002

TR-IT-0225

The ASSM toolkit for Polynomial Segment Models and Automatic Unit Design

Michiel Bacchiani

1997.06

This technical report describes the software that was produced to design and use a speech recognition system, based on Polynomial Segment Models (PSMs). The toolkit offers a variety of tools for the training, classification and recognition using PSMs. It also provides tools specifically aimed at design of automatically derived acoustic units. The software described in this report covers the available software package (ASSM) at this time, June 1997.

目次

1 Introduction

This technical report describes the software that was produced to design and use a speech recognition system, based on automatically derived acoustic units. The software described in this report covers the available software package (ASSM) at this time, June 1997. The algorithms were developed between March 1995 and January 1996 and are described in detail in technical report TR-IT-0147, "Speech Recognition System Design using Automatically Learned Non-Uniform Segmental Units" by M. Ostendorf, M. Bacchiani, Y. Sagisaka and K. Paliwal. The initial implementation of these algorithms in software were described in technical report TR-IT-0146, "Software for Design and Use of a Speech Recognition System based on Automatically Derived Non-Uniform Units". Although that software package was functional and allows the user to execute the developed algorithms, it was somewhat limited in terms of ease of use, ability to deal with large amounts of data and possibilities for expansion. Furthermore, related to the limited ability to deal with large amounts of data, some algorithmic steps quickly became computational infeasible. To alleviate the computational complexity problem, another tool (a lattice decoder) was developed at ATR in the period from May 1996 until June 1996. This made the tools computationally feasible but the other problems remained. A re-implementation of the software was started September 1996 at Boston University and was completed January 1997. In addition, since January 1997, some tools were developed to make pronunciation modeling in terms of automatic units possible. These new tools are far from complete but are included in the package as they are an implementation of a very simple form of pronunciation modeling and might therefore be useful.

The basis of the improvements in ease of use, better ability to deal with large amounts of data and possibility for expansion are mainly due to a change of the software model. The new ASSM package is based on the client-server model, where most applications are implemented as a client process which relates to the ASSM server process for many of the desired computational services.

The first section of this report describes the use of the developed tools. It explains how to setup the server and subsequently run an experiment. The second section of the report describes how new applications can be developed. Note that one can use any programming language as long as the language provides interfaces to the socket system calls. The third section of the report describes some of the structure implemented in the server to make it clear how the server can be expanded.

2 ASSM tools

This section describes the implemented tools in the ASSM package and how to use them. Many of the tools require services provided by the ASSM server. The following will therefore first describe how the server process is to be started and configured for running an experiment. For details on site-configuration, compilation, installation and file locations of the toolkit please refer to the Appendix 付録 B . For details on file formats, please refer to the Appendix 付録 A .

2.1 Running the server

The server program is named `ASSMserver`. Starting the server involves nothing more than just executing this executable. The server will immediately do a fork and the parent process will terminate, meaning that your prompt will immediately return. The fork will cause the `init` process to inherit the server process as a child which avoids creating zombie processes. The server will also change the current directory to the root directory in order to avoid occupying a temporary mounted file-system. To check if the server is running, use a system tool to list the running processes. To check the status of the server process, one can look at the log file, which at this point should look like:

```
Wed Mar 19 17:42:57 1997
 finch.bu.edu 11828 ASSMserver: Server initializing
Wed Mar 19 17:42:57 1997
 finch.bu.edu 11828 ASSMserver: Unix domain listen socket created
Wed Mar 19 17:42:57 1997
```

```
finch.bu.edu 11828 ASSMserver: TCP/IP listen sockets created
```

At this point, the server is sleeping and will continue to listen for connections from client processes (applications).

2.2 Configuring the server

After a server process is created, it is time to configure the parameters of the server. For this purpose, two applications are provided. One application named `ASSMservermaint` allows the user to interactively configure the server. Another named `fileinit` allows the user to configure the server with a parameter file.

(2.2.1) Interactive configuration

To configure the server running on the local host, simply run the application named `ASSMservermaint`. To configure a server running on another host, give the host-name as an argument. This will bring up a menu looking like:

```
UDS connection: 'ASSMserver' server pid 11867, uid 44772

ASSM server maintenance commands:
  1. Feature vector search parameters
  2. Label search parameters
  3. Sufficient statistics search parameters
  4. Model search parameters
  5. Slave server parameters
  6. Server output/computation parameters
  7. Exit

Command:
```

Options 1 trough 4 allow the user to specify where the server should look for files. Option 5 allows the user to manage a multi-server experiment setup. Option 6 allows the user to specify where the server should write results and allows the user to set the values of parameters needed in computations performed by the server. These options will be described in more detail in the following sections.

Guiding the search for files In general, files within the ASSM package are assumed to be named without extensions or directories. If a tool requires two types of files, the matching files are expected to have the same name but might reside in different directories and/or have different extensions. If a tool requires filenames, only the core names are to be given. The full path of a particular type of file will be constructed by use of the appropriate directory and extension. Exceptions are those tools that perform tasks such as the translation of one format to another.

Options 1 through 4 of the top-level menu allow the user to specify where the server should look for files. Among others, it allows the user to specify in which directories the server should look, what extensions to try and possibly which master files (a master file is way to bundle a number of data files in one file for efficiency) are to be used in the search. After descending through option 3 for example, the next menu looks like:

```
ASSM server sufficient statistics commands:
  1. List sufficient statistics search paths
  2. Add sufficient statistics search path
  3. Remove sufficient statistics search path

  4. List sufficient statistics search extensions
  5. Add sufficient statistics search extension
  6. Remove sufficient statistics search extension
  7. List sufficient statistics MasModelFiles
  8. Add sufficient statistics MasModelFile
  9. Remove sufficient statistics MasModelFile
 10. Spec sufficient statistics MasModelFile
 11. Exit

Command:
```

Option 1 will show which search paths are currently used by the server. Option 2 allows the user to add a search path. Option 3 allows the user to remove a search path. Similarly, options 4 through 6 allow manipulation of the file extensions. When an application attempts to open a file, the server will sequentially go through the list of paths. For each path it will sequentially go through the list of extensions and attempt to open the created filename. It will continue until it either successfully opened the file or until it runs out of possibilities in which case it will report an error to the application. Option 7 through 10 allow the user to manipulate the list of master files that the server can use in searching. If a master file is specified, the server will first attempt to find a file in the master file. If this fails, it will proceed with the search procedure described above.

Multi-server configuration As clustering experiments quickly become very demanding in terms of storage and computation requirements and as this type of process is not parallelizable (after repartitioning, clusters are re-estimated using the complete set of data), the package allows the user to utilize a number of hosts in parallel. The network will consist of a master server which is connected to one or more slave servers. Slave servers can of course also function as a master to other slaves. To set up a "network" of servers, one should configure the server that is going to function as the master server at some level. Then by descending down option 5 of the top-level menu, the user can define which slave servers this master server can use. The next menu will look like:

```
Connected to 0 slave servers
        SlaveServerId   MachineName             Load    UserId  ProcessId

ASSM slave server commands:
  1. Add a slave
  2. Remove a slave

  3. Exit

Command:
```

To add a slave, one is prompted for the host-name of the machine the slave is running on and what the load of the slave should be. More details on load-balancing can be found in the section that describes the `km-las` tool.

Server output and parameters Directions on where to write output and what parameters to use in computation can be given by descending through option 6 of the top-level menu. The following menu will look like:

```
Current ASSM server parameters:
   1. Polynomial order     : 0
   2. Covariance type      : FULLCOV
   3. Distance type        : SQ_EUCLID
   4. Frame offset         : 128000
   5. Variance floor       : 1e-06
```

```
 6. Cluster perturbation : 0.01
 7. Number of regions    : 1
 8. Duration weight      : 1
 9. Output label dir     : /
10. Output label exts    :
11. Output label MLF     :
12. Output stats. dir    : /
13. Output stats. ext    :
14. Output stats. MMF    :
15. Output PSM dir       : /
16. Output PSM MMF       :
17. Output div. tree dir : /
18. Master server load   : 1
19. No changes

Command:
```

The covariance option affects both the type of Polynomial Segment Models (PSMs) that are to be generated as well as the type of sufficient statistics that are to be produced. The description of a PSM model is defined by options 1, 2 and 7. Option 1 sets the polynomial order of the PSM, option 2 sets the covariance type of the model (either full or diagonal). Option 7 specifies the number of regions the PSM models will have. A region in a segment model plays a similar role as a state in an Hidden Markov Model. Most other options are self explanatory and the ones that are not will be described in more detail in the description of the tools in the package.

(2.2.2) Configuration files

An alternative to interactive server configuration is to use configuration files. The application `fileinit` allows the user to configure the server by using a configuration file. A detailed description of the format of such a configuration is described in detail in the Appendix 付録 A . A quick reminder of the format is provided by a skeleton file “ConfigFileSkel”. It allows the user to set all of the options described in the interactive configuration tool. Running `fileinit` without any arguments will give the following message:

```
USAGE: fileinit [options] configuration_file

 Option                                       Default

 -a s     Serving host                         current
```

The configuration file given as an argument to this program will be used to initialize the server running on the local host unless another host is specified through the a switch.

2.3 Acoustic segmentation using the asegm tool

The acoustic segmentation tool performs a dynamic programming (DP) search in order to find the segmentation resulting the minimum distortion under a set of given constraints. The model for distortion computations is a Polynomial Segment Model (PSM) for which the parameters are set in the server (polynomial order, number of regions and covariance type). The actual used distortion measure is also defined in the server (squared Euclidean or Mahalanobis). For the Mahalanobis distance, the distortion of a segment is the -log likelihood of that segment for a model derived by ML estimation using only that segment as training material. In addition, to avoid data sparsity problems, the assumption is made that the covariance used in the model is invariant within the utterance and is therefore derived from the utterance as a whole rather than from individually hypothesized segments.

The constraints the DP is subjected to are formed by a minimum and maximum segment length and either a average distortion per frame threshold, a fixed number of segments or a given segmentation. If an average distortion per frame threshold constraint is imposed, the DP search finds the minimum distortion segmentation under minimum and maximum segment length constraints. If a fixed number of segments constraint is imposed, the DP search finds the minimum distortion segmentation for the given number of segments. If a segmentation A is given two types of constraints can be imposed.

1. The DP search finds either the minimum distortion segmentation under the constraint of the number of segments in A
2. The DP search finds the minimum distortion segmentation under the constraints that:
    (a) Segment boundaries coincide with segment boundaries in A.
    (b) A fixed number of segments for each segment in the given segmentation A, specified on a per segment basis, is used.

When the acoustic segmentation tool is executed without any arguments, it produces the following output:

```
USAGE: asegm [options] filenames...

 Option                                    Default

 -a s     Serving host                      current
 -d n f1 f2....fn
          Set n distortion threshold(s) to
          f1,...,fn                         off
 -e       Use input labelfiles              off
 -f s     Use regex for label matches       *
 -m i     Set minimum segmentlength         2 frames
 -n i     Set maximum segmentlength         70 frames
 -o       Do not write label files          write
 -p       Do not write sufficient stats.    write
 -v       Verbose                           off
 -N i     Fix number of segments to i       off
 -S f     Set script file to f              none
 -W       Constraints define num. segments  boundaries
 -X s     Set constraint file dir to s      /
 -Y s     Use constraint file with ext. s   none
 -Z s     Use constraint file MLF list s    none

FILES            | REQUIRED | LOCATED THROUGH
---------------------------------------------
feature vectors  |  yes     |  server
label files      |  no      |  server
constraint files |  no      |  X,Y,Z switches
```

A segmentation is given in the format of a label file where the label identities can be ignored.

The a switch allows the user to specify the host on which the server that is to be used is running. By default, the server is expected to be running on the same host that asegm is executed on.

The average likelihood per frame thresholds to be used in the acoustic segmentation can be set using the d switch. If for example segmentations for the thresholds 3 and 3.5 are desired, one would give the option -d 2 3 3.5. If the filename was foo, the resulting segmentations are then written in files foo.3 and foo.3.5. Where these files are written and which extension these files will have is determined by the parameters set in the server. Also note that the distance type that is used for the segmentation is controlled by the server parameter.

A fixed number of segments in the resulting segmentation can be imposed through the N option.

Constraining the DP search by means of a given segmentation can be performed by specifying a search path and/or extention and/or one or more masterfiles through the X, Y and Z switch. If in addition the W switch is set, the resulting segmentation will have the same number of segments as the given segmentation. Otherwise, the resulting segmentation will have segment boundaries coinciding with the given segmentation. The resulting segmentation will also be subject to a fixed number of segments constraint. The score field of the constraint label files (see description of the label file format in the Appendix 付録 A) are to be used to specify the number of segment constraints on a segment to segment basis.

If no acoustic segmentation is desired but one desires to compute sufficient statistics for a given segmentation one has to set the e switch. Where to find the corresponding label file is determined by the server parameters. In addition, one can define a regular expression through the f switch resulting in sufficient statistics computation only for those labels matching the regular expression.

2.4 Clustering using the km-las tool

The clustering program performs either divisive or K-means style clustering. The distance measure used in clustering is either a squared Euclidean distance, a Mahalanobis distance or a likelihood distance. Each cluster center represents a PSM model and therefore has a mean-trajectory, covariance and length distribution associated with it (a set in the case of multi-region PSMs). The distance type, covariance type and number of regions of the cluster centers (PSMs) is to be specified in the server. The duration model is not used in these experiments.

If the clustering program km-las is run without any arguments, it produces the following output.

```
USAGE: km-las [options] [-b ssflist | -c fvflist]

 Option                                      Default

 -a s     Serving host                        current
 -b s     Sufficient Statistics filelist      none
 -c s     Feature Vector filelist             none
 -d i     Feature Vector loading epoch size   5
 -e s     Label match regex                   none
 -f s     Initial model list                  none
 -g s     Model set basename                  ASWU
 -h s     Model basename                      NUU
 -i i     New model start index               1
 -k s     Cluster task description            dN1
 -m i f   Divisive clustering iterations      10 0.1
 -n i     Minimum cluster size in frames      25
 -v       Verbose                             off

FILES                  | REQUIRED | LOCATED THROUGH
----------------------------------------------------

features | suff. stats |   yes    |  server
labels(features only)  |   yes    |  server
```

```
PSMs                    |  no    |  server
```

The data used for clustering is to be given by the required argument in the form of one or two lists of file names. The file list specified through the `b` switch defines the list of sufficient statistics files that are to be used for this clustering job. The directory where these files are expected to be found and which extension these files are expected to have is defined by the search paths/extensions set in the server. The file list given through the `c` switch specifies a list of feature vector files. For each of these feature vector files, a corresponding label file is expected to be found (once again, location and extensions of these files are defined by server parameters). The server will then compute sufficient statistics for the segments defined in the label file, from the corresponding segments in the feature vector file. If not all segments are to be used in clustering, a regular expression can be specified through the `e` switch in order to realize the desired filter.

The `d` switch is relevant if a large amount of data in the form of feature vector and label files is to be used. As the loading process in that case involves some temporary storage of feature vector and label files, it can become undesirable to issue one request to load all of the data to be used in the experiment. The granularity of the load requests can be controlled through the `d` switch. By default, the application will first issue a request to load the first 5 feature vector files. It then issues a request to load the first 5 corresponding label files. When these files are successfully loaded, the application requests the server to compute sufficient statistics for the loaded data after which the application will issue a request to release the storage used for the feature vectors and label files. The `d` switch allows the user to manipulate the granularity used in this loading process.

If a multi-server experiment is run, the user should pay attention to how the load-balancing parameters are set. Each server in the "server-network" has a load argument in the form of an integer. The load-balance between the servers in the network is proportional these integers. The master server (the server which directly connects to the application) will distribute files to be loaded in a round-robin fashion. The loading process becomes inefficient if the loading epoch size (set by the `d` switch) becomes smaller than the sum of the loads of all participating servers. If for example, the master server load is 2 and two connected slave servers have loads of 1 and 2 respectively, a loading epoch of less than 5 will avoid fully loading all the servers during the data loading process. If the `d` switch is set to 3, not all servers will be fully loaded all the time. The load-balancing will still work, but when the first loading request is issued, the second slave server will be unloaded as the 3 file request will be serviced by the master server and the first slave. The following 3 file load request will leave the first slave idle.

The desired task for the application is to be specified through the `k` switch. The switch takes a string as an argument. This string should consist of nothing but whitespace (spaces and tabs) separated words. The sequence of words specifies the sequential targets the clustering will aim for. Each word in this string should have the format `[dk][Ni|Lf]`. The first field specifies of the type of the next leg of the clustering task: a `d` specifies divisive clustering and a `k` specifies K-means style clustering. For a divisive task, if the next identifier is the letter `N` followed by a non-negative number `i` the divisive clustering continues until an inventory of `i` clusters is reached. If the next identifier is the letter `L` followed by a float, the divisive clustering will continue until the average distance per frame falls below this threshold. For a K-means style clustering task, the `N` option will specify the number of K-means iterations to be performed, the `L` option specifies an average distance threshold.

The `m` switch allows the user to control the iterations run after a cluster division. The program will run K-means cluster iterations for the data held in the original cluster that is split, using the two new cluster centers created by the split. It will continue for either `i` iterations or until the reduction of the average distance per frame per iteration falls under the threshold `f`. So by default, either 10 iterations or until the reduction of the average distance per frame pre iteration is less than 0.1.

An initial cluster inventory can be specified through the `f` switch. The name given to newly created units can be specified through the `g`, `h` and `i` switches. By default, the unit inventory is written in a subdirectory named `ASWU.N.M` where `N` denoted the number of units and `M` denoted the number of K-means iterations run. The new units will be named `NUUX` where `X` indicates an index starting from 1.

To avoid running into data sparsity problems, a minimum data threshold can be set through the n switch. In divisive clustering if a cluster contains fewer frames than the threshold, the cluster will be marked as un-splittable. If this situation occurs during K-means clustering, the cluster will be removed and all data contained in it will be partitioned over all remaining clusters. If during divisive clustering a cluster is encountered for which a non-invertible covariance is estimated, the splitted clusters are merged to create the original cluster again and this cluster is subsequently marked as un-splittable.

After completing a divisive clustering leg, the application requests the server to write a hierarchy file which describes the hierarchical relationship between the clusters. The location of the hierarchy files is determined by the output parameter of the server. The file written will by default be named **ASSM.X->Y** where X denotes the number of clusters at the start of the divisive clustering leg and Y denotes the number of clusters at completion. For details on the format of a cluster hierarchy file, please refer to the Appendix 付録 A .

2.5 Lattice decoding using the latrescore tool

The lattice decoder is an important part of the toolkit as it can be used for a number of tasks. First of all, it can be used to perform the "traditional" lattice decoding using an inventory of PSM models. Another important use for the lattice decoding tool is in re-segmentation and forced alignments. All the types of applications of the lattice decoder will be described in this section.

When the lattice decoder is executed without any arguments, it produces the following output:

```
USAGE: latrescore [options] [dict] mdltree files...

 Option                                     Default

 -a s     Serving host                      localhost
 -b       Do not write phone labels         write
 -c       Do not write word labels          write
 -d       Write word IDs rather than labels words
 -e       Input lattice is a phone lattice  wordlattice
 -f       Do a fast timeinitialization      exact
 -g i     Set neg. time direction win-size  5
 -h i     Set pos. time direction win-size  5
 -i s     Set the lattice format to s       SLF
 -k       Use hierarchy descendant models   do not
 -m f     Set the insertionpenalty to f     -10
 -v       Verbose                           off
 -S f     Set script file to f              none
 -X s     Set lattice directory to s        /
 -Y s     Set lattice extension to s        lat

FILES             | REQUIRED | LOCATED THROUGH
---------------------------------------------
Lattices          |  yes     |  X,Y switch
PSMs              |  yes     |  server
feature vectors   |  yes     |  server
```

(2.5.1) Lattice re-scoring

In a lattice re-scoring application, the lattice file that is to be given as an argument contains a word-level network. The lattice is assumed to have words associated with the arcs and optionally time instances associated with the nodes. If a node-labeled lattice is to be used, one should first convert the lattice to an arc-labeled lattice by using the `arcnodexchng` utility, described below.

Besides a lattice file, a pronunciation dictionary is required as an argument. The pronunciation dictionary has to be a binary format ASSM dictionary which can be generated from a number of dictionary formats using one of the text-preprocessing utilities and subsequently the dictionary parsing utility `dictpack`. A detailed description on how to generate a dictionary suitable for the use in a lattice decoding experiment will be given in the section 2.6.

To define the models that are to be loaded for the decoding run, a model-tree list is another required argument to the decoder. The model-tree file format is described in the Appendix 付録 A . It can be a hierarchical model tree as generated by the `km-las` tool. The location of the models, found in the model-tree file, is defined by the server parameters.

The lattice decoder will first parse the pronunciation dictionary. It will then parse the model-tree and request the server to load the required model inventory. It then starts the decoding for the lattice files given as arguments. If a large number of files are to be decoded, a list of lattices can be specified through the `S` switch. The decoder will perform the following steps for each of the lattices:

1. Parse lattice
2. Load feature vectors
3. Expand lattice with sub-lattices associated with nodes
4. Lookup and set index references into the dictionary for all words on all arcs
5. Expand arcs for which the associated word has multiple pronunciations into parallel arcs
6. Expand each compound word arc into a series of arcs
7. Expand each arc into a pronunciation network found in the dictionary for the word and pronunciation variation
8. Set references for each unit-level arc to the corresponding loaded model
9. Topologically sort the nodes in the lattice
10. Estimate time (or time windows) for each node in the lattice that does not have a time associated with it already
11. Extend the windows created by the previous step
12. DP search the lattice, using the constraint of the defined search space
13. Traceback and write result
14. Free storage used for the current lattice, feature vectors and results

To estimate times for the nodes that are unmarked (i.e. have no time associated with them) after loading the lattice file, the program recursively finds all paths between each pair of marked nodes. It estimates the times for the unmarked nodes by dividing the time difference between the marked nodes proportional to the the mean durations of the units along the path. If there are multiple paths passing through a node, the time estimation procedure will define a window for the node. If a large number of partial paths exist, this time initialization procedure can become computationally very expensive. To avoid spending a lot of resources on this, an approximation can be made by regarding both types of marked nodes (by means of estimation or by explicit marking from the lattice file) as equivalent. The time-initialization will become much less expensive in this way and no node will end up with more than one single time-instance associated with it (in contrast to a window). As this procedure is much less exact and restricts the search space considerably, a larger window parameter is probably appropriate in this case.

The majority of these tasks are performed by the application itself. The server is only responsible for keeping the model inventory and performing local DP. The application issues DP requests to the server by providing it with partial path scores from a source node and the window associated with the destination node. The server then execute the DP algorithm and returns the partial path scores and traceback information for the window associated with the destination node.

The times or windows associated with each node in the lattice after completing the time estimation procedure, are extended in positive and negative time directions by the values set through the `g` and `h` switches respectively. The marking procedure for the nodes that have no times associated with them is by default set to the exact method but can be set to the fast method by means of the `f` switch.

To compute the likelihood for a segment associated with an arc, the model associated with that arc is used. An alternative mode of operation can be achieved though by setting the `k` switch. If the `k` switch is set, the DP will not only consider the likelihood for the arc using the associated model but will also consider all other models that are within the same hierarchical group as the model associated with the arc.

(2.5.2) Re-segmentation

An alternative use for the lattice decoder `latrescore` is to re-segment an utterance using a model inventory. As a completely unrestricted re-segmentation (considering all possible segmentations) can be very expensive, the lattice decoder can be used to only search a part of the search space. The general procedure is that a lattice is created from an initial segmentation, allowing segment splits and/or merges. The lattice decoder is subsequently used to search for the new segmentation provided the search space restriction. In order to complete the first step in this process, a utility named `alab2lat` is provided. When this utility is run without any arguments, it produces the following output:

```
USAGE: alab2lat [options] labelfile latticefile

 Option                                 Default

  -c     Use label ID constraints         do not
  -m i   Maximum #segments in a merge     2
  -s i   Maximum #segments in a split     2
```

The initial segmentation files is to be provided as the first argument, the resulting lattice file name is to be provided as the second argument. The maximum number of splits or merges with respect to the initial segmentation can be controlled through the `s` and `m` switch respectively. By default, this utility will generate a lattice where each arc will be labeled as `*`. This indicates to the lattice decoder that no specific model is associated with the arcs and that all models are to be considered. If the `c` switch is set however, the utility will produces a lattice in which all arcs carry a label corresponding to the label found in the initial segmentation.

The second step in the algorithm now consists of simply running the lattice decoder to obtain the re-segmentation. The process is very similar to the one described in the previous section except that no word-to-unit expansion is required. One should therefore run `latrescore` in this case using the `e` switch which will indicate to the lattice decoder that the provided lattice is a unit-level rather than a word-level lattice.

(2.5.3) Forced alignment

Another use for the lattice decoder is to perform a forced alignment. The simplest case of a forced alignment can be achieved by use of the utility described in the previous section, setting the maximum number of segments in a split or a merge to 1 and forcing the utility to use the labels found in the initial segmentation. One might want to consider a forced alignment in which there is a certain probability that a unit can be reduced though. To offer this possibility, the utility `falign_latgen` is provided. When `falign_latgen` is ran without any arguments, the following output is produced:

```
USAGE: falign_latgen [options] labelfile latticefile

 Option                                   Default

  -w i  Maximum no emission sequence length  3
  -z s  Use empty emission probability file  do not
```

The maximum length of a sequence of units in the initial alignment that can have no observations associated with them can be specified through the **w** switch. A list with the probabilities for a particular unit to have no observations associated with it is to be given in a list file through the **z** option. The utility will build a lattice allowing reductions and places the probabilities of reductions (derived from the empty emission probability file) on the arcs.

2.6 Generating a dictionary file

In order to use a dictionary within the ASSM package, the user has to perform two steps. First a text processing tool has to be used to convert the dictionary file in a common text format. Then the dictionary parse utility `dictpack` can be used to create a binary version of the dictionary. This binary version of the dictionary is to be used with the tools that require a dictionary.

To provide access to a number of standard dictionary formats, 4 text-preprocessing utilities are provided. These text-preprocessing utilities provide access to dictionaries in the HTK, TIMIT, PRONLEX and BU formats. The utilities take a dictionary in their native format and convert it into the ASSM standard dictionary format. For details on this format, see the Appendix 付録 A and the format specification file in the `dictpreprocess` subdirectory of the ASSM package. To extend the package so as to provide access to other dictionary formats, it should be possible to generate new text-preprocessing tools.

Except for the preprocessing tool that converts BU format dictionaries, all provided utilities have only one switch. By default the text preprocessing tools for the HTK, PRONLEX and TIMIT format generate a dictionary in which multiple pronunciations are modeled as parallel linear pronunciation which only share a common start and end node. By setting the **s** switch, the tools will generate a dictionary with separate entries for the multiple pronunciations. One can use the `latbuild` tool which will be described in detail below to create pronunciation networks optimized for a minimum number of arcs and nodes from multiple pronunciations.

After an ASSM standard dictionary format file is generated, the final dictionary preprocessing step is to apply the dictionary parser `dictpack`. Executing this utility without any arguments generates the following output:

```
USAGE: dictpack [options] dictionary [wordlist]
 Option                                    Default

 -v      verbose                            off
 -M s    use 'mixture' spec. file s         none
 -O      binary lexicon output file name    <dict.>.bin
 -R s1 s2
         Map label s1 to s2                 none
```

The dictionary in the ASSM standard dictionary format is to be provided as the first argument to this utility. Optionally, one can provide a word-list as generated by the `wordlistbuild` tool described below. For a detailed description of the format of such a word-list file, please refer to the Appendix 付録 A . If such a word-list is provided, the resulting dictionary will only contain entries corresponding to the entries found in the word-list. The use of such a word-list provides a way to generate a dictionary containing multi-word lexical items from a dictionary that only contains single-word lexical items.

The `dictpack` utility provides two switches that allow the user to define label mappings. The `R` switch simply causes all occurrences of label `s1` to be replaced with label `s2`. The `M` switch allows the user to provide a more complex mapping. This switch allows the user to define a mapping that will cause all arcs with a particular label to be replaced by a number of parallel arcs. Besides defining the labels that these parallel arcs should carry, one can also define a an exponential weight factor for each of the parallel arcs. So if the exponential weight factor for a particular arc is denoted as α and the log likelihood score of the arc is denoted as β, the score used in decoding will be $\alpha\beta$. Details on the format of the file that is to be provided to the `M` switch is described in the Appendix 付録 A .

2.7 Efficient model re-estimation using the `mdlreest` tool

If a segmentation in which each segment is carrying a segment label exists, one might want to estimate ML model parameters from such a segmentation. One possible way to do this is by using the `km-las` tool. To estimate ML model parameters for model `M` from the segmentation, one could run a divisive clustering experiment with a numeric target of one cluster and use the `e` switch to filter the data so that only segments with label `M` are used. However, this is a very inefficient way to achieve this goal. As the `km-las` tool is designed to run clustering iterations, it will load all data in memory in the hope of fully utilizing the processor after completing the loading process. If only model re-estimation is desired, it is unnecessary to keep all data in memory as only incremental sufficient statistics are required (i.e. each new data item is only needed to update the sufficient statistics and will not be used after that). Another inefficiency one encounters using the `km-las` tool for model re-estimation is that for N models, the job would have to make $2N$ passes over all the data, each time filtering data for only one of the models. To avoid wasting resources on such a job, an application dedicated to the purpose of model re-estimation is provided. This application is named `mdlreest` and produces the following output if invoked without arguments:

```
USAGE: mdlreest [options] TaskFile filenames...

 Option                                   Default

 -a s     Set the serving host to s       localhost
 -b a     Set the model set name to s     REEST
 -v       Verbose                         off
 -S f     Set script file to f            none

FILES           | REQUIRED | LOCATED THROUGH
---------------------------------------------
feature vectors |  yes     |  server
label files     |  yes     |  server
```

A list of the models that are to be re-estimated together with the regular expression patterns that are to be used for the data filters for the models must be given in the `TaskFile`. For details on the format of such a task-file, please refer to the Appendix 付録 A or to the skeleton file provided in the package. The program will request the server to compile all the given regular expression patterns. It will then make one pass over the data applying all the compiled regular expression patterns and update incremental sufficient statistics for all the given models. After completing this pass, means are estimated for the models using the gathered sufficient statistics. The program then makes a second and final pass over the data, updating incremental sufficient statistics, and estimates covariances for all the models from the sufficient statistics after completion of this pass.

The polynomial order of the models, covariance type, number of regions, and location of where the models should be written are all set by the relevant server parameters.

2.8 Building of pronunciation networks using the `latbuild` tool

The `latbuild` tool allows the user to build a pronunciation network out of a list of linear pronunciations. Note that this tool can also be used to form a lattice out of an N-best list, as this is the same task. The program will take the first linear pronunciation and build a linear lattice out of it. It will then iteratively find the best alignment of the next pronunciations with respect to the current lattice using a DP match. The scores used in the DP match are -10 for a substitution, -7 for a deletion or an insertion and 0 for a match. After completing the DP alignment, the lattice is extended by adding arcs and nodes to the lattice at the points where substitutions, insertions and/or deletions occurred. After adding these arcs and nodes, the newly created lattice includes the last pronunciation.

When the lattice building program is executed without arguments, it generates the following output:

```
USAGE: latbuild [options] files...

 Option                                    Default

 -g s     Set the output lattice directory  current
 -h s     Set the output lattice extension  lat
 -v       Verbose                           off
 -X s     Set N-best list directory to s    current
 -Y s     Set N-best list extension to s    nbl

FILES           | REQUIRED | LOCATED THROUGH
---------------------------------------------
N-Best lists    |  yes     |  X,Y switches
Lattice files   |  no      |  g,h switches
```

The required arguments are the N-best lists of pronunciations in the standard N-best format. For details on the N-best file format, please refer to the Appendix 付録 A . the resulting pronunciation network is written in the directory set through the g switch using the extension set through the `h` switch.

2.9 Segment classification using the `segclass` tool

To conduct a classification experiment, the `segclass` tool is used. This tool will take a feature vector file and corresponding label file and will classify all selected segments as belonging to one of a collection of classes, defined by a model inventory.

If the `segclass` utility is executed without arguments, it produces the following output:

```
USAGE: segclass [options] modellist utterances...

 Option                                    Default
```

```
-a s    Serving host                      localhost
-b s    Use labels matching regexp s      *
-v      Verbose                           off
-S f    Set script file to f              none
-X s    Set label file dir to s           /
-Y s    Set label file ext. to s          lab
-Z s    Use label file MLF list s         none

FILES              | REQUIRED | LOCATED THROUGH
----------------------------------------------
PSMs               |   yes    |  server
label files        |   yes    |  X,Y,Z switches
feature vectors    |   yes    |  server
```

The model inventory to be used has to be given as the first argument, in the form of a list of model names. The feature vector files that contain the segments that are to be classified are given as the following arguments. The program will first request the server to load the model inventory. After successful completion of this, it iteratively requests the server to load the feature vector files together with the corresponding label files. By default it will classify all the segments defined in the label file but a segment filter can be used by defining a regular expression through the `b` switch.

2.10 Building a word-list using the `wordlistbuild` tool

The word-list building program takes as input a number of label files specifying where automatically derived unit boundaries are located. It also takes as input, label files specifying where word boundaries are located. An ascii file containing the lexical entries in the word-list as well as information on frequency and scores (as used in the word-list building algorithm) is produced as output.

The program first builds an initial word-list. This word-list will consist of a list of all unique words found in the word-level label files as well as a number of multi-word lexical items. The multi-word lexical items included in the list are all word sequences found in the word-level label files for which all internal word boundaries within the sequence "are crossed". A word boundary is considered to be crossed if the closest boundary between two automatically derived units is more than some threshold away.

After completing the initial word-list, the program computes a score for all multi-word lexical items based on the frequency and distance from the closest automatically derived unit boundary. It then iteratively prunes the multi-word lexical item with the lowest score by imposing a boundary constraint on the internal word-boundary with the lowest score. This iterative pruning will continue until the word-list has reached the desired size.

If the `wordlistbuild` program is executed without any command line parameters, the following output is produced:

```
USAGE: wordlistbuild [options] files...

 Option                                    Default

 -a i     Set the initial export interval  2000
 -b i     Set the final export interval    200
 -c i     Set the export threshold         5000
 -v       Verbose                          off
 -A s     Set auto-segment. labelfile dir  current
 -B s     Set auto-segment. labelfile ext  vit
 -C s     Set auto-segment. labelfile format HTK
 -D s     auto-seg. masterfile list s       none
 -F f     Set max time to fixed boundary     10 ms
 -H s     Set fixed boundary label file dir  current
 -J s     Set fixed boundary label file ext  fix
 -K s     Set fixed boundary masterfile to s none
 -M i     Set the multi-word Wordlist size   1000
 -O s     Set Wordlist output file to s      WrdLst.out
 -S f     Set script file to f               none
 -W s     Set word labelfile format          HTK
 -X s     Set word label file dir to s       current
 -Y s     Set word label file ext to s       wrd
 -Z s     Use word masterlabelfile list s    none

FILES             | REQUIRED | LOCATED THROUGH
----------------------------------------------
word labels       |  yes     |  X,Y,Z switches
auto-unit labels  |  yes     |  A,B,D switches
```

After the options, a list of label files is specified. These label files define the position of word boundaries. The program assumes that for each word-label file, a label file containing the positions of the boundaries between automatically derived labels in the corresponding utterance can be found in the directory specified through the `A` switch. The extension of the automatic-segmentation label file is specified through the `B` switch.

The threshold for considering a word boundary to be crossed by an automatic segment can be specified through the `F` switch.

The desired resulting word-list size (including both single and multi-word entries) is specified through the `M` switch. During the execution of the pruning algorithm, the program will write out the current word-lists each time a multiple of the export interval specified by the `a` switch is reached. Once the inventory size falls under the threshold set by the `c` switch, the program will write out the current word-list every time the inventory size is a multiple of the export interval set by the `b` switch.

The boundary constraints, necessary to prune the word-list down to the desired size are written in the directory specified through the `H` switch. The extension of the boundary constraints files can be set through the `J` switch.

2.11 Deriving pronunciation statistics using the pronstats tool

The `pronstats` tool can be used to gather label-dependent alignment statistics for two given segmentations. The program will iteratively go through the labels in segmentation A. For label X from segmentation A it will align all the labels in segmentation B for which the midpoint of the label is before the end of X.

When the `pronstats` tool is executed without arguments it produces the following output:

```
USAGE: pronstats [options] files...

 Option                                   Default

 -v s     Write average sequence lens to s  don't write
 -w s     Write mapping stat files to dir s don't write
 -x s     Set mapping stat file ext to s    prf
```

```
-A s     Set auto-unit labelfile dir        current
-B s     Set auto-unit labelfile ext        vit
-C s     Set list of auto-unit MLF's to s   none
-F f     Lexical label offset               12.8 ms
-G s     Set lexical label dir to s         current
-H s     Set lexical label ext to s         lab
-I s     Set list of lexical MLF's to s     none
-S s     Set script file to s               none
-W s     Set lexical labelfile format       HTK
-X s     Set auto-unit labelfile format     HTK

FILES            | REQUIRED | LOCATED THROUGH
---------------------------------------------
lexical labels   |  yes     |  G,H,I switches
auto-unit labels |  yes     |  A,B,C switches
```

After completing the alignment the program will by default only print label inventories found and statistics on frequency of sequence-lengths. If a directory is specified through the w switch though, the program will write full statistics on the alignment in binary files in the specified directory. For details on the file format of these statistics files please refer to the Appendix 付録 A .

2.12 Manipulating pronunciation statistics using the prf2pdict tool

As an unsupervised alignment of an automatic unit segmentation and a phonetic segmentation possibly generates a very large inventory of observed unit sequences per phone, it can become desirable to perform a forced alignment from the phone level transcriptions using only the most likely pronunciations of phones in terms of automatic units. This tool can be used to generate a pronunciation dictionary from the pronunciation statistics generated by the pronstats tool. The tool thresholds and re-normalizes the pronunciation statistics files and writes a multiple pronunciation dictionary where the pronunciation variants have probabilities derived from the relative frequencies of the sequences after re-normalization. A separate file is generated that defines the probability that a phone has no observations associated with it.

When the prf2pdict tool is executed without any arguments it produces the following output:

```
USAGE: prf2pdict [options] PrfDat_files...

 Option                                      Default

 -a i     Minimum number of pronunciations     2
 -b i     Maximum number of pronunciations    10
 -c f     Probability cutoff                  0.001
 -d s     Pronunciation dictionary file       Pdict
 -e s     Zero emission probabilities file    ZProbs
 -S s     Use scriptfile s                    none
```

The pronunciation statistics files that are to be used to generate the pronunciation dictionary are the required arguments. The probability threshold can be defined through the c switch. Another constraint can be applied using the a and b switches which define the minimum and maximum number of pronunciations that can be generated from each pronunciation statistics file. The list of probabilities of zero emissions is written to the file specified through the e switch. This list of zero emissions can be used as input to the forced alignment lattice generation utility falign_latgen described in (2.5.3).

2.13 Continuous speech recognition using the fmatch tool

The fmatch tool is an initial implementation of a decoder. The features the decoder operates on are the automatically derived units. The only pronunciation model available at this time is in the form of probabilities estimated from the relative frequencies produced by the pronstats program. The decoder first builds a decoding network from a back-off bigram language model.

The network will have the topology shown in figure 1. It then expands the word arcs with the pronunciation networks it finds in the pronunciation dictionary. After completing this step the actual DP decoding starts. In pseudo-code, the decoder performs the following steps:

```
Initialize network;
Iterate over automatic labels
{
   Update DP for emitting phones;
   Beam prune partial paths;
   Propagate scores through the LM arcs;

   Update DP for non-emitting phones;
   Beam prune partial paths;
   Propagate scores through the LM arcs;

   Update emitting phone tracebacks;
   Update tracebacks pointers of LM arcs;

   Update non-emitting phone tracebacks;
   Update tracebacks pointers of LM arcs;

   Create list of alternate tracebacks for emitting phones.
   Create list of alternate tracebacks for non-emitting phones.

   Rotate circular buffers;
}
```

An active phone record is connected to each phone arc in the network that has a valid partial path score within a time window from the current time backwards. The size of this time window is defined by the maximum sequence length of any mapping to a single phone. During the DP extension of partial paths, new active phone records can emerge if the previous phone in the network has a partial path score within the maximum duration window. Each active phone record holds partial path information within the maximum duration window for partial paths ending in either an emitting phone or a non-emitting phone.

The DP update steps in the algorithm extend partial paths up to the current time. The beam pruning step will mark partial paths as invalid in the case that the partial path score falls more than a threshold below the maximum score of a partial path at the current time. Separate beams for partial paths ending in an emission and ending in an empty emission are applied. The empty emission DP updates are to be executed after the network is updated up to the current time as the empty emission rely on partial path information being updated up to the current time. Note that only empty emission partial path extensions that follow an emitting partial path are considered, meaning that no two subsequent phones can have empty emissions.

After completing the DP update of the partial paths and applying the beam pruning, the traceback information is updated. Each partial path has a pointer into a traceback lattice that reveals the path history

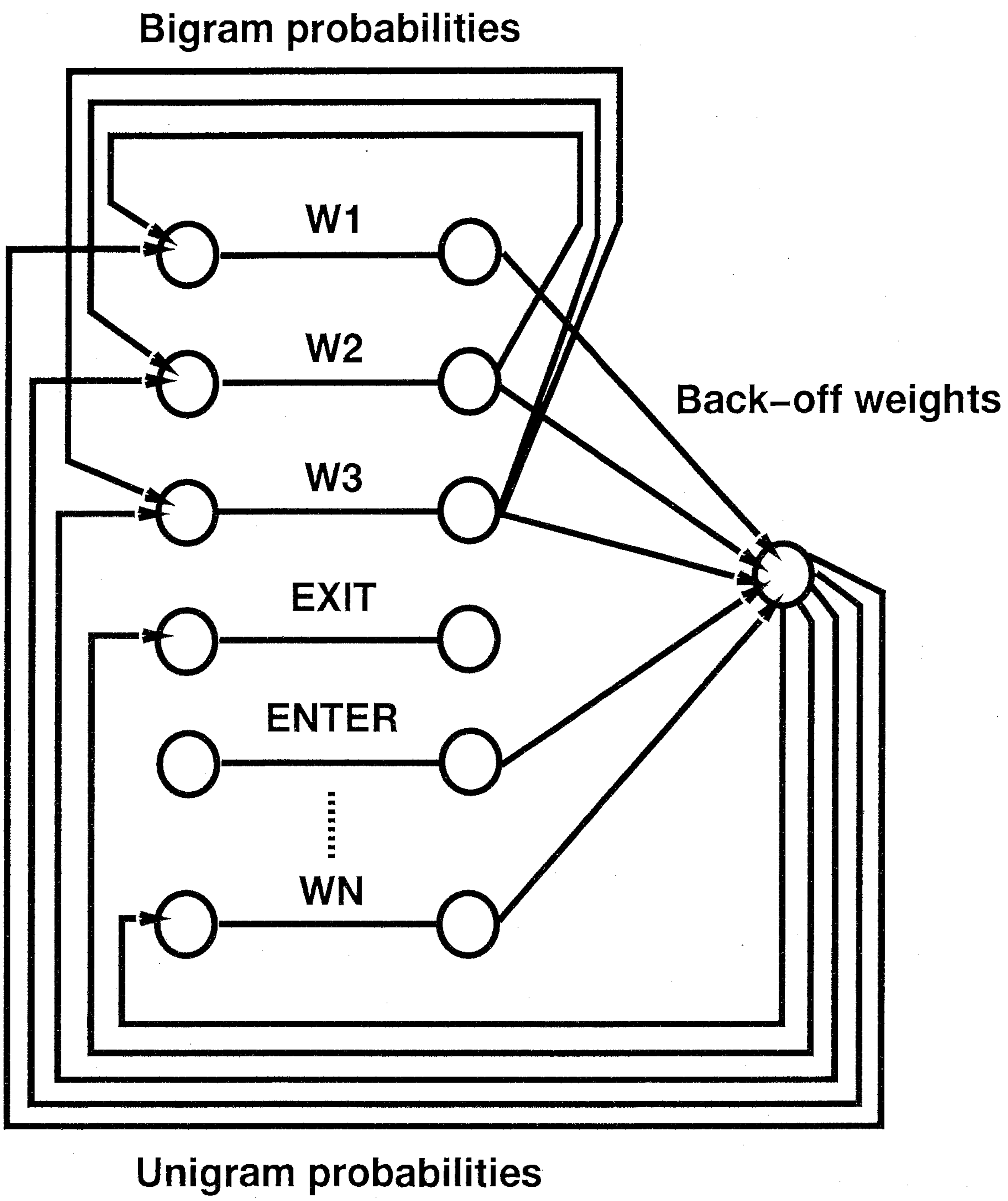

図 1: Word-level recognition network

that lead to the partial path score. Each item in the traceback holds besides the necessary path information (word/phone, score and time), a pointer indicating what the next record along the path is. Each record also holds a usage counter which registers how many records are pointing to it. If a phone-level alignment is desired, the traceback update will "push" a new traceback record onto the traceback lattice structure for each active phone record's valid partial path up to the current time. If only a word-level alignment is required, only the word initial phone arcs will push a new record onto the lattice. All word-internal phones will in this case only set a reference to the appropriate traceback record.

After the tracebacks are updated, a list is formed with alternative word-tracebacks for the current time. All partial paths up to the current time for all word ends are considered and are sorted in descending score order. The tracebacks for the partial paths within a beam from the maximum partial path score at any word-end are registered in a list. In fact, two lists are made, one with candidate word-ends ending in an emission and one list with candidate word-ends ending in an empty emission.

After completing the lists of alternatives, the circular buffers of all the active phone records are rotated to indicate the passing of time. The partial path storage for the time instance that will "go out of scope" (is becoming so old that it is not within the maximum duration window anymore) is now made available for the partial path information of the current time. Before making this partial path storage available, the traceback that is going out of scope is checked. If the traceback record indicates that no following traceback record is using it to store the path history and the traceback record is not a member of the word-end alternate traceback list (the usage counter is zero), then the traceback records are released for reuse.

After the DP search loop is completed, a recursive traceback from the exit word, using the alternate tracebacks lists for each time instance found along the the recursive traceback results in a lattice of hypotheses.

A caveat of generating a lattice in this manner is that within the word, the max-operator will allow only the best scoring path to stay alive. If the second best path is propagating through the same word as the best scoring path (the second best path only differs in the traceback history compared to the best scoring path), it dies because of the nature of the max-operator. To avoid search errors of this kind, a set of active records instead of a single record can be used for each phone in the network. Within the words, the partial paths held in the different level phone records are propagated within their level to the end of the word.

When the decoder `fmatch` is executed without arguments, it produces the following output:

```
USAGE: fmatch [options] PrfDatList Dict. ARPA_MIT-LL Labfiles...

 Option                                       Default

 -a       Write phone level alignment          off
 -b f     Set the beamwidth to f               20
 -c f     Set word-alternates beam width to f  3
 -d i     Limit length of mapped sequences     unlimited
 -f s     Set the wordlevel score format to s  AL
 -l f     Set the language model weight to f   0.1
 -m i     Set max # of DP records per phone    1
 -p f     Set the word insertion penalty to f  10
 -v       Verbose                              off
 -w s     Set wordlevel output directory       current
 -x s     Set wordlevel output extension       wrd
 -y s     Set phonelevel output directory      current
 -z s     Set phonelevel output extension      phn
 -L s     Set lattice output to directory s    1-Best
 -M s     Set lattice output file extension    lat
 -S s     Use scriptfile s                     none
```

The a switch sets the type of alignment output that is desired. The beam-width used in the beam-pruning can be set through the b switch. The beam width used to determine alternate word-tracebacks can be set through the c switch. The d switch allows the pronunciation statistics that are used to estimate the observation probabilities of sequences given a phone to be truncated. Any sequence larger than the set maximum will have a zero probability to be observed. The probability of observing other sequences will be estimated from the remaining statistics, normalizing appropriately for the obervations up to the set maximum sequence length. The directory where resulting lattice files are to be written is set through the L switch and the lattice file extensions are set through the M switch. If the L switch is not used, only the 1-best answer is written. Directories and extensions of the phone and word level alignments can be manipulated through the w, x, y and z switch. The scores for the labels in the word-level 1-best output can be formated using the f switch. If the switch is set to A the acoustic word score is written, if the switch is set to L the language model (and word-insertion penalty) is written for each word label. If both are set, the addition of the two is written as a score.

The number of phone records that are associated with an active phone arc in the network can be controlled through the m switch.

2.14 Diagnostic tools

The remaining tools in the ASSM package are for the purpose of diagnostics and manipulation of the results of the described tools.

(2.14.1) The binary file viewer `lmdl`

In order to check the contents of a binary file, the `lmdl` utility can be used. As each binary file contains a magic cookie, the viewer will automagically figure out what type of binary file you want to look at. The `lmdl` utility can be used to view sufficient statistics files, PSM files, dictionary files, pronunciation statistics files and master files. For details on the format of these files, refer to the Appendix 付録 A . Invocation is achieved simply by giving the files to view as command-line arguments. The contents of the binary files will be written in ascii form to the standard output.

(2.14.2) The feature vector file viewer `cepview`

Feature vector files can be visualized using the `cepview` utility. The utility requires a feature vector file to be given as an argument and optionally allows a label file to be given as a second argument. The feature vectors will be displayed in a grey-scale plot. If a label file is specified, the label boundaries are drawn in the form of red lines in between the feature vectors.

(2.14.3) Master file manipulation using the `mascntrl` utility

The ASSM package supports master files which can contain sufficient statistics files and/or PSM files. The `mascntrl` utility allows the user to view and manipulate the contents of such master files. The functionality of the program is very similar to the UNIX `tar` utility. Executing `mascntrl` without arguments results in the following output:

```
USAGE: mascntrl x|c|t|r|A [options] MasModelFile [files]...

 Option                                       Default

 -F      Extended verbose                     Off
 -O s    Output x-tracted files to dir s      current
 -S f    Set scriptfile to f                  none
```

The first argument is required and defines the type of operation that is to be taken on the master file. The master file to operate on is to be given as the next argument. Extraction is achieved by the `x` operation, construction by the `c` operator. The `t` operator lists the contents of a master file. Finally the `r` operator appends a new file to the master file and the `A` operator concatenates another master file to the current master-file.

(2.14.4) Lattice coverage analysis using the `latmatch` utility

The `latmatch` utility is provided to allow the user to determine how well a labeled segmentation is covered by the hypotheses comprised in a lattice file. When the `latmatch` utility is invoked without arguments, it produces the following output:

```
USAGE: latmatch [options] [dict] files...

 Option                                     Default

 -e       Input lattice is a phone lattice  wordlattice
 -f       Do a unit level match             wordlevel
 -g s     Set the input label directory     current
 -h s     Set the input label extension     lab
 -i s     Set list of input label MLFs      none
 -j s     Set the output directory          current
 -k s     Set the output extension          lmo
 -m       Find the worst match              best
 -n       Use wordIDs rather than words     words
 -v       Verbose                           off
 -X s     Set lattice directory to s        current
 -Y s     Set lattice extension to s        lat

FILES            | REQUIRED | LOCATED THROUGH
---------------------------------------------
Lattices         |  yes     |  X,Y switches
Label files      |  yes     |  g,h,i switches
```

The files to align have to be given as the required argument(s). The utility then performs a DP alignment of the transcription with respect to the lattice. The scores used in this DP search are equal to those used in the `latbuild` tool. The best alignment is written to a label file in the directory set by the `j` switch and with an extension that can be set through the `k` switch. If the verbose switch is turned on, the program reports on the alignment in terms of substitutions, insertions and deletions. By default, the lattice as well as the transcription is assumed to be on the word-level. If this is not the case, the `e` end `f` switches are to be used. If the lattice is a word-level lattice and the alignment is at the phone-level, a pronunciation dictionary is required and should be specified as the first argument to the program. One can obtain the worst match by setting the `m` switch.

(2.14.5) Lattice type conversion using the `arcnodexchng` utility

Even if there would be only one lattice file format in use, there still remains a significant confusion in how to relate the hypotheses information to the symbols in the lattice. One view is to associate words with the nodes in a network and view the arcs in the lattice as symbols of the word boundaries. This view will be denoted as a node-labeled lattice. Another view is to associate the words with the arcs in the lattice and associate the word boundaries with the nodes in the lattice. Such a lattice will be referred to as an arc-labeled lattice. Within the ASSM package lattices are assumed to be arc-labeled. If one has obtained a node-labeled lattice or one wants to use a lattice resulting from an ASSM tool in another package that uses node labeled lattices rather than arc-labeled lattices, the **arcnodexchng** utility provides a way to convert a lattice from one type to the other.

When the **arcnodexchng** utility is executed without any arguments, it produces the following output:

```
USAGE: arcnodexchng InputLattice OutputLattice

 Option                                    Default

 -a s     Set label for start dummy node   WORDSTART
 -b s     Set label for end dummy node     WORDEND
 -v       Verbose                          off
 -L s     Set lattice file format to s     SLF
```

The source lattice file is to be given as the first argument, the converted lattice will be written to the file given as the second argument.

(2.14.6) The likelihood computation utility `lhcomp`

The `lhcomp` utility allows the user to compute the cumulative likelihood of a segmentation. The program takes the label files that describe the segmentation as arguments. It then sums the score fields found for each of the labels in the label files and sums them. After completion it reports the resulting sum.

(2.14.7) Sufficient statistics analysis using the `ss2cep` tool

The `ss2cep` utility allows the user to generate a feature vector file from a sufficient statistics file. Using the mean trajectory parameter of a segment and it's length, it computes the mean trajectory sequence of frames for each segment and writes the resulting feature vector file.

(2.14.8) PSM analysis using the `psm2cep` tool

The `psm2cep` utility performs a task very similar to the `ss2cep` utility. It takes a model list and a label file as argument. It first attempts to retrieve the PSM inventory and subsequently generates the mean trajectories for the segments found in the label file using the mean trajectory parameters of the model corresponding to the label identities of the labels.

3 Using the ASSM server for new tools

There are a number of advantages in using a client server model of the software. It allows the coding of applications to be more straightforward as the results from the service offered by the server are available after a simple data exchange between the application and the server. The application is relieved from the burden of doing data and memory management while it still maintains the control and functionality. Although this advantage is also offered by using a well designed library, the client server model offers a number of additional advantages, not offered by the library. As the client server connection can be established through a network connection, it becomes easier to make the computation required for an application distributive. Yet another great advantage of the client server model is that the application can be written in a completely different programming language. Although many packages provide interfaces of their functionality without the constraint of the programming language it was written in, but they usually do involve special libraries and frequently requires the application programmer to spend quite some time getting acquainted with the interface. The client server model on the other hand truly simplifies the interface to the services offered by the server, regardless of the programming language of choice. The only requirement is that the language is capable of using the underlying system calls so as to provide access to the networking layers of the operating system. The client server model avoids the need for any material such as libraries to be provided by the ASSM package. The only thing required is a detailed description on how to call the services provided by the server and a list of the services available.

The next section will describe the format in which communication is to take place between client and server. The following section gives a detailed description of the available services provided by the server.

3.1 Communication protocol

The communication protocol can be divided into two sections. First of all there is the protocol for establishing a connection between the client and the server. Secondly there is the protocol used for exchange of requests and results.

(3.1.1) Connection establishment protocol

After the server is started it listens for incoming TCP/IP connections on a port defined at compile time (see the Appendix 付録 B for details). It also listens for incoming UNIX Domain Socket (UDS) connections through the listen socket created in the file-system. The location of this UDS is determined at compile time (refer to the Appendix 付録 B for details). For the client to establish a connection, it should create a socket and connect it to the listen socket of the server. It then has to execute the connection initialization protocol that involves the following steps:

1. send the size of an integer on the client's machine in the form of a single byte.
2. send the size of a short on the client's machine in the form of a single byte.
3. send the size of a long on the client's machine in the form of a single byte.
4. send the size of a float on the client's machine in the form of a single byte.
5. send the size of a double on the client's machine in the form of a single byte.
6. send the number 1 as a short in the client-machine's native byte order to establish possible byte order differences.
7. Send user id and process id as a regular request.

The basic type sizes are exchanged first to ensure that the machines running the server and application are compatible. If there is a difference in basic type sizes, the server immediately closes the connection. If the machines are compatible, differences in byte order are detected. If the byte order of the server's machine and application machine differ, the server will do the necessary byte swapping for all following communication between client and server. After the establishment of these basic properties the "normal" mode of communication can be entered. In this mode of communication, requests to the server and replies from the server are formated in a well defined manner allowing a variable number of arguments as well as arguments of different types. The details of this format will be described in the following section.

If the programming language of choice is C, a full implementation of the protocol is provided in the `Serv` library. A structure named `Connection` is defined that holds all the information relevant to a connection as well as the necessary buffers required for communication through the connection. The `Connection` type definition is as follows:

```
typedef struct{
  uid_t              UserId;
  pid_t              ProcId;
  unsigned long int  UserAddr;
  int                NeedSwaps;
  int                 SockFd;
  int                 ReqId;
  void               *Arguments;
  int                 ArgBufSize;
  int                 ArgDataSize;
  int                 NumArgs;
  char               *ArgClasses;
  int                 ArgClassBufSize;
}Connection;
```

The first two fields hold the user and process id of the connected process, the third field holds the IP address of the machine on which the connected process is running in case a TCP/IP connection is made. The `NeedSwaps` field is irrelevant to the application and is only used within the server. The `SockFd` is the file descriptor of the socket connection. All remaining fields are used to provide the "normal" mode of data exchange which will be described in detail in the next section. To establish a connection to the server, all the application has to do is call the function:

```
Connection     *GetServerConnection(char *ServerHost)
```

If the provided `ServerHost` argument is `NULL`, a UDS connection is attempted to be established. If the provided argument is not `NULL` a TCP/IP connection is attempted to be established to the host named `ServerHost`. If the connection can be established and initiated successfully, the function will return a pointer to a newly allocated `Connection` structure that holds the information relevant to the connection. If something fails, the program execution is terminated with a error message.

(3.1.2) Data exchange protocol

After the sizes of elementary types are exchanged and any byte order differences between the communicating machines are detected, all data is exchanged using the following well defined protocol.

- Send a request/result identifying integer.
- Send the number of arguments to the request/result.
- Send an argument format string.
- Send the actual arguments.

The format string specifies what type of arguments are going to follow and implicitly defines the amount of argument data that is going to follow. For each argument, there are is a 4 byte type qualifier in the format string. So if the transfer has 4 arguments, the format string is 16 bytes long. The 4 byte qualifier starts with a one character type qualifier and is optionally followed by a 3 byte human-readable string that denotes the argument's length. The character qualifiers are listed in table 1.

For example, a request identified by the integer 4 that takes an integer and a 35 byte string as arguments would result in transmission of the following data stream:

1. The number 4 as an integer (request id)
2. The number 2 as an integer (2 arguments will follow)
3. The string `i   t 35` (format string)
4. An integer value (argument 1)
5. A 35 byte string (argument 2)

Qualifier	Argument type	Length
i	integer	not used
s	short	not used
u	unsigned short	not used
l	long	not used
m	unsigned long	not used
f	float	not used
t	string	number of characters (string length)
v	vector	number of float (vector size)

表 1: Format string qualifier list

As for the connection establishment phase, using the C programming language considerably simplifies the communication as a large number of functions are provided in the `Serv` library.

The arguments and request id of the request that is to be sent can be held in the `Connection` structure. To append a new argument to the arguments already held in the `Connection` structure, one can use the function:

```
void PushArgument(char ArgType, int ArgSize, void *ArgVal,
                  Connection *ServerConn)
```

The first argument to this function is the one character qualifier corresponding to the new argument (see tabel 1). The second argument to the function specifies the size of the argument. A generic pointer to the memory location that holds the actual argument value is to be provided as the third argument. The connection structure to append the new argument to is to be given as the fourth argument.

To alter an argument already held in the `Connection` structure, the function,

```
int AlterArgument(char ArgType, int ArgSize, void *ArgVal,
                  int ArgIdx, Connection *ServerConn);
```

can be used. The first 3 arguments of this function serve the same purpose as the first 3 arguments to the `PushArgument` function. The fourth argument indicates the index of the argument that is to be altered. If the index refers to a non-existing argument or if the type of the indexed argument is different from the newly provided argument the function will return -1 indicating the error. On success the function returns 0.

To retrieve values held in a `Connection` structure, the following functions are provided:

```
int            *PopIntArg(Connection *ServerConn, int ArgIdx)
long           *PopLongArg(Connection *ServerConn, int ArgIdx)
short          *PopShortArg(Connection *ServerConn, int ArgIdx)
unsigned short *PopUnsignedShortArg(Connection *ServerConn, int ArgIdx)
unsigned long  *PopUnsignedLongArg(Connection *ServerConn, int ArgIdx)
float          *PopFloatArg(Connection *ServerConn, int ArgIdx)
char           *PopStringArg(Connection *ServerConn, int ArgIdx)
```

These functions return on success a pointer to the memory location that holds the desired argument and return a `NULL` pointer if an error occurred. The expected type of the argument to be retrieved is indicated by the chosen function call. For example, if the expected argument type is an integer, the `PopIntArg` is used. The `Connection` structure to extract an argument from is to be given as the first argument to these functions. The index of the argument that is extracted from the `Connection` structure is to be given as the second argument. An error can occur if the argument index is out of range or because the indexed argument is not of the expected type.

Although a vector is simply an ordered collection of floats, special argument manipulation functions are provided for this type, as these functions are more efficient than repetitive calls to the float argument manipulation functions. The

```
void PushVectorArgument(Vector Frm, Connection *ServerConn)
```

provides a way to add a vector argument to a `Connection` structure. The function

```
int PopVectorArg(Connection *ServerConn, int ArgIdx, Vector Data)
```

provides a way to retrieve a vector argument. The retrieval function will return 0 on success, -1 if an error occurred.

Similar to the vector type, a matrix is nothing more than an ordered collection of vectors but for efficiency sake a function dedicated to the retrieval of a matrix is provided. Adding a matrix as an argument to a `Connection` structure is efficient as the `Connection` structure has a reference to the end of the argument data block at all times. Retrieving a matrix from a `Connection` structure can become very inefficient though as for each retrieved vector, the retrieval function has to compute the offset of the vector in the argument data block from the argument format string member of the `Connection` structure. As a matrix retrieval will consist of retrieving a series of directly following vector arguments, a more efficient implementation exists. A matrix can be retrieved from a `Connection` structure by means of the function

```
int PopMatrixArg(Connection *ServerConn, int SArgIdx, Matrix Data).
```

The first vector argument of the matrix is expected to be the `SArgIdx`-th argument. The function returns the retrieved matrix in the `Data` argument and returns 0. On error the contents of the `Data` argument is undefined and the function returns -1.

When arguments to a `Connection` structure are provided, the arguments and request idea can be sent to the connected process by calling the function

```
void SendRequest(Connection *ServerConn).
```

To receive data from the connected process, one can call the function

```
void ReceiveRequest(Connection *ServerConn).
```

The receiving function will block until all arguments are received from the connected process. If the connected process is not ready to write the data yet, the operating system will put the blocking process to sleep until data becomes available. As a sleeping process does not utilize any CPU, this is an efficient way to have a process wait until further processing is possible.

The memory management of the buffers necessary to hold the arguments of a `Connection` structure are transparent to the user (i.e. more memory will be allocated by the argument manipulation functions as it becomes necessary). To reset the number of arguments held in the `Connection` structure, one only has to set the `ArgDataSize` and `NumArgs` fields of the structure to 0.

3.2 Services provided by the ASSM server

Now that it has become clear how to interact with the server, the available services provided by the server will be described. At this moment, the ASSM server provides 87 different services which will be described in this section. For C programmers, the header file `ASSMServProt.h` provides defines for the different request ids of the server. A shorthand reminder of the available requests and the required arguments is provided in the `ServerProtocol.txt` file in the package. See Appendix 付録 B for details on the location. The server will return a negative request id, with the same absolute value as the request id on success. On error, the server will return an error message as the only argument in the `Connection` structure.

The following list gives a short explanation of the available services. The string given between braces after the request id are the names of the defines provided in the C header file. The implications of issuing a request in a multi-server environment are given for each of the requests. A function is transparent to the user if issuing the request will automatically trigger the same request to be issued to all connected slave servers (including id translations). If a request is also server distributive, the computation/storage resulting from the request is distributed over the servers.

Request	: 1 (LISTFVPATHS)
Transparent	: No
Number of arguments	: none
Returns	: List of feature vector search paths (string)

Returns a list of search paths used for feature vector file retrieval. An empty lists indicates that only the root directory is searched.

Request	: 2 (ADDFVPATH)
Transparent	: No
Number of arguments	: 1
Argument 1	: FV1 (string)
Returns	: nothing

Adds FV1 to the list of feature vector search paths.

Request	: 3 (REMFVPATH)
Transparent	: No
Number of arguments	: 1
Argument 1	: i
Returns	: nothing

Removes the feature vector search path with index i from the list of feature vector search paths.

Request	: 4 (LISTFVEXTS)
Transparent	: No
Number of arguments	: none
Returns	: List of feature vector search extensions (string)

Returns a list of search extensions used for feature vector file retrieval.

Request	: 5 (ADDFVEXT)
Transparent	: No
Number of arguments	: 1
Argument 1	: FVE1 (string)
Returns	: nothing

Adds FVE1 to the list of feature vector search extensions.

Request	: 6 (REMFVEXT)
Transparent	: No
Number of arguments	: 1
Argument 1	: i (integer)
Returns	: nothing

Removes the feature vector search extension with index i from the list of feature vector search paths.

Request	: 7 (LISTLBPATHS)
Transparent	: No
Number of arguments	: none
Returns	: List of label search paths (string)

Returns a list of search paths used for label file retrieval. An empty lists indicates that only the root directory is searched.

```
Request             : 8 (ADDLBPATH)
Transparent         : No
Number of arguments : 1
Argument 1          : LB1 (string)
Returns             : nothing
```

Adds LB1 to the list of label search paths.

```
Request             : 9 (REMLBPATH)
Transparent         : No
Number of arguments : 1
Argument 1          : i
Returns             : nothing
```

Removes the label search path with index i from the list of label search paths.

```
Request             : 10 (LISTLBEXTS)
Transparent         : No
Number of arguments : none
Returns             : List of label search extensions (string)
```

Returns a list of search extensions used for label file retrieval.

```
Request             : 11 (ADDLBEXT)
Transparent         : No
Number of arguments : 1
Argument 1          : LBE1 (string)
Returns             : nothing
```

Adds LBE1 to the list of label search extensions.

```
Request             : 12 (REMLBEXT)
Transparent         : No
Number of arguments : 1
Argument 1          : i (integer)
Returns             : nothing
```

Removes the label search extension with index i from the list of label search paths.

```
Request             : 13 (LBMLFLIST)
Transparent         : No
Number of arguments : none
Returns             : List of MLF's (string)
```

Returns a list of the label MLF's used for label file retrieval.

```
Request             : 14 (LBMLFADD)
Transparent         : No
Number of arguments : 1
Argument 1          : MLF1 (string)
Returns             : nothing
```

Attempts to open MLF1 and adds this MLF to the list of MLF's used for label file retrieval if successful.

Request	: 15 (LISTSSPATHS)
Transparent	: No
Number of arguments	: none
Returns	: List of sufficient statistics search paths (string)

Returns a list of search paths used for sufficient statistics file retrieval. An empty lists indicates that only the root directory is searched.

Request	: 16 (ADDSSPATH)
Transparent	: No
Number of arguments	: 1
Argument 1	: SS1 (string)
Returns	: nothing

Adds `SS1` to the list of sufficient statistics search paths.

Request	: 17 (REMSSPATH)
Transparent	: No
Number of arguments	: 1
Argument 1	: i
Returns	: nothing

Removes the sufficient statistics search path with index `i` from the list of sufficient statistics search paths.

Request	: 18 (LISTSSEXTS)
Transparent	: No
Number of arguments	: none
Returns	: List of sufficient statistics search extensions (string)

Returns a list of search extensions used for sufficient statistics file retrieval.

Request	: 19 (ADDSSEXT)
Transparent	: No
Number of arguments	: 1
Argument 1	: SSE1 (string)
Returns	: nothing

Adds `SSE1` to the list of sufficient statistics search extensions.

Request	: 20 (REMSSEXT)
Transparent	: No
Number of arguments	: 1
Argument 1	: i (integer)
Returns	: nothing

Removes the sufficient statistics search extension with index `i` from the list of sufficient statistics search paths.

Request	: 21 (LISTSSMMF)
Transparent	: No
Number of arguments	: none
Returns	: Sufficient statistics MMFs (string)

Returns a list of the sufficient statistics MMF's used for sufficient statistics retrieval.

`Request`	`: 22 (ADDSSMMF)`
`Transparent`	`: No`
`Number of arguments`	`: 1`
`Argument 1`	`: MMF1 (string)`
`Returns`	`: nothing`

Adds the MMF1 to the list of MMF's used for sufficient statistics file retrieval.

`Request`	`: 23 (REMSSMMF)`
`Transparent`	`: No`
`Number of arguments`	`: 1`
`Argument 1`	`: i (integer)`
`Returns`	`: nothing`

Removes the MMF indexed `i` from the list of sufficient statistics MMF's used for file retrieval.

`Request`	`: 24 (SPECSSMMF)`
`Transparent`	`: No`
`Number of arguments`	`: 1`
`Argument 1`	`: i (integer)`
`Returns`	`: List of sufficient statistics file names (string)`

Returns a list of filenames of the sufficient statistics files held in the MMF indexed `i`.

`Request`	`: 25 (LISTMDLPATHS)`
`Transparent`	`: No`
`Number of arguments`	`: none`
`Returns`	`: List of PSM search paths (string)`

Returns a list of search paths used for PSM file retrieval. An empty lists indicates that only the root directory is searched.

`Request`	`: 26 (ADDMDLPATH)`
`Transparent`	`: No`
`Number of arguments`	`: 1`
`Argument 1`	`: MDL1 (string)`
`Returns`	`: nothing`

Adds `MDL1` to the list of PSM search paths.

`Request`	`: 27 (REMMDLPATH)`
`Transparent`	`: No`
`Number of arguments`	`: 1`
`Argument 1`	`: i`
`Returns`	`: nothing`

Removes the PSM search path with index `i` from the list of PSM search paths.

`Request`	`: 28 (LISTMDLEXTS)`
`Transparent`	`: No`
`Number of arguments`	`: none`
`Returns`	`: List of PSM search extensions (string)`

Returns a list of search extensions used for PSM file retrieval.

`Request`	: `29 (ADDMDLEXT)`
`Transparent`	: `No`
`Number of arguments`	: `1`
`Argument 1`	: `MDLE1 (string)`
`Returns`	: `nothing`

Adds `MDLE1` to the list of PSM search extensions.

`Request`	: `30 (REMMDLEXT)`
`Transparent`	: `No`
`Number of arguments`	: `1`
`Argument 1`	: `i (integer)`
`Returns`	: `nothing`

Removes the PSM search extension with index `i` from the list of PSM search paths.

`Request`	: `31 (LISTMDLMMF)`
`Transparent`	: `No`
`Number of arguments`	: `none`
`Returns`	: `PSM MMFs (string)`

Returns a list of the PSM MMF's used for sufficient statistics retrieval.

`Request`	: `32 (ADDMDLMMF)`
`Transparent`	: `No`
`Number of arguments`	: `1`
`Argument 1`	: `MMF1 (string)`
`Returns`	: `nothing`

Adds the MMF1 to the list of MMF's used for PSM file retrieval.

`Request`	: `33 (REMMDLMMF)`
`Transparent`	: `No`
`Number of arguments`	: `1`
`Argument 1`	: `i (integer)`
`Returns`	: `nothing`

Removes the MMF indexed `i` from the list of PSM MMF's used for file retrieval.

`Request`	: `34 (SPECMDLMMF)`
`Transparent`	: `No`
`Number of arguments`	: `1`
`Argument 1`	: `i (integer)`
`Returns`	: `List of PSM file names (string)`

Returns a list of filenames of the PSM files held in the MMF indexed `i`.

`Request`	: `35 (GETOPT)`
`Transparent`	: `No`
`Number of arguments`	: `none`
`Returns`	: `PolyOrder (integer)`
	`CovarType (integer)`
	`DistanceType (integer)`
	`FrameOffset (long)`
	`VarianceFloor (float)`

```
ClusterPerturbation (float)
NumberOfRegions (integer)
DurationWeight (float)
OutputLabelDir (string)
NumOutputLabExtensions (integer)
[OutputLabelExt1] ... [OutputLabelExtN] (string)
OutputLabelMLF (string)
OutputSuffStatDir (string)
OutputSuffStatExt (string)
OutputSuffStatMMF (string)
OutputPSMDir (string)
OutputPSMMMF (string)
OutputDivisiveTreeDir (string)
MasterServerLoad (integer)
NumberOfSlaves (integer)
[SlaveName1 (string)
 SlaveLoad1 (integer)
 SlaveUid1 (unsigned long)
 SlavePid1 (unsigned long)]
   ...
[SlaveNameN (string)
 SlaveLoadN (integer)
 SlaveUid1 (unsigned long)
 SlavePidN (unsigned long)]
```

Returns the computation and output parameters currently set in the server. See also the SETOPT command below.

```
Request             : 36 (SETOPT)
Transparent         : No
Number of arguments : variable
Argument 1          : OptionId1 [Val1_1] [Val1_2] ... [Val1_N1]
Argument 2          : OptionId2 [Val2_1] [Val2_2] ... [Val2_N2]
...
Argument M          : OptionIdM [ValM_1] [ValM_2] ... [ValM_NM]
Returns             : nothing
```

Sets the option with id OptionId to the value(s) Val_1...Val_N. The mapping from OptionId to output/computation parameter is given by the following table. The string given between braces after the option ids are the names of the defines provided in the C header file.

OptionId	Define	Argument(s)	Description
0	(POLYORDER)	i	Polynomial order
1	(COVARTYPE)	i	Covariance type (0=diagonal 1=full)
2	(DISTTYPE)	i	Distance type (0=squared Euclidean, 1=Mahalanobis, 2=likelihood)
3	(FRAMEOFFSET)	l	Frame offset of first frame in 100ns units
4	(VARFLOOR)	f	Variance floor
5	(CLSTEPSIL)	f	Cluster perturbation parameter
6	(NUMREGS)	i	Number of regions of PSM models

7	(DURWEIGHT)	f	Duration model weight
8	(OUTPUTLBDIR)	t	Output label directory
9	(OUTPUTLBEXT)	i [t]	Output label extension
			0 = new extension
			-index = remove index'th extension
			index = replace index'ed extension
10	(OUTPUTLBMLF)	t	Output label MLF
11	(OUTPUTSSDIR)	t	Output sufficient statistics directory
12	(OUTPUTSSEXT)	t	Output sufficient statistics extension
13	(OUTPUTSSMMF)	t	Output sufficient statistics MMF
14	(OUTPUTMDLDIR)	t	Output PSM directory
15	(OUTPUTMDLMMF)	t	Output PSM MMF
16	(OUTPUTTREEDIR)	t	Output hierarchy tree directory
17	(MSERVLOAD)	i	Master server load
18	(SLAVESERV)	t i	Add slave server with name t and load i
19	(SLAVECLS)	i	Close i-th slave connection

```
Request             : 37 (RXCOMP)
Transparent         : Yes
Distributive        : No
Number of arguments : 1 or more
Argument 1          : RegexPat1 (string)
Argument 2          : RegexPat2 (string)
...
Argument M          : RegexPatM (string)
Returns             : A list of M regular expression ids (integer)
```

Attempts to compile the regular expression patterns given as arguments. On success, the server returns ids for these regular expressions which can be used to refer to the compiled regular expression patterns.

```
Request             : 38 (RXDEL)
Transparent         : Yes
Distributive        : No
Number of arguments : 1 or more
Argument 1          : RegexId1 (integer)
Argument 2          : RegexId2 (integer)
...
Argument M          : RegexIdM (integer)
Returns             : nothing
```

Frees the resources allocated to the regular expressions with the ids given as arguments.

```
Request             : 39 (FVLOAD)
Transparent         : Yes
Distributive        : No
Number of arguments : 1
Argument 1          : FVN1 (string)
Returns             : Feature vector file id (integer)
```

Attempts to load the feature vector file named `FVN1`. If successful, the id to be used in further referencing to these feature vectors is returned.

```
Request             : 40 (FVSPEC)
```

Transparent	: No
Number of arguments	: 1
Argument 1	: FVIdx (integer)
Returns	: #frames (long)
	framerate (long)
	featuretype (integer)
	frameoffset (long)

Returns specs for for the loaded feature vectors with id `FVIdx`. Number of frames, frame rate, feature vector type and the time offset of the first frame are returned.

Request	: 41 (FVREM)
Transparent	: Yes
Distributive	: No
Number of arguments	: 1
Argument 1	: FVIdx (integer)
Returns	: nothing

Frees the resources allocated to the feature vectors with id `FVIdx`

Request	: 42 (LBLOAD)
Transparent	: Yes
Distributive	: No
Number of arguments	: 2
Argument 1	: LBFN1 (string)
Argument 2	: FVIDX1 (integer)
Returns	: Label file index (integer)

Attempts to open the label file with name `LBFN1` and converts the times in the label files to frame indices using feature vectors with id `FVIDX1`. If the `FVIDX1` argument is set to -1, no conversion will be performed. On success, the id for the loaded label file, that is to be used in further referencing to this label file, is returned.

Request	: 43 (LBSAVE)
Transparent	: No
Number of arguments	: 4
Argument 1	: LBIDX1 (integer)
Argument 2	: FVIDX1 (integer)
Argument 3	: LBFN1 (string)
Argument 4	: LBLEVEL1 (integer)
Returns	: nothing

Attempts to write the label file with id `LBIDX1` to disk. If the `FVIDX1` is not set to -1, the label file times are assumed to be in frame indices and the feature vector file with id `FVIDX1` will be used for frame index to time conversion. The `LBLEVEL1` argument indicates which level this label level belongs to (determines which label extension to use).

Request	: 44 (LBCREATE)
Transparent	: Yes
Distributive	: No
Number of arguments	: 4 or more
Argument 1	: CONTFLAG (integer)
Argument 2	: IDFLAG (integer)

`Argument 3`	: `SCOREFLAG (integer)`
`Argument 4`	: `STARTTIME (integer)`
`Argument 5...N`	: `LAB/SCORE/TIME (string/float/long)`
`Returns`	: `Label file id`

Allocates a label file structure for the label file data provided in the arguments. The first argument should be non-zero if the label file is continuous, zero otherwise. The second argument should be non-zero if the label file's label-names will be provided as arguments and should be zero otherwise. If this flag is set to zero, the server will generate label-names in the form of "Ax" with x denoting the label index. The third argument should be set to non-zero if the following arguments specify scores for each label and should be set to zero otherwise. The fourth argument is the start-time of the segmentation (start time of the first label). All following arguments provide the label information. The arguments should describe the segmentation in a sequential order, starting at the first label. The first label's name is specified first (if the label file has labels). Then the score of the first label file is to be specified (if the label file has scores). Subsequently, the boundary time between the first and second label has to be given in case of a continuous segmentation. For a discontinuous segmentation, the end-time of the first segment followed by the start time of the second segment are to be given.

`Request`	: `45 (LBREM)`
`Transparent`	: `Yes`
`Distributive`	: `No`
`Number of arguments`	: `1`
`Argument 1`	: `LBIDX1 (integer)`
`Returns`	: `nothing`

Frees the resources allocated for the label file with id `LBIDX1`.

`Request`	: `46 (SSLOAD)`
`Transparent`	: `No`
`Number of arguments`	: `1`
`Argument 1`	: `SSFN1 (string)`
`Returns`	: `Sufficient statistics id (integer)`

Attempts to open the sufficient statistics file named `SSFN1`. On success, the id that is to be used in further referencing to the sufficient statistics file is returned.

`Request`	: `47 (MEANCOMP)`
`Transparent`	: `No`
`Number of arguments`	: `4`
`Argument 1`	: `SSIDX1 (integer)`
`Argument 2`	: `FVIDX1 (integer)`
`Argument 3`	: `SFRM1 (integer)`
`Argument 4`	: `EFRM1 (integer)`
`Returns`	: `Sufficient statistics id (integer)`

Computes the mean sufficient statistics for sufficient statistics record with id `SSIDX1`. If a new sufficient statistics record is desired this argument should be set to -1. The sufficient statistics are to be computed from the feature vectors with id `FVIDX1` for the segment starting at frame `SFRM1` up to and including frame `EFRM1`. The polynomial order and covariance type server parameters affect the actual computation that is performed. On success, the sufficient statistics id to be used in further referencing is returned.

`Request`	: `48 (COVARCOMP)`
`Transparent`	: `No`

Number of arguments	: 4
Argument 1	: SSIDX1 (integer)
Argument 2	: FVIDX1 (integer)
Argument 3	: SFRM1 (integer)
Argument 4	: EFRM1 (integer)
Returns	: nothing

Computes the covariance for sufficient statistics record with id SSIDX1. Using the mean derived from the sufficient statistics held in the record, the covariance is computed from the feature vectors with id FVIDX1 using frame SFRM1 up to and including frame EFRM1.

Request	: 49 (SSSAVE)
Transparent	: No
Number of arguments	: 2 or more
Argument 1	: SSFN1 (string)
Argument 2	: SSID1 (integer)
...	
Argument M	: SSIDM (integer)
Returns	: nothing

Write the sufficient statistics held in the records with ids SSID1, ..., SSIDM to a sufficient statics file named SSFN1.

Request	: 50 (LBSSWRT)
Transparent	: No
Number of arguments	: 3 or 4
Argument 1	: LBIDX1 (integer)
Argument 2	: FVIDX1 (integer)
Argument 3	: REGEXID1 (integer) (optional)
Argument 4	: SSFN1 (string)
Returns	: nothing

Computes sufficient statistics using the feature vector frames with id FVIDX1 for the segments defined in the label file with id LBIDX1 optionally filtered by the regular expression with id REGEXID1 to a file names SSFN1.

Request	: 51 (SSFILEREM)
Transparent	: No
Number of arguments	: 1
Argument 1	: SSFIDX1 (integer)
Returns	: nothing

Frees the resources allocated to the sufficient statistics file with id SSFIDX1.

Request	: 52 (SSSEGREM)
Transparent	: No
Number of arguments	: 1
Argument 1	: SSFIDX1 (integer)
Returns	: nothing

Frees the resources allocated for the sufficient statistics record with id SSFIDX1.

Request	: 53 (DISTVARWGHT)
Transparent	: No

Number of arguments	: 3
Argument 1	: FVIDX1 (integer)
Argument 2	: SFRM1 (integer)
Argument 3	: EFRM1 (integer)
Returns	: nothing

Estimates a diagonal covariance matrix to be used in subsequent Mahalanobis distance computations for feature vector file with id `FVIDX1` if the distance type is set to a Mahalanobis distance (returns successful without doing any computation otherwise). The frames to be used for the estimations are to be specified through the `SFRM1` (start frame) and `EFRM1` (end frame) arguments.

Request	: 54 (DISTDPWIN)
Transparent	: No
Number of arguments	: 4
Argument 1	: FVIDX1 (integer)
Argument 2	: EFRM1 (integer)
Argument 3	: MAXSFRM1 (integer)
Argument 4	: MINSFRM1 (integer)
Returns	: vector of distances

Computes distances for the segments (with respect to a model with parameters estimated from the segments themselves) starting at any frame between `MINSFRM1` and `MAXSFRM1` up to and including frame `EFRM1`. The used feature vector file is the one with id `FVIDX1`. On success, a vector with distances for the different start-times of the segment is returned.

Request	: 55 (UTTDPDIST)
Transparent	: No
Number of arguments	: 3
Argument 1	: FVIDX1 (integer)
Argument 2	: MINSLEN1 (integer)
Argument 3	: MAXSLEN1 (integer)
Returns	: distance matrix

Computes distances for all possible segments within the utterance with feature vectors with id `FVIDX1` under the constraint that each segment can not be larger than `MAXSLEN1` frames and not smaller than `MINSLEN1` frames. a distance matrix for all distances is returned where scores for all segments ending at frame i are given on row i of the matrix. The distance in the j-th column is for the segment starting at frame $i-j+1$ and runs up to and including frame i.

Request	: 56 (SSSETFA)
Transparent	: Yes
Distributive	: Yes
Number of arguments	: 2 or more
Argument 1	: SSIDX1 (integer)
Argument 2	: SSFN1 (string)
...	
Argument M	: SSFNM (string)
Returns	: Sufficient statistics set id (integer)
	Number of segments added to the set (integer)
	Number of frames in the segments added to the set (long)

Increases the number of sufficient statistics files held in the sufficient statistics set with id `SSIDX1`. If a new sufficient statistics set is desired, this argument should be set to -1. The file names of the sufficient statistics

```
Request             : 65 (INCSSCE)
Transparent         : No
Number of arguments : 1
Argument 1          : INCSSIDX1 (integer)
Returns             : nothing
```

Estimates the covariance parameter from the (incrementally) gathered sufficient statistics.

```
Request             : 66 (INCSSPC)
Transparent         : No
Number of arguments : 2
Argument 1          : INCSSIDX1 (integer)
Argument 2          : PSMSETIDX1 (integer)
Returns             : The id of the PSM set (integer)
```

After parameters are estimated from the incremental sufficient statistics, this function can be used to add the newly estimated PSM model to be added to a PSM set. The PSM model is estimated from the incremental sufficient statistics set with id `INCSSIDX1` and is added to the PSM set with id `PSMSETIDX1`. If a new PSM set is to be created, the `PSMSETIDX1` argument should be set to -1. On success the id of the PSM set is returned.

```
Request             : 67 (PSMSETMM)
Transparent         : Yes
Distributive        : No
Number of arguments : 2 or more
Argument 1          : PSMSETIDX1 (integer)
Argument 2          : PSMNM (string)
Argument 3          : PSMMW1 (float)
...
Argument M          : PSMMWN (float)
Returns             : nothing
```

Creates a mixture model from the mixture components held in PSM set with id `PSMSETIDX1`. The name of the new mixture model is `PSMNM`. Optionally, mixture weights can be specified by the remaining arguments `PSMMW1` through `PSMMWN`. The set is expected to contain N PSMs or an error will be reported. If mixtire weights are no specified, the weights are derived by the relative frequency of the PSMs in the training set.

```
Request             : 68 (PSMSETAM)
Transparent         : Yes
Distributive        : No
Number of arguments : 2 or more
Argument 1          : PSMSETIDX1 (integer)
Argument 2          : PSMFN1 (string)
...
Argument M          : PSMFNN (string)
Returns             : The id of the PSM set (integer)
```

Attempts to open the PSM files named `PSMFN1`, ..., `PSMFNN` and add these PSMs to the PSM set with id `PSMSETIDX1`. In fact, the server will first search the inventory of models currently available within the server's memory space. If the model is not available, the server will attempt to retrieve the model from disk. If a new PSM set is to be created, the `PSMSETIDX1` argument should be set to -1. On success the id of the PSM set is returned.

Request	: 69 (PSMSETAX)
Transparent	: Yes
Distributive	: No
Number of arguments	: 2 or more
Argument 1	: PSMSETIDX1 (integer)
Argument 2	: PSMMMIDX1 (integer)
...	
Argument M	: PSMMMIDXN (string)
Returns	: The id of the PSM set (integer)

Adds the mixture models (previously generated by PSMSETMM calls) specified by ids PSMMMIDX1 through PSMMMIDXN to the PSM set with id PSMSETIDX1. If a new PSM set is to be created, the PSMSETIDX1 argument should be set to -1. On success the id of the PSM set is returned.

Request	: 70 (PSMSETRM)
Transparent	: Yes
Distributive	: No
Number of arguments	: 2 or more
Argument 1	: PSMSETIDX1 (integer)
Argument 2	: [!]PSMNM1 (string)
...	
Argument M	: [!]PSMNMN (string)
Returns	: nothing

Removes the PSM named PSMNM1, ..., PSMNMN from the PSM set with id PSMSETIDX1. The initial exclamation mark character is optional for each PSM-name. If it's present, the resources allocated for the PSM are released for reuse. If the exclamation mark character is not present, the PSM is removed from the set but does remain in the server's memory space. This might avoid more time consuming disk access if any process connected to the server requests the model file to be loaded again.

Request	: 71 (PSMSETDS)
Transparent	: Yes
Distributive	: No
Number of arguments	: 2
Argument 1	: LINGFLG1 (integer)
Argument 2	: PSMSETIDX1 (integer)
Returns	: nothing

Releases the resources allocated for the PSM set with id PSMSETIDX1. If the LINGFLG1 argument is equal to zero, all the resources allocated to the models held within the PSM set are released. If the LINGFLG1 argument is positive, the models remain in the server's memory space possibly avoid excessive disk access in future requests.

Request	: 72 (PSMSETSV)
Transparent	: No
Number of arguments	: 3
Argument 1	: PSMSETIDX1 (integer)
Argument 2	: PSMSETNM1 (string)
Argument 3	: SSETIDX1 (integer)
Returns	: nothing

Writes the models contained in the PSM set with id PSMSETIDX1 to disk in a subdirectory named PSMSETNM1. If segment duration frequencies are not yet estimated, the SSETIDX1 index should be set to the id of the sufficient statistics set that is to be used for segment duration frequency estimation (PSMs are derived from clustering sufficient statistic records). If segment duration frequencies are already estimated (PSMs are estimated using incremental sufficient statistics records or are derived from disk retrieval) this argument can be set to -1.

Request	: 73 (SLVRFREQ)

This function is not to be called by the user at any time and is used for inter-server communication only.

Request	: 74 (PSMSETMD)
Transparent	: No
Number of arguments	: 1
Argument 1	: PSMSETIDX1 (integer)
Returns	: mean-duration derived from PSM segment duration frequencies (float)

Reports the mean durations of the PSMs held within the PSMset with id PSMSETIDX1. These mean-durations are estimated from the segment duration frequencies stored in the PSM files.

Request	: 75 (DIVTREESV)
Transparent	: Yes
Distributive	: No
Number of arguments	: 2
Argument 1	: PSMSETIDX1 (integer)
Argument 2	: PSMSETNM1 (string)
Returns	: nothing

Recursively traverses the model tree of models in the PSMset with id PSMSETIDX1 which was build by divisive clustering. It writes the found tree to disk in a file with name PSMSETNM1. To reset the process (stating that all PSMs in a set are independent top-level clusters) is achieved by the MARKTOP function (see below).

Request	: 76 (PSMSETCC)
Transparent	: Yes
Distributive	: No
Number of arguments	: 1
Argument 1	: PSMSETIDX1 (integer)
Returns	: nothing

Checks if the PSMs held in the PSM set with id PSMSETIDX1 are consistent in terms of polynomial order, covariance and number of regions. The function returns successful if the set is consistent, flags an error otherwise.

Request	: 77 (SERVERCC)
Transparent	: Yes
Distributive	: Yes
Number of arguments	: 0
Returns	: nothing

Checks for consistency between the master and connected slave servers in terms of polynomial order, covariance type, distance type and number of regions. The function returns successful if the server parameters are consistent, it flags an error otherwise.

Request	: 78 (SLVSRVCC)

This function is not be called by the user at any time and is used for inter-server communication only.

Request	: 79 (CMPCLSCT)
Transparent	: Yes
Distributive	: Yes
Number of arguments	: 3 or 4
Argument 1	: SSSETIDX1 (integer)
Argument 2	: PSMSETIDX1 (integer)
Argument 3	: CLUSTIDX1 (integer)
Argument 4	: PSMNAME1 (string) (optional)
Returns	: The id of the PSM set (integer)

Estimates the mean trajectory parameters of cluster indexed `CLUSTIDX1` in the PSM set with id `PSMSETIDX1`. The mean trajectory parameters are estimated from the sufficient statistics records labeled as belonging to the cluster with index `CLUSTIDX1` held in the sufficient statistics set with id `SSSETIDX1`. When a new PSM set is to be created, the `PSMSETIDX1` argument should be set to -1. This will result in mean-trajectory parameter estimation from all the data held in the sufficient statistics set for the first PSM/cluster in the set (indexed 0). In this case, a name can be given for the new PSM through the `PSMNAME1` argument. If such an argument is not provided, the server will assign an automatically derived unique name for the new cluster.

Request	: 80 (SLAVEMEANSS)

This function is not be called by the user at any time and is used for inter-server communication only.

Request	: 81 (CMPCLSCV)
Transparent	: Yes
Distributive	: Yes
Number of arguments	: 3
Argument 1	: SSSETIDX1 (integer)
Argument 2	: PSMSETIDX1 (integer)
Argument 3	: CLUSTIDX1 (integer)
Returns	: cluster distortion after re-estimation
	(Likelihood distance only) (float)

Estimates the covariance for the PSM/cluster with index `CLUSTIDX1` in the PSM set with id `PSMSETIDX1`. The sufficient statistic records used for the estimation are those held in the sufficient statistics set with id `SSSETIDX1` that are labeled as belonging to the cluster with index `CLUSTIDX1`. After successful completion of the covariance estimation, the new cluster distortion is reported in case a likelihood distance is used. The server returns the maximum float value otherwise.

Request	: 82 (SLAVECOVARSS)

This function is not be called by the user at any time and is used for inter-server communication only.

Request	: 83 (RECLASS)
Transparent	: Yes
Distributive	: Yes
Number of arguments	: 2 or 4
Argument 1	: SSSETIDX1 (integer)
Argument 2	: PSMSETIDX1 (integer)
Argument 3	: CLUSTIDX1 (integer) (optional)
Argument 4	: CLUSTIDX2 (integer) (optional)
Returns	: #segments in cluster 1 (integer)
	#frames in cluster 1 (long)
	distortion of cluster 1 (float)

```
#segments in cluster 2 (integer)
#frames in cluster 2 (long)
distortion of cluster 2 (float)
...
#segments in cluster N (integer)
#frames in cluster N (long)
distortion of cluster N (float)
```

Repartitions the data or part of the data held in the sufficient statistics set with id `SSSETIDX1` using the model inventory or part of the model inventory held in the PSM set with id `PSMSETIDX1`. If the third and fourth argument are not given, all data is partitioned using all PSMs (K-means style repartitioning step). In this type of operations, the server reports the sizes of the clusters held in the PSM set in terms of number of segments as well as number of frames and reports the cluster distortions after re-partioning. If 2 clusters are specified through the `CLUSTIDX1` and `CLUSTIDX2` arguments however, only the sufficient statistics labeled as belong to either PSM are reassigned to one of the 2 PSMs (divisive clustering). In this type of operation, the function reports on the new sizes and cluster distortions of the 2 specified clusters only.

```
Request              : 84 (SLVRCLASS)
```

This function is not be called by the user at any time and is used for inter-server communication only.

```
Request              : 85 (MARKTOP)
Transparent          : Yes
Distributive         : No
Number of arguments  : 1
Argument 1           : PSMSETIDX1 (integer)
Returns              : nothing
```

Marks all the PSMs held in the PSM set with id `PSMSETIDX1` as being top-level units. This means that any currently available information on hierarchical relationships between the models is destroyed and that further divisive clustering after completion of this function will indicate the models currently held in the PSM set to be independent top-level units.

```
Request              : 86 (SPLITCLST)
Transparent          : Yes
Distributive         : No
Number of arguments  : 3 or 4
Argument 1           : PSMSETIDX1 (integer)
Argument 2           : SSSETIDX1 (integer)
Argument 3           : CLUSTIDX1 (integer)
Argument 4           : PSMNAME1 (string) (optional)
Returns              : nothing
```

Increases the PSM inventory held within the PSM set with id `PSMSETIDX1` by 1. The new cluster has a mean-trajectory that is derived from the mean-trajectory of the PSM with index `CLUSTIDX1` by adding a perturbation. The size of the perturbation can be controlled by the user by setting the cluster perturbation parameter of the server. If a likelihood distance measure is used, the perturbation is proportional to variance of the original PSM model. A perturbation proportional to the variance of the sufficient statistics set with id `SSSETIDX1` is added if a Mahalanobis distance is used. If a squared Euclidean distance is used the perturbation is simply the cluster perturbation parameter itself. The name of the newly created PSM can be set through the `PSMNAME1` argument. If such an argument is not provided, the server will automatically generate a unique name for the new PSM.

`Request`	`: 87 (MERGECLST)`
`Transparent`	`: Yes`
`Distributive`	`: No`
`Number of arguments`	`: 4 or more`
`Argument 1`	`: SSSETIDX1 (integer)`
`Argument 2`	`: PSMSETIDX1 (integer)`
`Argument 3`	`: CLUSTIDX1 (integer)`
`Argument 4`	`: CLUSTIDX2 (integer)`
`...`	
`Argument M`	`: CLUSTIDXN (integer)`
`Returns`	`: nothing`

Removes clusters `CLUSTIDX2`, ..., `CLUSTIDXN` from the cluster inventory held in the PSMset with id `PSMSETIDX1` and assign all the data found in the sufficient statistics set with id `SSSETIDX1`, labeled as belonging to one of these clusters to cluster `CLUSTIDX1`. It then re-estimates the parameters of cluster `CLUSTIDX1` on the basis of the new set of data held in cluster `CLUSTIDX1`.

`Request`	`: 88 (ZAPCLUST)`
`Transparent`	`: Yes`
`Distributive`	`: No`
`Number of arguments`	`: 3 or more`
`Argument 1`	`: SSSETIDX1 (integer)`
`Argument 2`	`: PSMSETIDX1 (integer)`
`Argument 3`	`: CLUSTIDX1 (integer)`
`...`	
`Argument M`	`: CLUSTIDXP (integer)`
`Returns`	`: #segments in cluster 1 (integer)`
	`#frames in cluster 1 (long)`
	`distortion of cluster 1 (float)`
	`#segments in cluster 2 (integer)`
	`#frames in cluster 2 (long)`
	`distortion of cluster 2 (float)`
	`...`
	`#segments in cluster N (integer)`
	`#frames in cluster N (long)`
	`distortion of cluster N (float)`

Removes the clusters with indices `CLUSTIDX1`, ..., `CLUSTIDXP` from the PSMset with id `PSMSETIDX1` and reassigns all data held in the sufficient statistics set with id `SSSETIDX1`, labeled as belonging to one of these clusters, to the remaining PSMs in the PSMset. After successful completion of this task, the server reports on the new data partitioning and reports the sizes of all clusters remaining in the PSMset in terms of number of segments and frames as well as the cluster distortions.

`Request`	`: 89 (DPMATCH)`
`Transparent`	`: No`
`Number of arguments`	`: 9 or more`
`Argument 1`	`: FVIDX1 (integer)`
`Argument 2`	`: SWINLEN (integer)`
`Argument 3`	`: SWINSTART (integer)`
`Argument 4`	`: PPATHSCORE_1 (float)`
`...`	
`Argument N`	`: PPATHSCORE_SWINLEN (float)`

```
Argument N+1          : PSMSETIDX1 (integer)
Argument N+2          : CLUSTIDX1 (integer)
Argument N+3          : EWINLEN (integer)
Argument N+4          : EWINSTART (integer)
Returns               : END_PPATHSCORE_1 (float)
                        TRACEBACK_1 (integer) (optional)
                        PSMIDX_1 (integer) (optional)
                        END_PPATHSCORE_2 (float)
                        TRACEBACK_2 (integer) (optional)
                        PSMIDX_2 (integer) (optional)
                        ...
                        END_PPATHSCORE_EWINLEN (float)
                        TRACEBACK_EWINLEN (integer) (optional)
                        PSMIDX_EWINLEN (integer) (optional)
```

Performs a DP search from a start window to an end window using a particular PSM or PSM set. The start window is defined by the arguments `SWINSTART` which denotes the start of the start window and `SWINLEN` which denotes the length of the start window. The partial path scores for this start window are given by the argument `PPATHSCORE_1`, ..., `PPATHSCORE_SWINLEN`. The model to use to compute scores in the DP is the PSM with index `CLUSTIDX1` from the PSMset with id `PSMSETIDX1`. When the argument `CLUSTIDX1` is set to -1, the user indicates that any model in the PSMset should be considered. The end window to be used in the DP is specified through the `EWINSTART` argument which defines the start of the end window and the `EWINLEN` argument which defines the length of the end window. On successful completion of the DP, the server reports the partial path scores for the frames in the end window. If there are no valid scores reaching a particular frame in the end window, the server reports a score of "log(0)". For the frames that do have a valid DP partial path score, the server reports the likelihood, traceback time relative to the start of the start window and model index.

4 Extending the ASSM server

Although the server offers a large number of generally useful services, there is no doubt that there will be interest in extending the number of services offered by the server. This section will not give a very detailed description of the server internals as the code itself is very modular and thought to be very readable. The topics that will be discussed are those related to locating where the code should be altered to add new functionality and how to access globally defined data.

4.1 Adding a service

After the server received a request the trampoline function `ProcessRequest()` is called which will switch on the id of the new request and call the appropriate function. To add a new call to the trampoline, add a define for the new request in the include file `ASSMServProt.h`. Then add a switch for the new request id in the switch statement in the trampoline and make it call the entry function of the new function added to the server. It is strongly suggested that a new source module is added to the server when one desires to extend the services offered by the server (unless the new function is clearly part of an already existing source module).

A pointer to the received `Connection` structure is to be provided to the entry function from which it can extract the arguments required for the function and to which results are to be written after completion of the function. After successful completion of the function, the function has 3 tasks to complete before returning from the entry function if data is to be returned:

1. Set the request id i to $-i$.

2. Reset the arguments held in the `Connection` by setting the `NumArgs` and `ArgDataSize` fields to 0.

3. Append the data to be returned to the Connection structure as described in the (3.1.2) section.

If the function returns nothing, only the first step is required. If the function failed, the following 2 tasks are to be completed before returning from the entry function:

1. Reset the arguments held in the Connection by setting the NumArgs and ArgDataSize fields to 0.

2. Append an error message to the Connection structure.

After returning from the entry function, the data in the Connection structure is transmitted to the application.

4.2 Accessing globally defined data

When a service is added as described in the previous section, a few lines of code have to be added to the trampoline to render control to the new function defined in a new source module. The programmer is then free to define new local variables and use resources. For accessing the memory resources of the machine, three allocation functions are provided in the ASSM library named CheckedMalloc, CheckedCalloc and CheckedRealloc, each of which is described in the header file GenStuff.h.

The only other form of "close" contact the new module will have with the rest of the server is by means of accessing and possibly manipulating the globally defined data held within the server. The comments given in the global variable definition file Globals.h are probably self-explanatory but it might not be completely clear from the comments how the memory management is performed. Therefore, a few words on this topic.

All of the data of a particular type is held in one data array X. The size of the array is held in a separate variable SX and if new resources are required that are not available in the current array, the array size is enlarged by means of the CheckedRealloc function. Besides the data array X and the size SX of it, a second array XO of the same dimension as X of characters is maintained. There is a one-to-one relationship between the items in X and the items in XO. The items in XO are set to zero if the corresponding items in the data array X are not in use; they are set to a positive value otherwise. When new resources are requested, the XO array is first checked for unused records. If the requested resources cannot be provided by use of unused records in X, the X array (and therefore also the XO array) are expanded using the CheckedRealloc function.

For example, the PSM models are held in an array named PSMInventory. The available storage in terms of the number of PSM storage records is held in the variable with name NumPSMModels. Which of the available PSM storage records are actually in use is registered in the array PSMSlotOcc. For an example of memory management functions, refer to the first five functions in the PSMSets.c source file.

付録 A　ASSM file types

This appendix will give details on the file types used within the ASSM package. As the package is based on the HTK V1.5 library, the supported file-types are a superset of the file-types supported by the library.

A.1 Feature vectors

The feature vector files are to be given in HTK format and can be produced by the HCode tool of the V1.5 HTK package or by the HCopy tool of the HTK V2.0 package. In the original HTK V1.5 library one can only use feature vectors that were generated on a machine with the same byte order as the machine that generated the feature vectors. This limitation is removed so that feature vectors can now be shared among machines with different byte orders. Byte order differences are detected automatically making it completely transparent to the user. For details on the exact file format of an HTK feature vector file, please refer to the HTK manual.

A.2 Label files

A number of label file types are supported by the ASSM package as a number of label file formats are supported by the underlying HTK library. The expected label file format for label input and format of label output can be manipulated through environment variables and can also be manipulated through command-line switches in some of the tools. For details on supported file formats and relevant environment variables, please refer to the HTK manual.

A.3 Sufficient statistics files

Sufficient statistics files are not supported by the original HTK library and library support for these type of files is added on by the ASSM package.

Sufficient statistics files are binary files containing sufficient statistics for one or more segments. The file contains:

- Number of segments in the file.
- Vector dimensionality.
- Polynomial order used in segmentation.
- Polynomial order of sufficient statistics.
- Frame duration.
- Frame offset.
- Distance type.
- Covariance type.
- Segment durations.
- Segment distances.
- Mean trajectories.
- Segment covariances.

The binary format of the file:

- The number 1 for byte order detection. (short)
- A magic cookie (0xfe65a2b3) for automagical file-type detection. (unsigned integer)
- Number of segments in the file. (integer)
- Vector dimensionality. (integer)
- Polynomial order used in segmentation. (integer)
- Polynomial order of sufficient statistics. (integer)
- Frame duration in 100 ns units. (long)
- Frame offset in 100 ns units. (long)
- Distance type. (integer) (0=squared Euclidean, 1=Mahalanobis)
- Covariance type. (integer) (0=diagonal, 1=full)
- Segment durations. (integer)
- Segment distortions. (float)
- Segment mean trajectories. (float)
- Segment covariances. (float)

The mean-trajectory matrices are written row by row. For a diagonal covariance only the diagonal of the covariance matrix is written. For a full covariance, the lower triangle of the covariance matrix together with the diagonal are written row by row.

A.4 Polynomial Segment Model files

PSM files are not supported by the original HTK library and library support for these type of files is added on by the ASSM package.

The PSM file contains:

- Model name
- Vector dimensionality.
- Number of regions.
- Time warping method (only linear time warping is supported at this time)
- Polynomial order.
- Number of training segments.
- Number of training frames per region.
- Minimum segment duration.
- Maximum segment duration.
- Frequencies of segment durations.
- Mean trajectories.
- Covariance type.
- Covariances.

The binary format of the file:

- The number 1 for byte order detection. (short)
- A magic cookie (0x14dc0f3b) for automagical file-type detection. (unsigned integer)
- Vector dimensionality. (unsigned short)
- Number of regions. (unsigned short)
- Time warping type. (integer) (0=linear)
- Polynomial order. (unsigned short)
- Number of training segments. (unsigned integer)
- Minimum segment duration. (unsigned short)
- Maximum segment duration. (unsigned short)
- Covariance type (integer) (0=diagonal, 1=full)
- PSM name. (string)
- Number of frames per region. (long)
- Segment duration frequencies. (float)
- Mean trajectories per region. (float)
- Covariances per region (float)

Strings are written as an unsigned short indicating the length of the string followed by the string itself. The mean-trajectory matrices are written row by row. For a diagonal covariance only the diagonal of the covariance matrix is written. For a full covariance, the lower triangle of the covariance matrix together with the diagonal are written row by row.

A.5 Lattice files

Lattices were not yet supported in the HTK V1.5 library and library support for these type of files is added on by the ASSM package. The library module provides access to lattice files in 2 formats. The default format is the Standard Lattice Format (SLF) defined in the HTK V2.0 package. For a detailed description please refer to the HTK documentation. In addition to the SLF, an ATR lattice format is supported. This format is a superset of the SLF, adding a few additional fields:

- A word id field can be used rather than words alone. A word id is defined by the `WORDID` qualifier.
- The name of the acoustic model can be defined through the `amname` qualifier.
- The length of the utterance in seconds can be defined through the `utime` qualifier.
- The time offset of the times given in the lattice can be defines through the `abstime` qualifier.
- The CPU time used to generate the lattice can be specified through the `cputime` qualifier.

A.6 N-Best files

The format of the N-Best files that can be used in the ASSM package is:

- All items in a hypothesis are to be defined on separate subsequent lines.
- Hypotheses are separated by a special line of the form "///".
- Each line for which the first non-whitespace character equals the '#' character will be considered as a comment and ignored.

A.7 Dictionary files

As described in the section 2.6, the ASSM package provides support for a number of dictionary file formats. The dictionaries in one of these file formats can be converted to the standard ASSM dictionary file format using one of the provided tools. Finally, a file in the standard ASSM dictionary file format can be converted into a fast-loadable binary dictionary file. Of the supported dictionary formats, the HTK, PRONLEX and TIMIT dictionaries only support linear pronunciations (although multiple pronunciation are allowed) while the BU dictionary format defines pronunciation network but don't allow multiple pronunciation networks to be defined. The following section describes the supported dictionary formats, the ASSM standard dictionary format and the binary dictionary format.

(A.7.1) HTK format dictionaries

HTK format dictionaries are supported and can be converted using the `PreprocessHTKdict` utility. For a detailed description of the HTK dictionary format, please refer to the documentation provided with the HTK package.

(A.7.2) TIMIT format dictionaries

The TIMIT format dictionary is supported and can be converted using the `PreprocessTIMITdict` utility. The expected TIMIT dictionary format is defined as:

- Each dictionary entry is given on a separate line.
- A line for which the first non-whitespace character is a ';' character is considered a comment and all text on such a line is ignored.
- The format for a dictionary entry line is in the form of 2 or more words separated by white-space.

- The first word specifies the dictionary entry. Such a dictionary entry can have a pronunciation variation identifier appended to it. The boundary between the end of the dictionary entry and the start of the pronunciation variation is to be indicated by a ' ' character.
- The remaining words on the line describe the pronunciation of the dictionary entry in terms of units (usually phones). The first unit of the pronunciation should start with the '/' character and the last unit should end with the '/' character.
- Each unit on the line can have a stress-marker associated with it in the form of a '1' or '2' character directly following the unit's name.

For example, the following line is a valid example of a pronunciation for the word "use", pronunciation variation "verb":

```
use~verb  /y uw1 z/
```

(A.7.3) PRONLEX format dictionaries

The PRONLEX format dictionary is supported and can be converted using the `PreprocessPRONLEXdict` utility. The expected PRONLEX dictionary format is defined as:

- Each dictionary entry is given on a separate line.
- Multiple pronunciations for a dictionary entry can be defined by multiple definitions for the same dictionary entry.
- The format for a dictionary entry line is in the form of 2 or more words separated by white-space.
- The first word specifies the dictionary entry.
- The remaining words on the line describe the pronunciation of the dictionary entry in terms of units (usually phones).

For example, the following lines form a valid example of 2 pronunciations for the word "ARBITRAGE".

```
ARBITRAGE        aa r b ih t r ih jh
ARBITRAGE        aa r b ih t r aa zh
```

(A.7.4) Boston University format dictionaries

The BU format dictionary is supported and can be converted using the `PreprocessBUdict` utility. The expected BU dictionary format contains a header section and a body. The expected header section format is defined as:

- A line defining the size of a unit (phone) inventory containing all unit names of units used in the dictionary.
- A line containing the unit names of the inventory.
- A line defining the number of dictionary entries to be found in the body of the dictionary.

The expected body section format is defined as:

- Each dictionary entry is given on a separate line.
- Multiple pronunciations are to be given in the form of a pronunciation network.
- The format for a dictionary entry line is in the form of 5 or more words separated by white-space.
- The first word specifies the dictionary entry.

- The remaining words on the line describe the pronunciation network of the dictionary entry in terms of units (usually phones).
- Each phone in the network is defined as an arc starting from node 'x' ending a node 'y'. The arcs are defined by triplets that define the phone-label, then the start node and finally the end node. The entries in such triplets are to be separated by whitespace. Triplets themselves are also expected to be separated by whitespace.
- The end of a pronunciation network definition is indicated by the "//" word.

For example, the following line forms a valid example of the (linear) pronunciation network for the word "additional".

```
additional ae 0 1 d 1 2 ih 2 3 sh 3 4 ax 4 5 n 5 6 ax 6 7 1 7 8 //
```

(A.7.5) The ASSM standard dictionary format

For all the supported dictionary formats, utilities are provided that can convert a dictionary to the standard ASSM dictionary format. The standard ASSM dictionary format contains a header section and a body. The header section format is defined as:

- A line defining the size of a unit (phone) inventory containing all unit names of units used in the dictionary.
- A line containing the unit names of the inventory.
- A line with 5 flags which are either 0 or 1 indicating if a type of information is (1) or is not (0) available. The flags indicate if the:
  1. dictionary has multiple pronunciations
  2. dictionary has pronunciation probabilities
  3. dictionary has word ids
  4. dictionary has word equivalent id series
  5. dictionary has word frequencies.
- A line defining the number of dictionary entries to be found in the body of the dictionary.

The ids options allows the dictionary to be used for Japanese speech recognition as it avoids having to deal with the multiple orthographic representations that exist for lexical entries. The equivalent id series can be used to reveal information on how compound words are constructed.

The expected body section format is defined as:

- Each dictionary entry is given on a separate line.
- For a multiple pronunciation dictionary, the number of pronunciations is to be given on the same line as the entry, separated by whitespace. The ids of the multiple pronunciations are to be given after that, separated by whitespace. If the dictionary also supports pronunciation probabilities, each pronunciation variation id is to be followed by it's probability.
- If the dictionary supports word id's the dictionary entry line is extended further, defining the word id. The word id is to be separated from the rest of the line by whitespace.
- For a dictionary supporting equivalent word id series, the dictionary line is extended even further, providing the length of the id series and then the word ids in the series.
- Then for each of the pronunciation variations, a pronunciation network is to be defined on a new line.

- If the dictionary supports word frequencies, each pronunciation variation line starts with the frequency of the variation.
- The pronunciation network lines are then extended defining the number of nodes in the network, followed by the number of arcs in the network.
- The actual pronunciation network is defined by whitespace separated triplets defining model label, start node and end node in that order.

For example a valid entry for the word "live" which has id 445 and for the sake of the example is said to be equivalent to the word series 2, 973, has 2 pronunciations with ids "adj", probability -2.4, frequency 233 and "v" with probability -0.7, frequency 492 would be:

```
live 2 adj -2.4 v -0.7 445 2 2 973
233 3 2 1 0 1 ay 1 2 v 2 3
492 3 2 1 0 1 ih 1 2 v 2 3
```

(A.7.6) The ASSM binary dictionary format

The binary ASSM dictionary format contains all the information that is held in an ASSM standard dictionary format file but in binary form so that the dictionary can be loaded faster (no complex parsing is required). To make the loading and parsing even faster, index referencing of nodes to incoming and outgoing arcs is written explicitly in the dictionary file. One additional field is provided in the dictionary which allows a exponential weight factor to be associated with each arc in the pronunciation network (see the description of the `dictpack` and `latrescore` tools for details). A binary dictionary can be generated from an ASSM standard dictionary file using the `dictpack` utility. The contents of a binary dictionary can be viewed using the `lmdl` utility. The binary format of the file is:

- The number 1 for byte order detection. (short)
- A magic cookie (0xab7d00ef) for automagical file-type detection. (unsigned integer)
- Number of entries in the label hashtable. (integer)
- Number of entries in the dictionary. (long)
- A bit flag indicating presence or absence of certain types of information (unsigned short):
  - bit 0: word ids
  - bit 1: equivalent id series
  - bit 2: multiple pronunciations
  - bit 3: pronunciation probabilities
  - bit 4: word frequencies
- Label hash table (string)
- Dictionary entries (string)
- Number of pronunciation variations immediately followed by pronunciation variation ids (integer, string1, ..., stringN)
- Pronunciation probabilities for pronunciation variations. (float)
- Word frequencies. (long)
- Word ids. (long)
- Equivalent id series lengths immediately followed by id series. (integer, long1,..., longN)

- Number of arcs in the network. (integer)
- For each arc: Label idx, start node, end node and arc weight. (integer, integer, integer, float)
- Number of nodes in the network. (integer)
- For each node: Number of incoming arcs, number of outgoing arcs, in-arc ids, out-arc ids. (integer, integer, integer1, ..., integerN, integer1, ..., integerM)

Strings are written as an unsigned short indicating the length of the string followed by the string itself.

A.8 Pronunciation statistics files

Pronunciation statistics files are not supported by the original HTK library and library support for these type of files is added on by the ASSM package.

Pronunciation statistics files are binary files containing pronunciation statistics for one particular model. Pronunciation statistics file contain counts of observed unit sequences. A pronunciation statistics file contains:

- Number of unique units found in mapped sequences.
- Unit inventory found in the mapped sequences.
- Frequencies of units in the mapped sequences.
- Minimum sequence length.
- Maximum sequence length.
- Total number of mappings.
- Number of mapping per sequence length.
- Total number of unique sequences.
- Number of unique sequences per length.
- Mapped sequences with their frequencies.

The binary format of the file:

- The number 1 for byte order detection. (short)
- A magic cookie (0x7045ae9d) for automagical file-type detection. (unsigned integer)
- Size of the unit hash table. (integer)
- Minimum sequence length. (integer)
- Maximum sequence length. (integer)
- Total number of mappings. (long)
- Total number of unique sequences. (long)
- Unit hash table. (string)
- Unit frequencies. (long)
- Number of mapping per sequence length. (long)
- Number of unique sequences per length. (integer)
- For each unique sequence for each length: Sequence frequency followed by the sequence itself. (long, integer1, ..., integerN)

Strings are written as an unsigned short indicating the length of the string followed by the string itself.

A.9 Word-list files

A word-list file as produced by the `wordlistbuild` tool produces a word-list file in the following format:

- Each entry in the word-list appears on a separate line.
- The word-list entry is to appear as the first word or word sequence on the line.
- To define word or word sequence boundaries, the word or word sequence is to be captured between '"' characters.
- Each entry is to be followed by 3 fields which indicate word or word sequence frequency, number of word boundaries within the sequence and score respectively.

A.10 MasModelFile master files

The master file (`MasModelFile`) can contain PSM files and/or sufficient statistics files. The `MasModelFile` simply keeps track of the number of files that it contains and at which offset in the file the individual files start. To access a file contained within a `MasModelFile`, a lookup function is called which positions the stream pointer to the start of the desired file. The "regular" file IO function can subsequently be used for retrieval of the data from the file. Therefore, the information that is particular to the `MasModelFile` is:

- Number of files contained.
- File names of the contained files.
- Byte offsets of the files.
- Stream pointer to the opened `MasModelFile`

The binary file format of the `MasModelFile` is:

- Number of models. (integer)
- Header size. (long)
- Size of valid header contents. (long)
- File names. (string)
- Byte offsets. (long)

Strings are written as an unsigned short indicating the length of the string followed by the string itself.

A.11 Model Tree files

Divisive clustering causes a hierarchical relationship to be defined between PSMs. A detailed description of this hierarchical relationship is described in a model tree file. Such a model tree file is also used by other tools using model inventories. A skelleton of such a model tree file is provided in the package. The format of a model tree file is:

- Each model or model relationship is to be set has to be defined on a separate line.
- Each line for which the first non-whitespace character equals the '#' character will be considered as a comment and ignored.
- The first non-comment line of the tree file can optionally specify the hierarchy separator word used in the rest of the file and indicate if the tree file does or does not specify hierarchical relationships between models. The format of the first line is:

```
TREEDESCENDANTSEPARATOR <SEPARATOR> [HASDESCENDANTS]
```

The hierarchy separator should be a single word and should directly follow the keyword `TREEDESCENDANTSEPARATOR`. The keyword `HASDESCENDANTS` can optionally follow. If it is present, the rest of the file is expected to specify hierarchical relationships. Ommiting this line will cause default values to be used which are `->` as the separator and assuming no hierarchical relationships.

- A single model name on a line is to be used to define a new (top-level) model.
- Hierarchical descendant of a model are specified after the descendant separator as a white-space separated list.
- Mixture models are to be specified by first defining the mixture model name directly followed by a `=` character and then capturing the mixture component names between { braces on a single line.
- Mixture weights can be specified for the mixture components by adding a white-space separated list between `(` braces within the { braces.

A.12 Task-files for the `mdlreest` tool

The `mdlreest` tool requires a task description file to be given as an argument. This task-file is to describe what data is to be used for the re-estimation of a particular model. The format of such a task file is:

- Each target is to be given on a separate line.
- Each line for which the first non-whitespace character equals the '#' character will be considered as a comment and ignored.
- Each line is to consist of 2 or more whitespace separated words.
- The first word specifies the name of the model that is to be re-estimated.
- All the words following on the line define regular expression data filters. The data filters are applied in sequence but not in cascade. Any data remaining after applying the first data filter is used in re-estimation together with all the data that remains after applying only the second filter etc..

A.13 Server configuration files

The `fileinit` tool allows the user to configure the server with the parameters given in a configuration file. A skeleton of such a file ("ConfigFileSkel") is provided in the package. The format of a configuration file is:

- Each options that is to be set has to be defined on a separate line.
- Each line for which the first non-whitespace character equals the '#' character will be considered as a comment and ignored.
- The first word on the line is used to identify the option that is to be set. An option is identified using the names of the defines as given with service descriptions in section 3.2. by the ASSM server".
- Each option identifier is to be followed by pairs defining the value to be set. The first word in the pair is the single character type qualifier as defined in table 1. The second member of a pair is the value the option should take.
- Server computation and output parameters are a special case. Computation and output parameters are identified by 2 option identifier. The first one is `SETOPT` for all cases, the second one is one of the defines lists under the `SETOPT` function described in section 3.2.

A.14 Mixture specification files

The `dictpack` utility allows the user to do replacements of arcs with a particular label in the pronunciation networks in the dictionary with a number of parallel arcs, assigning an exponential weight factor to each of the parallel arcs in the new network. The arc replacement operation is defined by a mixture specification file with the following format:

- Each target is to be given on a separate line.
- Each line for which the first non-whitespace character equals the '#' character will be considered as a comment and ignored.
- Each line is to consist of 2 or more whitespace separated words.
- The first word specifies the name of the arc that is to be replaced by a number of parallel arcs.
- The remaining words on the line specify the replacement arcs and their weights. The label of a replacement arc is to be given directly followed by it's weight.

付録 B Compiling the toolkit

The package requires the following software tools to be available:

- The GNU C compiler. The package will most likely compile with other C compilers as well but this has never been tested.
- The Perl5 interpreter.
- The GNU make utility version 3.74 or later.
- The GNU gzip compression utility.
- The GNU tar utility is preferred but not required.

All utilities above are freely available by anonymous ftp from prep.ai.mit.edu. In addition, a license for the HTK toolkit is required to use the ASSM package.

Provided that the software tools are available, the package can be build. The first step consists of extracting the package files from the archive file or tape. If the package is to be extracted from the archive named `ASSM.3.0.tar.gz` and the `GNU tar` utility is available, the following commands will extract the package and write the files to a subdirectory named `ASSM_3.0`:

```
tar xvfz ASSM.3.0.tar.gz
```

The `ASSM_3.0` subdirectory will have 30 subdirectories. To make sure that all files are properly extracted, one could verify that the list of files in the file `PackageFileList` are actually available under the `ASSM_3.0` directory tree.

After successful extraction of the files in the package, the next step is to configure the package. The configuration is achieved by editing two files in the package. The `Make.glob` file in the top-level directory of the file will after extraction look like:

```
MACHINE         = LINUX
#MACHINE can take the values: HPUX SUN SOLARIS ALPHA LINUX
#EXTRAINCL      = -I../include/Cincludes
EXTRAINCL       =
OPTDEB          = -g -O3
CC              = gcc
PERLINTERP      = /usr/bin/perl -w
DESTDIR         = /home/bacchian/bd/src/ASSM_3.0.Linux/bin/
```

This file should be tailored to fit the desired installation. The MACHINE variable should be set to the machine the package is built for. The package is tested extensively under the SOLARIS 2.5.1, LINUX 2.0.27 and OSF1 operating systems and has been succesfully built under the SunOs 4.1.4 and HPUX 9.05 operating systems. If the package is compiled on a Sun4 machine it will be necessary to uncomment the EXTRAINCL line and comment the empty definition. This should be unnecessary for other platforms. Flags can be passed to the compiler through the OPTDEB variable. The compiler itself can be set by definition of the CC variable. The command that is to be used to execute the Perl interpreter is to be defined through the PERLINTERP. The binaries of the package will be written to the directory set through the DESTDIR variable.

After these variables are appropriately defined, the file ServerDefines.h in the subdirectory include is to be edited. Directly after extracting from the archive, this package will look like:

```
#ifndef _ServerDefines_
#define _ServerDefines_

#ifdef __cplusplus
extern "C" {
#endif

#define UDSDIR "/tmp/"
#define ASSMSERVERUDS "/tmp/ASSMserver.uds"

#define ASSMSERVERTCPIPPORT 5555

#define ASSMSERVERLOG "/home/bacchian/bd/src/ASSM_3.0/logs/ASSMserver.log"

#define STALE 20

#ifdef __cplusplus
}
#endif

#endif
```

The directory defined for the UDSDIR is the directory where the server and application can write UNIX Domain Sockets (UDS). The server will on startup create a listen UDS on which it listens for incoming connection from applications. The full path of this UDS is to be defined for ASSMSERVERUDS. The server will also listen to the port defined for ASSMSERVERTCPIPPORT for incoming TCP/IP connections. One should make sure that the port number defined for this variable is not conflicting with any port used for another network service. For a list of used ports on your system, please refer to the file /etc/services. Finally the core name of the server log file is to be defined for ASSMSERVERLOG. The actual log files that are created will be this core name, appended with the name of the machine the server is running on and the PID of the server process.

One should keep in mind that if the package is used among machines running different operating systems, sharing disks through the Network File System (NFS), problems can arise if a UDS is written on a remote disk. Not only will one loose the speed advantage of a UDS connection over a TCP/IP connection (data is in this case still sent through the network while the UDS connection tries to avoid this for the purpose of communication efficiency), in many cases, the file type of a socket on a remote disk is confused causing the establishment of a connection to fail. If the UDS directory is created in the tmp directory, the UDS is guaranteed to be on a local disk.

After completing the configuration steps, the final step will consist of giving the command make in the top directory of the package causing the package the binaries to be created in the defined destination directory.

Make sure that the destination directory does exist before the `make` command is given.

After the `make` command finishes, the libraries that application C programmers need to link against if library functions are called from their applications are found in the `lib` subdirectory of the package. Two static libraries are to be found with the name `libServ.a` and `libASSM.a`. The header files that provided prototypes for the functions defined in these libraries can be found in the `include` subdirectory of the package. The header file defining the functions available in the `Serv` library are `GenServer.h`, `UDSserv.h` and `TCPIPserv.h`. All other include files in the `include` directory are function definitions of the functions provided in the `ASSM` library. A reminder of the server functions and their arguments are provided in the `ServerProtocol.txt` file in the `ASSMserver` subdirectory of the package.