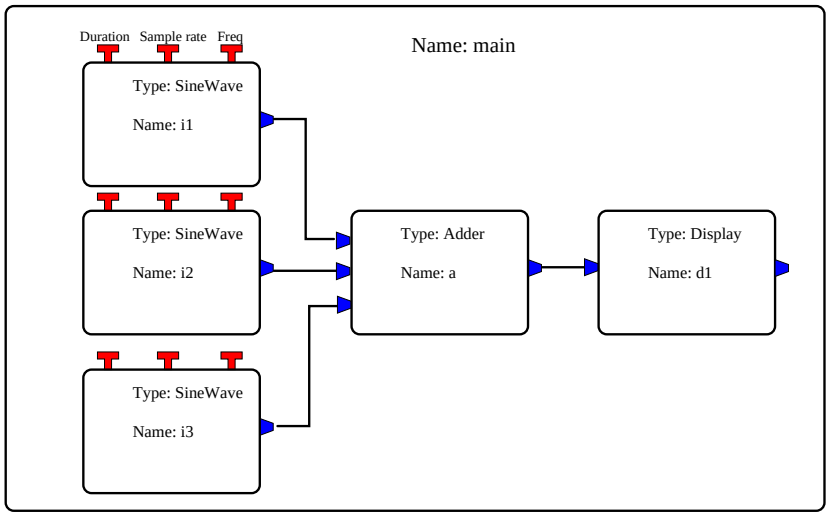

Figure 4.1:

A window should appear displaying a periodic waveform. If the MATLAB interface display is used (i.e. `MDisplay`) there will be a short pause while the MATLAB engine starts up.

The waveform is the addition of sinusoids with frequencies of 2, 3 and 4 Hz. By changing the `FREQ` parameters of the `SineWave` blocks and rerunning the script you can see the effect of adding sinusoids with different frequencies. Alternatively you can set the sinusoid frequencies from the command line. To do this you must first edit the script so the `SineWave` blocks are defined as:

```
ADD i1=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=$1)
ADD i2=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=$2)
ADD i3=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=$3)
```

The $1, $2 and $3 are command-line variables. You can now execute the command by proceeding it with three argument values. For example, try typing:

```
example1.ctk 2 3 4
```

or

```
example1.ctk 100 200 300
```

or with any other three frequency values after the command name.

We now wish to make a spectrogram of the signal. To do this we need to break the signal into a series of windowed frames and calculate the magnitude of the complex FFT of each frame. This is done using the two more inbuilt blocks,

named `Frame` and `FFT`, connected together in series. However, a spectrogram is a commonly used representation and it is rather obscure and cumbersome to have to keep writing out its inbuilt block definition in full every time we wish to use it. e.g. to construct a spectrogram we need to add something like the following to the script:

```
ADD frame=Frame
ADD fft=FFT
```

and it would be better is we could simply type:

```
ADD s=Spectrogram
```

We can in fact use this neater alternative if we first define an intermediate level block called `Spectrogram` in terms of the the inbuilt blocks `Frame` and `FFT`. A number of useful intermediate level block definitions (including the spectrogram block) come supplied with the toolkit and are stored in the toolkit scripts directory. So rather than defining our own `Spectrogram` intermediate-level block we can simply use the `INCLUDE` command to add this prewritten block to our script. So directly before the line '`BLOCK main`,' type:

```
INCLUDE $CTKROOT/scripts/spectrogram.ctk
```

Now add the `Spectrogram` block at the end of the main block definition (i.e. immediately before the `ENDBLOCK`)

```
ADD s=Spectrogram
```

We do not need a `CONNECT` command as by default the `Spectrogram` block will take input from the proceeding `Display` block (`Display` blocks are designed to both display their input and *pass it on to their output*). Finally we want to display the output of the `Spectrogram`. So we need to add another display block. For Qt-based output type:

`ADD d2=Display` (Qt-based output)

and for MATLAB output, first add:

```
ADD d2=MDisplay(BEFORE_PLOT="subplot(2,1,2)")
```

and then edit the definition of block `d1` to be

```
ADD d1=MDisplay(BEFORE_PLOT="subplot(2,1,1)")
```

The `BEFORE_PLOT` parameter of the `MDisplay` allows the MATLAB plot to be customised by issuing MATLAB commands that are executed immediately before the plot is generated.[2] In this case it is used to make the waveform and the spectrogram appear as subplots in a single window.

The script is now complete. If you now go back to the Unix command shell and type:

```
example1.ctk 10 100 200
```

a window showing both the waveform and the spectrogram something like that shown in Figure 4.2 should appear. If the script fails to run examine the error message and check your script against the script listed in Appendix A.1.

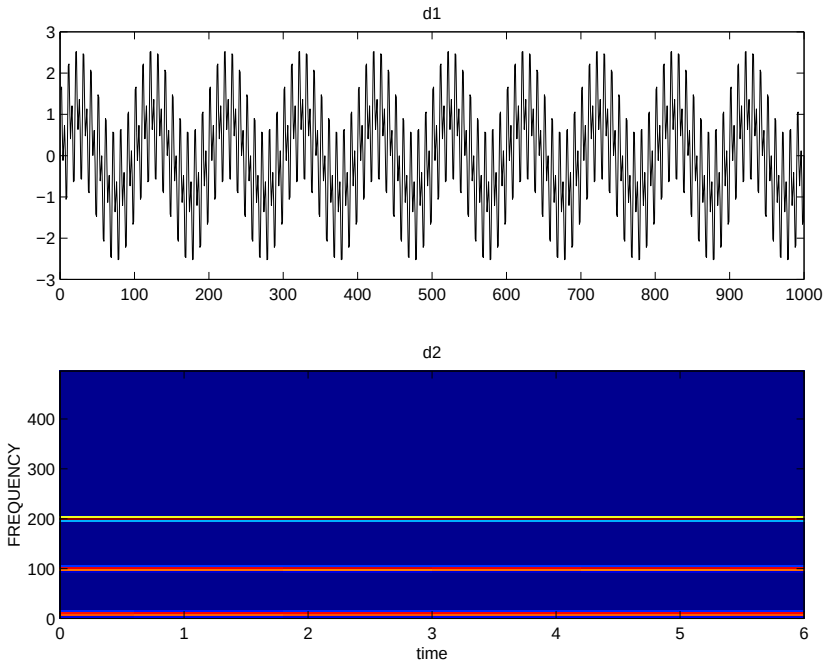

Figure 4.2:

CTK Data Flow

This example also serves to illustrate an important feature of the data flow underlying the CASA toolkit. Whereas the data entering the `Spectrogram` block is a simple signal with time as the only dimension (i.e. it is 1-dimensional) the data leaving the `Spectrogram` block has two dimensions, time and frequency. All data must have time as a dimension but on top of that it may have any number of other dimensions. Each inbuilt block is designed to examine the

[2] There is a corresponding `AFTER_PLOT` command that allows MATLAB commands to be executed immediately *after* the plot has been generated.

dimensionality of the data entering it and to act appropriately. So, in this example the output of the first `Display` block is a simple graph with time along the x-axis, while the output of the second `Display` block is a colour image map with time and frequency along the x and y axes. There is only one type of `Display` block but it behaviour alters according to the nature of its inputs.

4.2 Tutorial 2 - Designing an intermediate level block

In the previous tutorial we saw how we can use an intermediate level block to perform the function of several connected inbuilt blocks. We saw how an intermediate block called Spectrogram was used which was composed of the series combination of a `Frame` block and an `FFT` block.

Why go to the trouble of defining an intermediate blocks when we could just write it out longhand in terms of inbuilt blocks? The obvious advantage is that it simplifies scripts making them both shorter and easier to read and write. However, a far more important advantage of intermediate blocks is that they allow a degree of 'design reuse'. Intermediate blocks definitions can be written and stored in a library. These intermediate blocks can then be used as parts of a more complex system through use of the INCLUDE mechanism. So a simpler unit, such as a spectrogram, can be designed once and then reused in any number of more complex systems. It should be noted that an intermediate level block may itself be designed in terms of other intermediate level blocks and there is no limit to the level of such nesting that may be used.

In the previous tutorial we used the *pre-defined* `Spectrogram` intermediate block that is provided as part of th CTK intermediate block library. All we had to do was add the appropriate `INCLUDE` line at the top of our script file then we could use `Spectrogram` exactly as if it was one of the toolkit inbuilt blocks. In this tutorial we will see how to define our own intermediate level blocks so that they can be used like inbuilt blocks in this way.

In this tutorial we will construct a lossy delay line like that shown in Figure 4.3. Each block in the delay line performs a delay operation, a scaling operation, and a teeing operation to split the data flow into two outputs. There is no one inbuilt block that can perform all these operations in one go, instead we must build an intermediate block from the inbuilt blocks `Delay`, `Scale` and `Tee`. Figure 4.4 shows the architecture of this intermediate block.

So first we will construct the intermediate block shown in Figure 4.4 and give it a typename 'tap' and then we will construct the delay line by arranging these tap blocks in series as in Figure 4.3.

In any text editor open a new file and call it 'example2.ctk'.

At the top of the file type:

```
#!/usr/local/bin/CTKScript
```

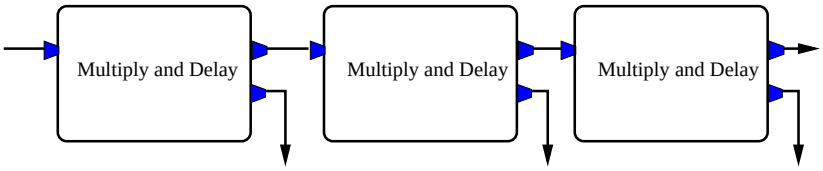

Figure 4.3:

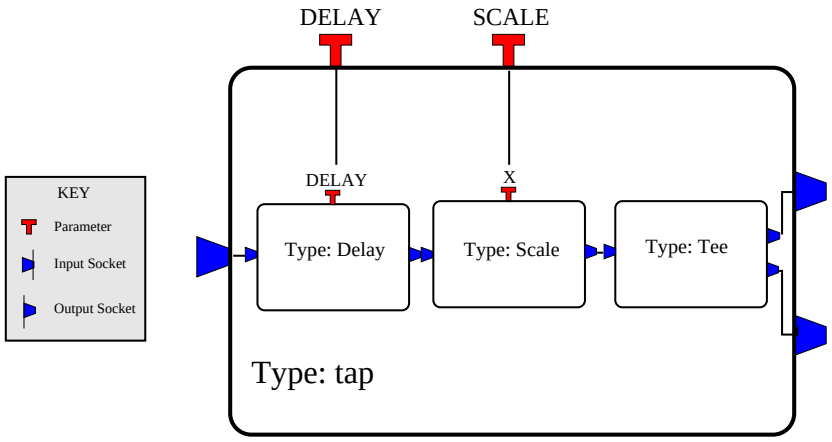

Figure 4.4:

Next we need to start the description of a new block to which we will give the typename 'tap'. To do this we just type:

```
BLOCK tap
```

Now we add the subblocks `Delay`, `Scale`, and `Tee` which go together to make up the tap block. To do this we just add the lines:

```
ADD d1=Delay(DELAY=20)
ADD s1=Scale(X=0.5)
ADD t1=Tee
```

These blocks need to be connected in series. We do not need any explicit `CONNECT` statements because the default series connections will give the right result. However, once these three are connected there will still be a few loose connections, namely, the input of the `Delay` block and the two outputs of the `Tee` block. These connections are intended to be inputs and outputs of the intermediate block itself. This must be made explicit and is done so by use of the `INPUT` and `OUTPUT` commands. To handle the input add the line:

```
INPUT in1=d1
```

This declares that the intermediate has an input socket and this socket feeds into the input of the subblock called `d1` i.e. it feeds into the `Delay` block. Likewise for the outputs we need to add the lines:

```
OUTPUT out1=t1:out1
OUTPUT out2=t1:out2
```

This declares that the intermediate block has two output sockets and they are respectively output socket 1 and output socket 2 of the block called `t1` (i.e. the `Tee` block).

We can now complete the block definition with the line:

```
ENDBLOCK
```

With this definition the Tap block has no parameters. The delay and scale factor will be fixed at 20 and 0.5 as stated in the definition of its `Delay` and `Scale` subblocks. If we want tap blocks with adjustable delays and scale factors we must make the parameters of its subblocks visible as parameters of the tap block itself (see on Figure 4.4 how tap has a parameter `DELAY` that is linked to the parameter `DELAY` of its subblock `Delay`). To do this we use the `PARAMETER` command. So just before the `ENDBLOCK` insert the lines:

```
PARAMETER DELAY=d1:DELAY
PARAMETER SCALE=s1:X
```

This gives the tap block parameters `DELAY` and `SCALE`, and attaches these to the `DELAY` parameter of the `Delay` block and the X parameter of the `Scale` block respectively.

Now that the tap block has been defined it is ready to be used in the construction of the main block. The main block will have a `SineWave` source block, then a series of tap blocks. Add to following lines:

```
BLOCK main
  ADD i1=SineWave(DURATION=0.5, FREQ=5, SAMPLE_RATE=1000)
  ADD tap1=tap
  ADD tap2=tap
  ADD tap3=tap
  ADD tap4=tap
  ADD tap5=tap
```

The default connections will string these blocks together in series. Now lets display the signal at a couple of different points along the delay line. To do this we simply add a couple of display blocks and connect them to the delay line using the spare outputs of the tap blocks. For example if you are using the MATLAB interface add the following:

```
ADD first_output=MDisplay(BEFORE_PLOT="subplot(2,1,1)")
ADD second_output=MDisplay(BEFORE_PLOT="subplot(2,1,2)")
```

or if you are using the Qt-based graphical output add:

```
ADD first_output=Display
ADD second_output=Display
```

If you have no graphical output you can monitor the output on the terminal using:

```
ADD first_output=Output
ADD second_output=Output
```

Then to connect the outputs to the delay line add the following:

```
CONNECT tap1:out2 first_output
CONNECT tap4:out2 second_output
```

Here we have connected the display to `tap1` and `tap4`, but they could equally well be connected at any two other positions.

And finally finish the main block definition by adding:

```
ENDBLOCK
```

The example is now ready to be run. To do this you first need to make sure the script file is executable. In the Unix command shell type:

```
chmod u+x example2.ctk
```

Then to run the script, simply type:

```
example2.ctk
```

If all goes well two sinewaves should appear with one reduced in magnitude and delayed with respect to the other. Using the MATLAB interface this should look something like Figure 4.5. If the script fails to run then you have probably made a typing mistake. Examine the error message and check your script against the script listed in Appendix A.2.

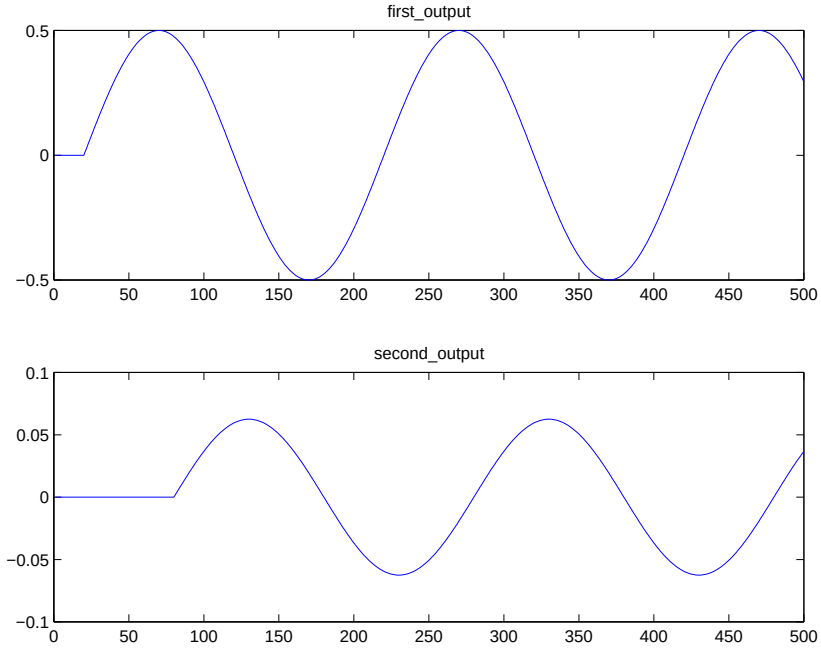

Figure 4.5:

4.3 Tutorial 3 - Missing Data Speech Recognition

In this final tutorial we will look at how the CASA toolkit can be used to run speech recognition experiments.

Figure 4.6 shows the block diagram for a system that performs 'missing data' speech recognition. At the bottom of the diagram is the block which performs

the 'missing data' Viterbi decoding. This is similar to a standard speech decoder but as well as taking a representation of the speech signal, it also has an input called a 'mask' which tells the decoder which elements of the representation may be considered to be reliable. Typically some form of spectro-temporal representation is used, and the mask should indicate which spectro-temporal elements have a favourable local SNR (i.e. which elements are 'clean').

Following Figure 4.6 we see that the first operation is to read the noisy speech from an AU sound file. In this example recognition will be based on a 'ratemap' representation. The ratemap is a kind of auditory-inspired spectrogram and is formed by passing the signal through a gammatone filterbank, and then performing some leaky integration and downsampling. The ratemap is then duplicated using a tee block and one copy passed directly to the decoder and the other copy is passed down the right hand side of the figure through the blocks that generate the mask. The mask is generated by performing a simple noise estimation and then using a comparator to set the mask to true at the spectro-temporal points where the estimated clean signal (i.e. the noisy signal after subtraction of the estimated noise) makes up over half the energy of the noisy signal. The mask is then passed into the second input of the decoder.

The CTK script to describe this process is listed in Appendix A.2. The script shown differs slightly from the diagram in that it omits the initial ratemap generation stage and instead reads in precomputed ratemaps. Note, that the default series block connection means that very few of the block connections need to be specified explicitly.

We will now see how this script can be used both for recognising a single utterance, or recognising a small corpus of test utterances. First make a new directory called example3 and copy the example3 CTK script supplied with the distribution into this directory

```
mkdir example3
cd example3
cp $CTKROOT/tutorial/example3.ctk .
```

The script takes two arguments: first, the name of the ratemap file to be recognised and second a string representing the correct transcription against which the result will be compared. A set of ratemaps for TIDigits mixed with factory noise at various SNRs is distributed with the toolkit under the `$CTKROOT/data` directory. So, to test the script on the utterance 1159 at 10dB SNR type:

```
example3.ctk $CTKROOT/data/factory/1159a.10 1159 ""
```

After typing this there will be a short pause while the HMM files are read in. Once the HMMs have been read a few lines appear summarising the HMM data. After this, recognition will commence and digits should appear on the screen as they are recognised. When recognition is complete a line of statistics will appear summarising the systems performance. The final output should look something like this:

```
(yelp)85% example3.ctk $CTKROOT/data/factory/1159a.10 1159
num_HMMs = 12
num_states = 8
num_mixes = 10
vec_size = 64
num_dist = 960
1157  1159   Nin: 4 Nout: 4 (H=3 I=0 D=0 S=1) Cor: 75 Acc: 75 --  Ni 4 Nout: 4 (H=3 I=0 D=0 S=1) Cor: 75 Acc: 75 --
```

This shows that the utterance '1159' was recognised as '1157'. The statistics on the left show that 4 digits were 'read in' and 4 digits were output; there were 3 hits, no insertions, no deletions and 1 substitution. Word correctness is 75% and word accuracy is 75%.

The system seems to work well with this single utterance, but to really test it we need to run it over a large test set of several hundred utterances. One approach would be to write a shell script to repeatedly execute the `example3.ctk` command each time with a different ratemap file and a different transcription string. However, there are two problems with this. First, reading in the HMMs represents a sizeable computational overhead. We do not want to have to reread the HMMs for every utterance to be recognised. Second, there is no convenient way of calculating the *overall* performance if we have invoked `example3.ctk` separately for each utterance to be recognised. In order to overcome these problems we can use the CTK **script argument list** mechanism.

Script argument lists

A script argument list is a text file where each line supplies the arguments for a separate invokation of the block process described by the script. These argument lists are introduced to the script on the command line using `CTKScript`'s `-S` option. If a script requires more than one argument these arguments can all be specified in a single argument list containing several columns, or they can be split across multiple argument lists with fewer columns. When more than one argument list is used each list is introduced on the command line with a separate `-S`. An example makes this clear. Make your window as wide as possible and then try the following command:

```
  example3.ctk -S $CTKROOT/data/flists/test240.10.flist -S $CTKROOT/data/
transcripts/transcripts_240 ""
```

The first `-S` parameter introduces the argument list containing the paths of the complete test set of 240 files to be recognised. The second argument list is a list of the correct transcriptions of these 240 utterances. Note, all the argument lists should have the same length, if they do not then an error will be reported.

This command should produce output that starts like that below:

```
(yelp)333% example3.ctk -S $CTKROOT/data/flists/test240.10.flist -S $CTKROOT/data/transcripts/transcripts_240
num_HMMs = 12
num_states = 8
num_mixes = 10
vec_size = 64
num_dist = 960
1157     1159    Nin: 4 Nout: 4 (H=3 I=0 D=0 S=1) Cor: 75 Acc: 75 --       Nin: 4 Nout: 4 (H=3 I=0 D=0 S=1) Cor: 75 Acc: 75 --
12773o73 1273o73 Nin: 7 Nout: 8 (H=7 I=1 D=0 S=0) Cor: 100 Acc: 85.7143 -- Nin: 11 Nout: 12 (H=10 I=1 D=0 S=1) Cor: 90.9091 Acc: 81.8182 --
1627     127     Nin: 3 Nout: 4 (H=3 I=1 D=0 S=0) Cor: 100 Acc: 66.6667 -- Nin: 14 Nout: 16 (H=13 I=2 D=0 S=1) Cor: 92.8571 Acc: 78.5714 --
6128o    128o    Nin: 4 Nout: 5 (H=4 I=1 D=0 S=0) Cor: 100 Acc: 75 --      Nin: 18 Nout: 21 (H=17 I=3 D=0 S=1) Cor: 94.4444 Acc: 77.7778 --
12       12      Nin: 2 Nout: 2 (H=2 I=0 D=0 S=0) Cor: 100 Acc: 100 --     Nin: 20 Nout: 23 (H=19 I=3 D=0 S=1) Cor: 95 Acc: 80 --
```

To recognise the full set of 240 utterances may take some time. The recognition can be halted at any time by using the `Ctl-C` key. The statistics on the left of the screen apply to the current utterance, and those on the right are a running total showing the performance so far. The final recognition accuracy after all 240 utterances have been processed should be about 84%.

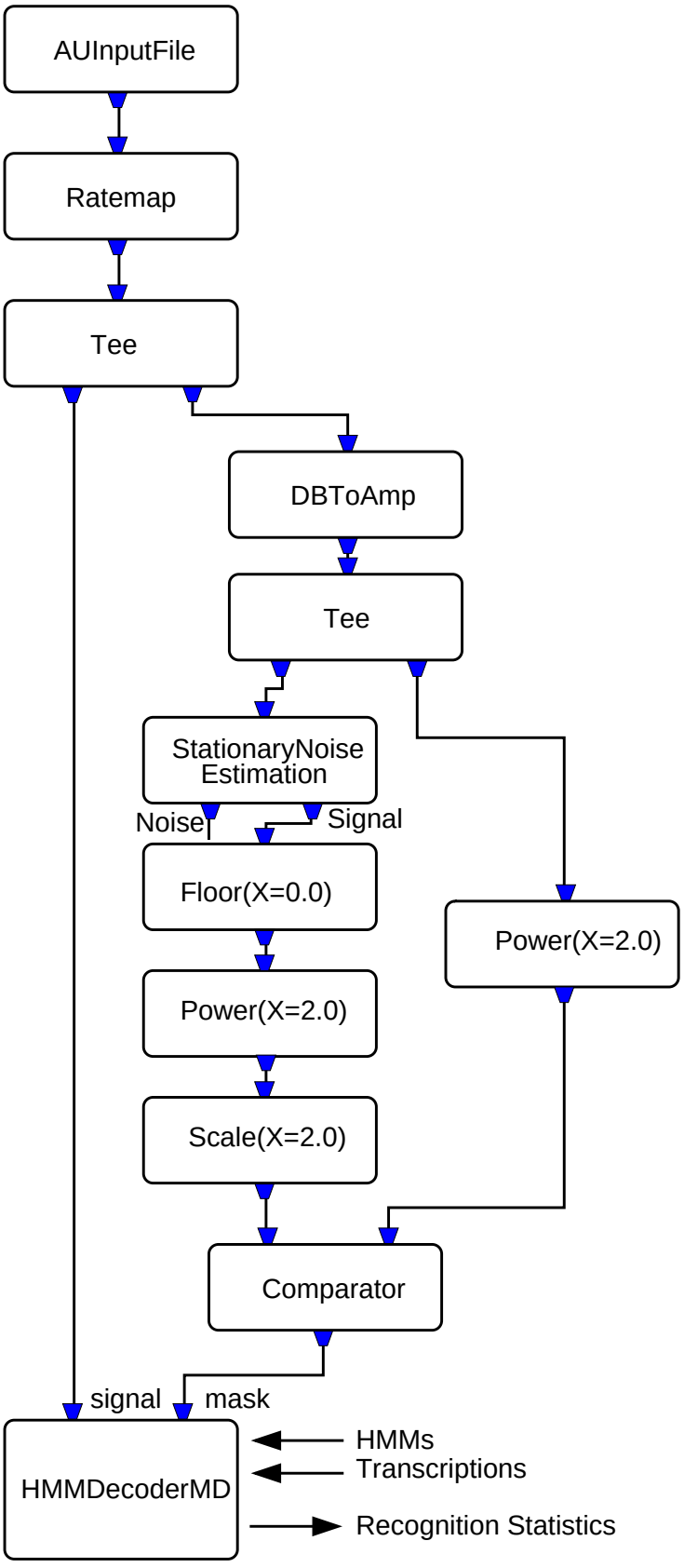
AUInputFile
Ratemap
Tee
DBToAmp
Tee
StationaryNoise Estimation
Noise
Signal
Floor(X=0.0)
Power(X=2.0)
Power(X=2.0)
Scale(X=2.0)
Comparator
signal
mask
HMMDecoderMD
HMMs
Transcriptions
Recognition Statistics

Figure 4.6:

Chapter 5

Using the CTK Graphical User Interface

MORE STUFF HERE

Chapter 6

Extending the CTK Inbuilt Block Library

This chapter provides some brief notes to help user's who wish to add their own primitive blocks to the toolkit.

6.1 Setting Up Your Local CTK

Before you add your own blocks to CTK you have to make your own local version of the CTK binaries. This is done by following the steps:

1. First you have to make your own local CTK directory. For example:

   ```
   mkdir /home/jon/CTK
   ```

2. Then if your shell is csh or tcsh edit your .login and change the definition of `CTKLOCAL` to point to this directory:

   ```
   setenv CTKLOCAL  /home/jon/CTK
   ```

   If your shell is bash, ksh, zsh or sh then edit your .profile to include the line:

   ```
   CTKLOCAL=/usr/local/CTK/local
   ```

3. Now copy the files under `$CTKROOT/local` to `$CTKLOCAL`:

   ```
   cp -r $CTKROOT/local/* $CTKLOCAL
   ```

4. Now test that it is working by trying to rebuild the binaries locally:

```
cd $CTKROOT/src
make
```

5. Finally edit your `$PATH` to add the `$CTKLOCAL/bin` to your path making sure it precedes `$CTKROOT/bin`.

6.2 Writing Your Own Primitive Block

The within `$CTKLOCAL/src` there are files named `ctk_NEW_BLOCK_TEMPLATE.cpp` and `ctk_NEW_BLOCK_TEMPLATE.hh`. These files provide an annotated template for the code you need to implement when writing your own inbuilt blocks. The files `my_blocks.cpp` and `my_blocks.hh` contain code for a few simple example blocks. Further examples are in the `$CTKROOT/src` directory.

More detailed CTK developers documentation should eventually be available.

Once you have written the .cpp and .hh files for the block you wish to add you need to perform the following three steps:

1. Add your new .hh file to the list of includes in the file `my_block_headers.hh`.
2. Add a line to the file `my_translation_table` according to the instructions it contains.
3. Edit the file `Makefile.local` in `$CTKLOCAL/src` and change the definition of `OBJS` to include the file you have added.

Once these steps have been completed move to the directory `$CTKROOT/src` and type 'make' and the local CTK binaries should now be rebuilt to include your new block in the CTK library. To check that the rebuild has been successful you can type 'ctk' to start up the GUI and check that your block appears correctly in the pull-down block menu.

Appendix

Appendix A

Scripts for User Guide Tutorials

A.1 Script for Tutorial 1

A.2 Script for Tutorial 2

A.3 Script for Tutorial 3

```
#!/usr/local/bin/CTKScript

#*************************************************************
### Tutorial 1 - A spectrogram of the addition of 3 sinusoids

INCLUDE $CTKROOT/scripts/spectrogram.ctk

BLOCK main

  ADD i1=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=$1)
  ADD i2=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=$2)
  ADD i3=SineWave(DURATION=1, SAMPLE_RATE=1000, FREQ=$3)

  ADD a=Adder(NINPUTS=3)

  ADD d1=MDisplay(BEFORE_PLOT="subplot(2,1,1)");
  ADD s=Spectrogram
  ADD d2=MDisplay(BEFORE_PLOT="subplot(2,1,2)");

  CONNECT i1 a:in1
  CONNECT i2 a:in2
  CONNECT i3 a:in3

ENDBLOCK

#*************************************************************
```

Figure A.1: Tutorial 1

```
#!/usr/local/bin/CTKScript

#*************************************************************
### Tutorial 2 - The Delay Line

BLOCK tap
   ADD d1=Delay(DELAY=20)
   ADD m1=Scale(X=0.5)
   ADD t1=Tee

   INPUT in1=d1
   OUTPUT out1=t1:out1
   OUTPUT out2=t1:out2

   PARAMETER SCALE=m1:X
   PARAMETER DELAY=d1:DELAY

ENDBLOCK


BLOCK main
   ADD i1=SineWave(DURATION=0.5, FREQ=5, SAMPLE_RATE=1000)
   ADD tap1=tap
   ADD tap2=tap
   ADD tap3=tap
   ADD tap4=tap
   ADD tap5=tap

   ADD first_output=MDisplay(BEFORE_PLOT="subplot(2,1,1)")
   ADD second_output=MDisplay(BEFORE_PLOT="subplot(2,1,2)")

   CONNECT tap1:out2 first_output
   CONNECT tap4:out2 second_output

ENDBLOCK

#*************************************************************
```

Figure A.2: Tutorial 2

```
#!/usr/local/bin/CTKScript

#*************************************************************
### Tutorial 3 - Missing Data Speech Recognition

BLOCK main

  ADD i = BinaryInputFile(FILE_NAME=$1, HEADER_SIZE=256, BYTES_PER_SAMPLE=4,
SWAP_BYTES=Yes, SAMPLE_RATE=100, SAMPLES_PER_FRAME=64, FLOATING_POINT=Yes)

  ADD tee1 = Tee

  ADD dbtoamp = DBToAmp
  ADD tee2 = Tee

  ADD nest=StationaryNoiseEstimation(NFRAMES=10);

  ADD floor = Floor(X=0.0)
  ADD pow2 = Power(X=2.0)
  ADD scale = Scale(X=2.0)
  ADD comp = Comparitor(X=0.0)

  ADD decoder = HMMDecoderMD(HMM_FILE="mlist", LABELS="123456789ozs",
SILENCE="s", USE_ERF_TABLE=T, USE_BOUNDS=T, WORD_PENALTY=0, TRANSCRIPTION=$2)

  ADD pow1 = Power(X=2.0)

  CONNECT nest:out2 floor
  CONNECT tee2:out2 pow1
  CONNECT pow1 comp:in2
  CONNECT tee1:out1 decoder:in1

ENDBLOCK

#*************************************************************
```

Figure A.3: Tutorial 3