SAM
Sequence Alignment and Modeling
Software System

Richard Hughey Anders Krogh

sam-info@cse.ucsc.edu
Baskin Center for Computer Engineering and Science
University of California
Santa Cruz, CA 95064

Technical Report UCSC-CRL-96-22
Updated for SAM Version 2.1.1
April, 1998

Contributors:
Christian Barrett, Michael Brown, Melissa Cline, Mark Diekhans,
Leslie Grate, Rachel Karchin, Kevin Karplus, Kimmen Sjölander,
Christopher Tarnas

supersedes UCSC-CRL-95-07

Abstract

The Sequence Alignment and Modeling system (SAM) is a collection of flexible software tools for creating, refining, and using linear hidden Markov models for biological sequence analysis. The model states can be viewed as representing the sequence of columns in a multiple sequence alignment, with provisions for arbitrary position-dependent insertions and deletions in each sequence. The models are trained on a family of protein or nucleic acid sequences using an expectation-maximization algorithm and a variety of algorithmic heuristics. A trained model can then be used to both generate multiple alignments and search databases for new members of the family. SAM is written in the C programming language for Unix machines, and includes extensive documentation.

The algorithms and methods used by SAM have been described in several pioneering papers from the University of California, Santa Cruz. These papers, as well as the SAM software suite, several servers, and links to related sites such as HMMer are available on the World-Wide Web to `http://www.cse.ucsc.edu/research/compbio/sam.html`

Contents

1 Introduction

The Sequence Alignment and Modeling system (SAM) is a collection of software tools for creating, refining, and using a type of statistical model called a linear hidden Markov model for biological sequence analysis. Linear hidden Markov models only model primary structure (sequence) information; long-range iterations, such as base pairing in RNA, require more complex models such as stochastic context-free grammars, as described by Sakakibara *et. al* (NAR 22(23):5112–5120), also available from the UCSC computational biology WWW site.

The algorithms and methods have been described in several papers, some of which are available via anonymous ftp to `ftp.cse.ucsc.edu` in the `protein` directory, as well as on our WWW site,

`http://www.cse.ucsc.edu/research/compbio/sam.html`.

The primary papers from UCSC (copies of these papers and several others are available from the SAM WWW site) include:

- A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531, February 1994.
- R. Hughey and A. Krogh. Hidden Markov models for sequence analysis: Extension and analysis of the basic method, CABIOS, 12(2):95–107, 1996.
- R. Karchin and R. Hughey. Weighting hidden Markov models for maximum discrimination Bioinformatics, to appear, 1998.
- C. Tarnas and R. Hughey. Reduced space hidden Markov model training. Bioinformatics, to appear, 1998.
- K. Karplus, Kimmen Sjölander, C. Barrett, M. Cline, D. Haussler, R. Hughey, L. Holm, and C. Sander, "Predicting protein structure using hidden Markov models," *Proteins: Structure, Function, and Genetics*, Supplement 1, 1997.
- K. Sjolander, K. Karplus, M. Brown, R. Hughey, A. Krogh, I.S. Mian, and D. Haussler. Dirichlet Mixtures: A Method for Improving Detection of Weak but Significant Protein Sequence Homology. CABIOS 12(4), 1996.
- C. Barrett and R. Hughey and K. Karplus. Scoring Hidden Markov Models. CABIOS 13(2):191–199, 1997.
- J. A. Grice, R. Hughey, and D. Speck. Reduced space sequence alignment. CABIOS 13(1):45–53, 1997.
- D. Haussler, A. Krogh, I. S. Mian, and K. Sjölander. Protein modeling using hidden Markov models: Analysis of globins. In *Proceedings of the Hawaii International Conference on System Sciences*, volume 1, pages 792–802, Los Alamitos, CA, 1993. IEEE Computer Society Press.
- A. Krogh, I. S. Mian, and D. Haussler. A hidden Markov model that finds genes in *E. coli* DNA. *Nucleic Acids Research*, 1994.

- R. Hughey and A. Krogh. SAM: Sequence alignment and modeling software system. Technical Report UCSC-CRL-96-22, University of California, Santa Cruz, CA, September 1996.
- K. Karplus. Regularizers for Estimating Distributions of Amino Acids from Small Samples. Technical Report UCSC-CRL-95-11, University of California, Santa Cruz, CA, 30 March 1995.

We would appreciate references to the first two articles in work that cites or uses the SAM system.

Because the software is an active research tool, there are a vast selection of options, many of which have, through experimental study, been set to reasonable defaults.

If you have suggestions for enhancements, new ways of using SAM, or other comments, please contact us.

SAM incorporates the readseq package by D. G. Gilbert, who allows it to be freely copied and used. The `hmmedit` and `sae` programs use ACEdb by Richard Durbin and Jean Thierry-Mieg. The source code for `hmmedit` and `sae` is available from ftp://ftp.cse.ucsc.edu/pub/protein/hmmeditsaesrc.tar.Z.

If you wish to be informed of future releases, please send your email address to `sam-info@cse.ucsc.edu` for addition to our mailing list. Please also use this address for any questions or comments you may have.

You will also find Sean Eddy's system, HMMER (http://genome.wustl.edu/eddy/hmm.html), to be of interest.

1.1 Acknowledgments

We thank I. Saira Mian and Finn Drablos for their important evaluations of the system, and Kevin Karplus for the prior library. Finally, we thank the entire UCSC Baskin Center Computational Biology group, led by David Haussler, who got this whole thing started. This work was supported in part by NSF grants CDA-9115268, IRI-9123692, BIR 94-08579, and MIP-9423985; DOE grant 94-12-048216; ONR grant N00014-91-J-1162; NIH grant GM17129; a grant from the Danish Natural Science Research Council; and a gift from Digital Electronics Corporation.

2 Version enhancements

2.1 Version 2.1.1

April, 1998. Minor revisions and updates; first externally-released revision of Version 2.1.

- User-defined alphabets for sequences. See Section 6.1.1 on page 23.

- Negative fimstrength values will adjust both insert and FIM states. See Section 7.5 on page 37.

2.2 Version 2.1

Modifications and improvements to the hmmscore program are the highlight of this upgrade. February, 1998.

- The multdomain program has been removed and its function has been merged into hmmscore. The mdNLLminusNULL parameter has been renamed mdNLLnull. The multdomainshort parameter has been renamed alignshort. The old names are currently aliased to the new names for these two parameters. See Section 9.2.5 on page 66.
- The hmmscore program can now print selected sequence alignments and selected sequence multiple domain alignments during scoring. See Section 9.2.3 on page 63 and Section 9.2.5 on page 66.
- The interactive mode of hmmscore has been removed. See Section 9.2 on page 56.
- The scored sequence letter counts null model has been removed. Null model scores can now be calculated based on the reverse sequences. The simple_threshold variable determines when complex null model calculations should be performed in terms of the simple null model score. See Section 9.2.1 on page 59.
- The content of score files has changed, as has the use of select_seq, select_score, sort, and subtract_null. See Section 9.2 on page 56.
- The uniqueseq program has been updated. See Section 9.7.5 on page 84.
- The checkseq program has been updated. See Section 9.7.1 on page 82.
- Scoring examples in this manual have been changed to use fully-local scoring and hmmscore now prints a warning whenever fully-local scoring is not used. See Section 9.2.4 on page 64.
- The protein_prior and nucleotide_prior variables can be used to specify default prior libraries. The Dirichlet mixture recode1.20comp is now used by default with protein sequences. See Section 7.1 on page 26.
- Internal weighting in buildmodel is now by default turned on with internal_weight set to 1. If an external weight file is specified and internal_weight is not explicitly set on the command line, internal weighting will be turned off. See Section 8.4.3 on page 47.
- The sequence_models variable, when set, causes buildmodel to create initial models from random sequences in the training set which are then regularized. The each single sequence is given a weight equal to the value of sequence_models. This option is recommended and is expected to become default behaviour in a future release. It can both reduce runtime by providing an initial starting point when an alignment is not available and increase modeling performance. See Section 7.3 on page 31.
- The seed_runs parameter has been removed from buildmodel.

2.3 Version 2.0

November, 1997.

- A complete rewrite of the inner dynamic programming loop to save memory (see the Grice, Hughey, and Speck, and the Tarnas and Hughey papers mentioned in the introduction) and allow local and semi-local scoring and alignment, as well as Viterbi-based training. Memory use is now proportional to the product of model length and the square root of the sequence length rather than the model length and the sequence length. See Section 9.1.1 on page 53 and Section 9.2.4 on page 64.
- HSSP-based structural transistion regularizer. See Section 7.1.2 on page 28.
- The `multdomain` program now performs scoring adjustments identical to those of `hmmscore` when SW is set. See Section 9.2.4 on page 64.
- Internal sequence weighting inspired by HMMer's Maximum Discrimination method has been implemented. It significantly increases discrimination performance in the presense of biased training sets. See Section 8.4.3 on page 47.
- The `a2mdots` variable can be cleared to avoid printing dots in a2m files, leading to an at times considerable space reduction. See Section 9.2 on page 56.
- The `hmmscore` program can parition a database to aid in distributed scoring. See Section 9.2.6 on page 69.
- SAM will now read its input from compressed (.gz or .Z) files. See Section 11 on page 86.
- The alphabet code has been rewritten to make it simpler for those with source licenses to modify the code.
- Weight files for `buildmodel` initial alignments and `modelfromalign` alignments can be specified with the `alignment_weights` parameter. See Section 8.4 on page 44.
- A broader collection of Dirchlet mixture and transition regularizers is included with this version. See Section 7.1 on page 26.
- The MasPar implementation is no longer supported.

2.4 Version 1.4

August, 1996.

- The geometric average of the match state probabilities is now available for use (and the default) with simple and complex null models. Complex null models are now built from the transition and insert probabilities of the model, and the geometric average of all the model's match tables in the match table. See Section 9.2.1 on page 59.
- The `train_reset_inserts` variable causes `buildmodel` to, at the completion of reestimation cycle, reset all the insertion and FIM tables to (by default) the geometric average of the match states. Set to 0 to turn off. See Section 7.6 on page 38.

- If no IDs are specified, the `hmmscore` and `multdomain` programs will read in sequences a few at a time, instead of all at once, saving a tremendous amount of memory. If this is done, sequence output by `hmmscore` is no longer sorted by score, though the score file can still be sorted. Given a sorted score file and unsorted sequence file, the new `sortseq` program will sort the sequences according to the score file. See Section 9.2 on page 56, Section 9.2.5 on page 66, and Section 9.7.4 on page 83.
- The `uniqueseq` program will eliminate sequences with duplicate IDs from a file. The `checkseq` program will read a sequence file and print information about it. See Section 9.7.5 on page 84.
- Training noise is reduced by `retrain_noise_scale` (default 0.1) whenever an initial model or alignment is provided. Noise is also reduced between the first and successive surgery iterations by `surgery_noise_scale` (default 0.1). See Section 8.1 on page 40.
- Models can be edited using the new utility program, `modifymodel`. See Section 9.5.3 on page 75.
- The program `makehist` will turn one or two `.dist` score file into a histogram, `makeroc` will turn two `.dist` score files into a false positive/false negative plot showing score vs. counts (number of sequences with the score) and `makeroc2` will turn two `.dist` score files into a plot of false positive vs. false negative as a function of threshold score. All three programs require `gnuplot`. See Section 9.6 on page 78.
- Weight file reading is more robust. We plan to implement a WWW weighting server which, given a multiple alignment, will return sequence weights under a variety of weighting schemes.
- The `a2mallcaps` variable has been removed: `modelfromalign`, `buildmodel`, and other alignment reading routines will first check to see if the file is an HSSP file, if not, the a2m format will be checked, and if that does not result in every sequence having the same number of columns, all characters will be treated as uppercase. See Section 7.3 on page 31.
- Binary model output. Models can be printed in human unreadable binary form. This reduces file size to about one quarter, and greatly increases model reading speed. See Section 7.4.3 on page 36.

2.5 Version 1.3

May, 1996.

- Weighted training. See Section 8.4 on page 44.
- Sequence weight annealing. See Section 8.4.2 on page 47.
- The ability to use files as model type specifiers rather than keywords such as REGULARIZER. See Section 4 on page 19.
- The ability to print scores of only those sequences doing better than some threshold. See Section 9.2 on page 56.
- When multiple models are trained, training is stopped for each model individually according to the `stopcriterion`. Previously, training was stopped when the average score difference reached the `stopcriterion`. See Section 8 on page 39.

- `Modelfromalign` can be told to treat all letters as match columns, and turns the letter 'O' (capital 'o') into a FIM. See Section 9.4 on page 73.
- Efficiency improvements to model reading.
- SAM alignments have undergone significant changes. `Align2model` output is now in a normal sequence format, though still with uppercase, lowercase, '.' and '-' meanings. `Prettyalign` can read any `readseq` format, with lower-case letters indicating insertions. `Prettyalign` can no longer be used as a pipe. `Modelfromalign` can read any `readseq` format. The `buildmodel` program can be given an initial alignment. See Section 9.4 on page 73.
- The method for specifying multiple database files or multiple sequence IDs has changed. Multiple `db` or `id` declarations on the command file or a parameter file will add to a list of database files or id files.
- The command lines for many programs has changed. Except for `prettyalign`, all programs now take arguments in the form of a run name followed by variable name and value pairs. See Section 5 on page 21.
- The `modelfromalign` program now uses prior libraries. See Section 9.4 on page 73.
- FIM normalization has been moved to another place in the code, and can be avoided if desired. See Section 9.3.1 on page 71.

2.6 Version 1.2

March, 1996.

- The ability to globally apply various FIM and insertion table settings to the regularizer during training and to the model during scoring. This reflects a general cleaning up of the log-odds scoring introduced in 1.1. The defaults are to use training set letter counts in both FIMs and insert states for training, and match state frequency averages for FIMs during scoring (with no change to the insert states). See Section 7.6 on page 38. See Section 9.2.1 on page 59.
- The ability to score sequences according to the difference between two models, such as models trained on positive and negative family examples. See Section 9.2.1 on page 59.
- By default, `hmmscore` and `multdomain` will add FIMs to a model before scoring it. See Section 9.2.1 on page 59.
- By default, SAM will start with three models of random length, and then pick the best model for surgery and further reestimation. This will increase runtime from Version 1.1, but improves model generation.
- Several new parameters exist. See Section 11 on page 86.
- An interface is provided between HMMer and SAM. See Section 9.5.4 on page 76.
- Non-default initial models are no longer printed in model files. This led to too much confusion about which model was the real one, as well as those pesky "non-default model being replaced" messages.

- The use of Dirichlet mixture priors has been updated to reflect in our most recent work (`http://www.cse.ucsc.edu/research/compbio/dirichlet.html`). The format of prior libraries has changed slightly, and only one prior library (`uprior9.plib`) is included in the distribution. **Mixtures in the earlier format may crash the program.** Mixture priors are particularly useful in database search from a small set of training examples. See Section 7.1 on page 26.

2.7 Version 1.1

November, 1995.

- The default protein regularizer has been changed from having the uniform distribution in the insert states to having the background distribution. This generally helps discrimination experiments, though may hurt sequence alignment. See Section 7.1 on page 26.
- The default scoring has been changed from calculating Z-scores using length bins to NULL model subtraction, which accounts for both sequence length and wildcards. For scoring, FIMs must be added to a model before it is scored for this to produce valid results. This corresponds to the log-odds scoring used in Sean Eddy's HMMER. See Section 9.2.1 on page 59.
- Scoring and training with wildcards has been modified so that sequences with many wildcards can be properly scored with null models.
- An iterative program for finding multiple motifs in a single sequences is part of SAM. See Section 9.2.5 on page 66.

2.8 Version 1.0

January, 1995.

- First general release.

3 Quick overview

The Sequence Alignment and Modeling (SAM) suite of programs includes several tools for modeling, aligning, and discriminating related DNA, RNA, and protein sequences. Given a set of related sequences, the system can automatically train and use a linear HMM representing the family.

SAM uses a linear hidden Markov model (Figure 1) to represent biological sequences. The model is a linear sequence of *nodes*, each of which includes *match* (square), *insert* (diamond), and *delete* (circle) states. Each match state has a distribution over the appropriate alphabet indicating which characters are most likely. The chain of match states forms a model of the family, or of columns of a multiple alignment. Some sequences may not have characters in specific positions — delete states enable them to skip through a node without 'using up' any characters. Other sequences may

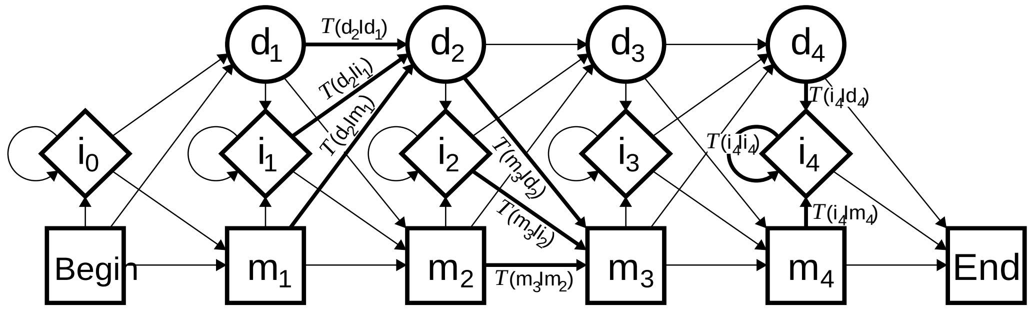


Figure 1: A linear hidden Markov model.

have extra characters, which are modeled with the insert states. Insertions are thus used when a small number of sequences have positions not found in most other sequences, while delete states are used when a small number of sequences do not have a character in a position found in most other sequences. As with the match states, all transition probabilities (the chance of having a delete or moving to an insertion state) are local, enabling, for example, the system to strongly penalize sequences that delete conserved regions.

The primary programs include:

buildmodel Create a new model from a family of sequences, or refine an existing model.

align2model Create a multiple alignment of sequences to an existing model. The `prettyalign` program will make `align2model` output more readable.

hmmscore Calculate the negative log-likelihood (NLL) scores for a file of sequences given a model, as well as smooth curves and Z-scores. This program is used for discrimination experiments. Sequences that score better than (or worse than) a threshold can be saved, as can their alignments or multiple domain alignments.

modelfromalign Use an existing multiple alignment to create an initial model. Such a model is usually then refined using `buildmodel`.

addfims Add free insertion modules to models trained on clipped and complete sequences.

sampleseqs Generate typical sequences from a model.

hmmer2sam, sam2hmmer Convert, as much as possible, between the two HMM formats.

makehist Create a histogram from a score file. This is an excellent means of viewing model performance.

makeroc Plot false negatives and false positives against score. Critical for discrimination experiments.

makeroc2 Plot false negatives vs. false positives as a function of threshold score. Uses linear interpolation to create a smooth curve.

modifymodel Delete, add, or change nodes and subdivide or catenate models.

readseq A modified copy of Don Gilbert's `readseq` program for format conversion is included in the `readseq` subdirectory. You may whish to compile the standalone version, as it is exceedingly useful.

A basic flowchart for using SAM is shown in Figure 2.

As a simple example, consider the task of modeling the 10 tRNAs included in the file `trna10.seq` of the distribution. For this experiment, default program settings will be used: the many adjustable parameters are described Sections 5 and 11.

3.1 Building a model

To start, we need to create a model from the sequence file using `buildmodel`. This program always requires a name for the run: if the name is `test`, the system will create the model output file `test.mod`, which will include parameter settings, iteration statistics, and CPU usage, as well as the initial and final model.

Parameters to buildmodel are specified with hyphens. For this experiment, first we try the command:

```
buildmodel test -train trna10.seq
```

This specifies the run name and where the sequences for training can be found. All the sequences will be used to train the model, though often sequences will be reserved to test the model as well. Having no other information, the program will check the first sequence of the sequence file to determine what alphabet to use. In this case, the check tells the system to use the RNA alphabet. The alphabet can also be specified on the command line if the sequence data is not conclusive:

```
buildmodel test -alphabet RNA -train trna10.seq -seed 0
```

Here, to hopefully make this example reproducible, a seed for the random number generator has also been specified.

Buildmodel then prints out a line on standard output such as:

```
-34.24  -29.07  -30.16   1.55  68 3 74
```

This is a brief summary of the statistics provided in the output model file, and is discussed in more detail in Section 10.1. If no random seed had been specified, `buildmodel` would use the process number, and the statistics line would be different for multiple runs.

The run has generated a file called `test.mod`. This file contains various statistics, described later, as well as the final model. Statistics are printed to the file after each reestimation so that the progress of a run can be readily checked.

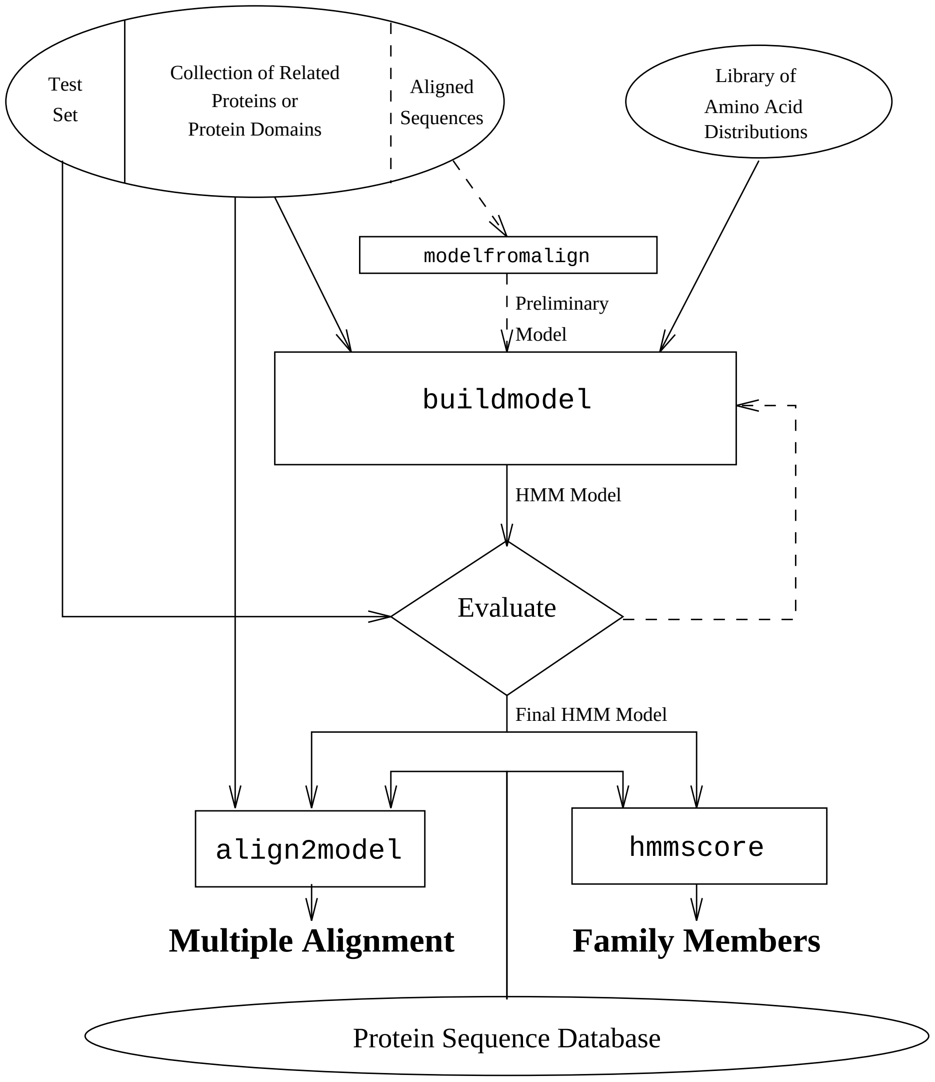


Figure 2: Overview of SAM.

3.2 Aligning sequences

To generate a multiple alignment, a command such as the following is used:

```
align2model trna10 -i test.mod -db trna10.seq
prettyalign trna10.a2m -l90 > trna10.pretty
```

This aligns each sequence to the model, places the alignment in the file `trna10.a2m`, and then cleans up the output and places it in the file `trna10.pretty`, with 90 characters per line. The alignment will look something like:

```
;  SAM: ../src/prettyalign v2.1.1  (Apr 24, 1998) compiled 04/27/98_12:32:27.
;  SAM:  Sequence Alignment and Modeling Software System
;  (c) 1992-1998 Regents of the University of California, Santa Cruz
;  http://www.cse.ucsc.edu/research/compbio/sam.html
;
; ------ Citations (HMMs, SAM) ------
; A. Krogh et al., Hidden Markov models in computational biology:
;   Applications to protein modeling, JMB 235:1501-1531, Feb 1994.
; R. Hughey, A. Krogh, Hidden Markov models for sequence analysis:
;   Extension and analysis of the basic method, CABIOS 12:95-107, 1996.
; -----------------------
               10            20         30        40        50        60        70
                |             |          |         |         |         |         |
TRNA1  GGGGAUGUAGCUCAG-.UGG...U.AGAGCGCAUGCUUCGCAUGUAUGAGGCCCCGGGUUCGAUCCCCGGCAUCU---CCA
TRNA2  GCGGCCGUCGUCUAGU.CUGgauU.AGGACGCUGGCCUCCCAAGCCAGCAAUCCCGGGUUCGAAUCCCGGCGGCCG---CA
TRNA3  GGCCCUGUGGC-UAGC.UGG...UcAAAGCGCCUGUCUAGUAAACAGGAGAUCCUGGGUUCGAAUCCCAGCGGGGCCUCCA
TRNA4  GGGCGAAUAGUGUCAG.CGG...G.AGCACACCAGACUUGCAAUCUGGUAGGGA-GGGUUCGAGUCCCUCUUUGUCCACCA
TRNA5  GCCGGGAUAGCUCAGU.UGG...U.AGAGCAGAGGACUGAAAAUCCUCGUGUCACCAGUUCAAAUC---UGGUUCCUGGCA
TRNA6  GGGGCCUUAGCUCAGC.UGG...G.AGAGCGCCUGCUUUGCACGCAGGAGGUCAGCGGU-CGA-CCCGCUAGGCUCCACCA
TRNA7  GGGCACAUGGCGCAGU.UGG...U.AGCGCGCUUCCCUUGCAAGGAAGAGGUCAUCGGUUCGAUUCCGGUUG---CGUCCA
TRNA8  GGGCCCGUGGCCUAGU.CUGga.U.ACGGCACCGGCCUUCUAAGCCGGGGAUCGGGGGUUCAAAUCCCUCCGGGU---CCG
TRNA9  CGGCACGUAGCGCAGCcUGG...U.AGCGCACCGUCCUGGGGUUGCGGGGGUCGGAGGUUCAAAUCCUCUCGUGCCGACCA
TRNA10 UCCGUCGUAGUCUAGG.UGGu..U.AGGAUACUCGGCUUUCAC-CCGAGAGACCCGGGUUCAAGUCCCGGCGACGGAACCA
```

Here, hyphens indicate deletes while lower-case letters, and the corresponding periods ('.') indicate inserts. The column numbers refer to match states in the model, not to column numbers, so that insertions are disregarded in calculating these index points. It is important to remember that insertions are not aligned among them selves: the fact that two insertion characters are in the same printed column only means that they were generated by the same insertion state, not that they should be aligned.

3.3 Examining models

Model structure can be quite interesting. The `drawmodel` program generates postscript drawings of models that include match-state histograms and transition line styles that correspond to their frequency of use. These drawings are most useful when derived from frequency counts, values that can be optionally included in the output file:

```
buildmodel test -train trna10.seq -seed 0 -print_frequencies 1
```

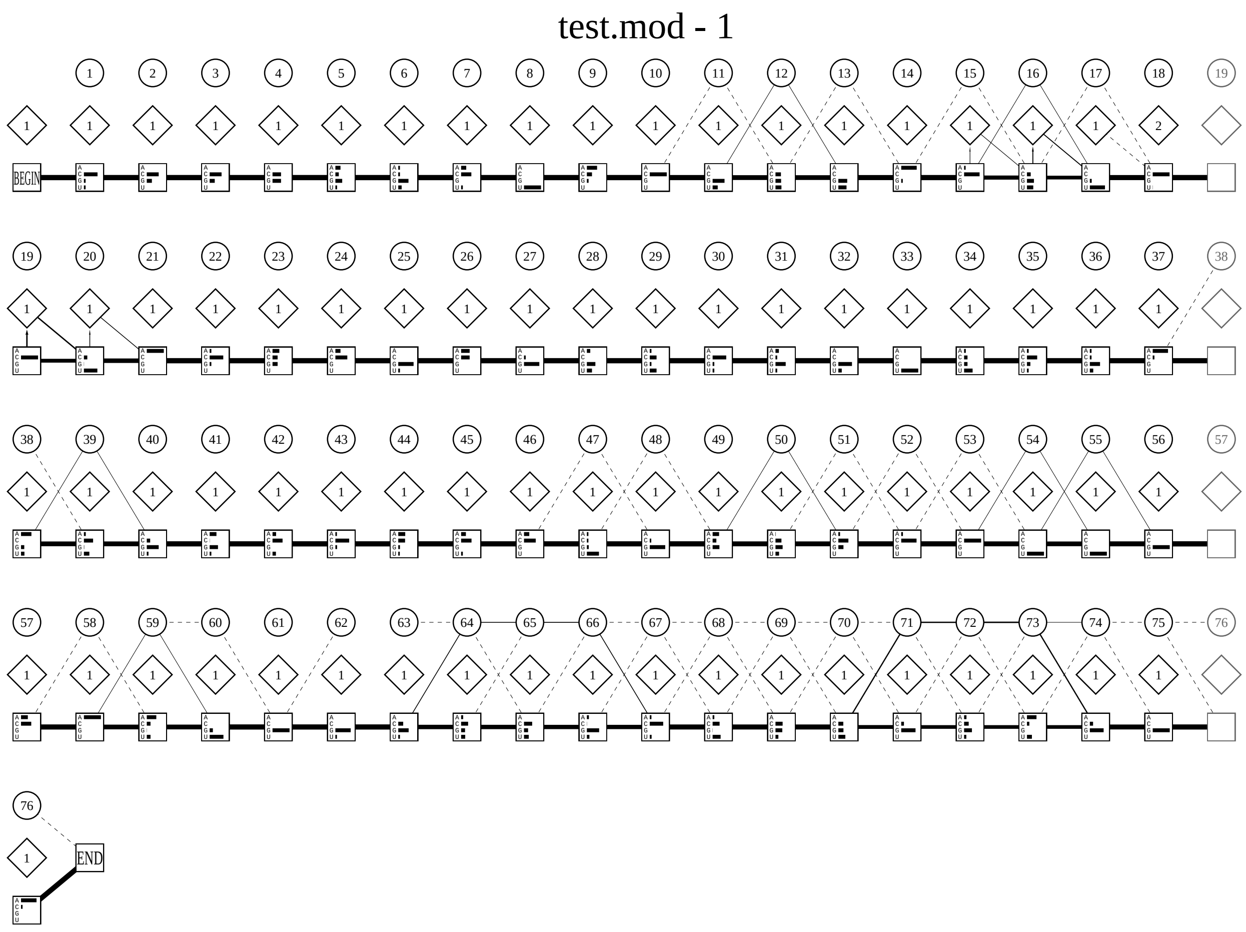


Figure 3: Output of `drawmodel`. Match state histograms are in AGCU order.

The command `drawmodel test.mod test.ps` will run the drawmodel program, which scans the file finding a model and frequency count data. By selecting the frequency count data in 'overall' mode, the postscript drawing in Figure 3 is generated.[1] The histograms in the match states correspond to the columns of the multiple alignment above, the numbers in the diamond insert states correspond to the average number of insertions for each sequence that uses that state, and the node numbers are given in the circular delete states. Transitions that are not used by a significant number of the sequences are not drawn.

`Drawmodel` is explained in more detail in Section 9.5.1.

3.4 Scoring sequences

The scoring program, `hmmscore`, generates a file of the negative log-likelihood minus NULL model (NLL-NULL, or log-odds) scores for each sequence given the model. Let's see how the 10 tRNA sequences fit the model (`test.mod`):

```
hmmscore test -i test.mod -db trna10.seq -sw 2
```

The arguments are the name of the run, the model file, the database sequence file, and a specifier to use fully-local scoring similar to that of the Smith & Waterman algorithm (this is suggested for all scoring runs unless semilocal or global scoring is specifically required).

`hmmscore` produces the `test.dist` file:

[1]Unfortunately, `drawmodel` does not generate postscript bounding boxes. For insertion in this LaTeX file using the `psfig` macros, `ghostview` was used to determine the bounding box.

```
%  SAM: ../src/hmmscore v2.1.1  (Apr 24, 1998) compiled 04/27/98_12:32:02.
%  SAM:  Sequence Alignment and Modeling Software System
%  (c) 1992-1998 Regents of the University of California, Santa Cruz
%  http://www.cse.ucsc.edu/research/compbio/sam.html
%
% ------ Citations (HMMs, SAM) ------
% A. Krogh et al., Hidden Markov models in computational biology:
%   Applications to protein modeling, JMB 235:1501-1531, Feb 1994.
% R. Hughey, A. Krogh, Hidden Markov models for sequence analysis:
%   Extension and analysis of the basic method, CABIOS 12:95-107, 1996.
% ----------------------
%  Run start: Mon Apr 27 12:50:01 1998
%  Run name:  test
%  On host:   alpha
%  In dir:    /auto/projects/compbio3/samtmp/sam/SAMBUILD/alpha/demos
%  By user:   rph
% ------------------------------------------------------------
% Inserted Files:  test.mod
% Database Files:  /projects/compbio3/samtmp/sam/demos/trna10.seq
%
% FIMs automatically added (auto_fim = 1).
% Subsequence-submodel (local) (SW = 2).
% S&W simple NULL scores adjusted by ln(seqlen) (adjust_score=2).
% 10 sequences with 747 residues.
% This run used EM scoring.
% The model has 77 positions.
%
% Using total residues as number of starting possibilities,
%  0.01 significance at <= ln(0.01)-ln(747)= -4.6 -  6.6 = -11.2
% Adjusting for model length gives 10 starting points,
%  0.01 significance at <= ln(0.01)-ln(10) = -4.6 -  2.3 = -6.9
% If entire sequences are modeled (i.e., no FIMs),
%  0.01 significance at <= ln(0.01)-ln(10) = -4.6 -  2.3 = -6.9
% Values for 10.0 significance are  -4.3,  -0.0, and  -0.0.
% Significance level is higher for multiple scoring runs.
% Sequence scores selected :  All  (select_score=8)
%
% Column 1: NLL-NULL using simple FIM (node 0) insert probabilities
% Column 2: the raw NLL score
% Scores sorted by column 1, best first
%
% Sequence ID   Length      Simple         Raw      X count
TRNA7              73       -36.52       58.72
TRNA3              76       -35.59       64.57
TRNA8              75       -34.61       59.74
TRNA2              76       -33.78       63.24
TRNA9              77       -33.54       63.72
TRNA1              72       -33.33       60.42
TRNA5              73       -32.77       67.51
TRNA6              74       -32.76       61.07
TRNA4              75       -32.51       67.67
TRNA10             76       -31.78       69.03
```

The score file contains five columns. The first is the sequence identifier, followed by sequence length, the 'NLL-NULL' score using a simple null model, and the raw negative log-likelihood score. The simple null model is just the product of the character probabilities in each sequence multiplied together. Thus, the NLL-NULL score show how much improvement modeling with an HMM gained versus modeling with just a single insertion state of from the HMM. The comments also indicate, for two significance levels and three possible counting assumptions, what score indicates a match to the model. Because all the sequences are tRNAs, all score well below any of the six significance level thresholds.

Other options of `hmmscore` enable the selective output of sequences according to their NLL scores, NLL per base scores, or Z-scores. The `hmmscore` program enters an interactive mode when called with no command line arguments. See Section 9.2 on page 56.

3.5 Discrimination versus alignment

Hidden Markov models can be used for both database discrimination and alignment of families. To obtain the best performance in each of these tasks may require the construction of different models, one for discrimination and one for alignment. While SAM's default parameters perform reasonably well in both cases, further research needs to be performed to enhance HMMs in both abilities. Possibilities can include different Dirichlet mixture prior and transition regularization (Section 7.1), different training algorithms (EM versus Viterbi), and different insertion table. In any case, sequence weighting, either internal or external, is highly recommended (Section 8.4).

4 File types

SAM produces a variety of files. This section summarizes both the extensions SAM uses and the extensions we conventionally use when naming output that goes to standard output.

The following file extensions are automatically used:

.mod A model file created by a `buildmodel`, `addfims`, or `modelfromalign`. When `many_files` is not set (the default), `buildmodel` places the run statistics, parameter settings, and the frequency model into this file as well. See Section 8.3 on page 42.

- **.f.mod** A subfamily model for family number 'f'. See Section 8.4 on page 44.
- **.a.mrrr.mod** If `print_all_models` is set, model number m during reestimate cycle 'rrr' will be placed in the file.
- **.s.rrr.mod** If `print_surg_models` is set, the winning model is printed after each surgery iteration, where 'rrr' is the reestimation index.

.stat The statistics of a `buildmodel` run when `many_files` is set, including reestimation scores and initial parameters. See Section 8.3 on page 42.

.freq A frequency model generated by `buildmodel` when `many_files` is set. This is a model that reports the frequency of each letter in each state when the training set is evaluated according to the model. See Section 8.3 on page 42.

.dist A scoring file listing sequence identifiers, lengths, and scores, produced by `hmmscore`. See Section 9.2 on page 56.

.sel Sequences that passed the selection criteria used in `hmmscore`. See Section 9.2 on page 56.

.smooth The curve used for Z-scoring, produce by `hmmscore`. See Section 9.2 on page 56.

.a2m A SAM alignment file, as created by `align2model` or `hmmscore`. A FASTA-compatible format. See Section 9.1 on page 50.

.weightoutput A list of sequence weights from the internal weighting option of `buildmodel`. Present when `print_all_weights` and internal weighting are both enabled. See Section 8.4.3 on page 47.

.samrc A file of default parameters either in a home directory or the current directory. See Section 5 on page 21.

.mult One of two multiple domain alignment output files created by `hmmscore`. This will contain all alignments to a motif that were found. See Section 9.2.5 on page 66.

.mstat The other multiple domain output file. This file contains the sequence identifier and scores for the data in the corresponding `.mult` file. See Section 9.2.5 on page 66.

.seq A file of sequences, as for example created by `sampleseqs`.

.plt A `gnuplot` command file created by `makehist`. See Section 9.6 on page 78.

.data A `gnuplot` data file generated by `makehist`. See Section 9.6 on page 78.

.hmmer An HMMER-format model produced by `sam2hmmer`. See Section 9.5.4 on page 76.

The following file extensions are conventionally used in this manual.

.ncomp A Dirichlet mixture prior library file, where n is the number of components to the mixture, as in `null.1comp` or `mall-opt.9comp`. See Section 7.1 on page 26.

.pretty The output (stdout) from `prettyalign`. See Section 9.1 on page 50.

.plib A Dirichlet mixture prior library file (old naming convention). The same extension is used both for match state regularizers and HSSP-based transition regularizers. See Section 7.1 on page 26.

.regularizer A transition regularizer, generally used in conjunction with a Dirichlet mixture regularizer for the match states. See Section 7.1 on page 26.

.weights A file of sequence weights. See Section 8.4 on page 44.

In most cases, SAM can read compressed (.gz or .Z) files, specified either as their complete name or as their root name without the compression suffix. If the root name is given and both a compressed and an uncompressed file exist, the uncompressed file is read. If both a .gz and a .Z file is present, the .gz file is used.

5 Parameter specification

Parameter values are drawn from four sources: command line arguments, inserted parameter files, default parameter files, and the program itself. Initial models and regularizers cannot be specified on the command line.

Each parameter, including the initial model and regularizer, has a reasonable setting hardwired in the SAM code. These are the default values listed in Section 11. The default regularizer is actually two defaults, one for RNA or DNA, and the other for proteins.

These hardwired values can be overridden by a user-specific default file or command line specification. This file can be one of three alternatives. First, if the environment variable SAMRC is set, new default values are read from that file. Second, if the SAMRC variable was not set and a `.samrc` file exists in the current directory, that file is used as the default. Third, if SAMRC was not set and `.samrc` was not found in the current directory, `$HOME/.samrc` is checked.

Parameter files can cause other parameter files to be read using the `insert` directive. When this directive is used in a default such as `.samrc`, the inserted file is assumed to have defaults as well. Non-default files are specified on the command line as, for example,

```
buildmodel test -alphabet RNA -insert trna.init
```

In this case, the alphabet is set to RNA, and the file `trna.init` will override default parameters hardwired in the program or specified in one of the `.samrc` files. If the file contained, for example, the line `alphabet DNA`, the alphabet would be switched to DNA with an appropriate warning message. Values are set and insert files are read according to their position on the command line or within a file.

Command line arguments are evaluated in the order they are presented to the program. If one of the command line arguments specifies an inserted parameter file, that file is processed before the next command line argument. If one file inserts another, the inserted file is processed before completing the original file. Thus, to override values specified in an inserted file, insert the file *first* on the command line, and then specify the parameters to reset: the last specified values win.

It is often important to conditionally specify initialization information. In addition to the `insert`, three conditional insertion directives are also available: `insert_file_dna`, `insert_file_rna`, and `insert_file_protein`. These cause a file to be inserted if the alphabet matches the directive. If the alphabet is not yet set when one of these is encountered, and warning message is generated.

Two parameter names have abbreviated forms: `i` can be used in place of `insert`, and `a` can be used in place of `alphabet`. The following will set the alphabet to RNA and read in the file named `parameters`.

```
buildmodel test -a RNA -i parameters
```

The model output (such as `test.mod`, in the command line above) includes statistics about the run and a listing of all parameters that have been changed from their default values. Inserted file names are listed, but commented out, because their effect has been recorded in the list of all changed parameter values. Random number seeds created based on the pid are also commented out so that new seeds will be created if the program is rerun on the file.

Models are usually specified using the insert file (`-i`) command line argument. In this case, the model type (i.e., model, regularizer, frequency counts, or null model, discussed in Section 7.4) is read from the file. Alternatively, a `model_file`, `regularizer_file`, or `nullmodel_file` can be specified, in which case the very first model structure in that file (which could be a regularizer or frequency count model, for example) is read in. These file names will override any models present in the inserted files, even if the inserted file occurs after the `model_file` parameter on the command line. This option is particularly useful for discrimination training with positive and negative examples, in which case a model generated by the negative examples can be used as the null model. See Section 9.2.1 on page 59.

There are two special paramater names, `db` and `id`, that form lists of strings. That is, when multiple database or sequence identifiers are found on the command line or in a paramater file, they are added to the current list of databases or sequence indentifiers, rather than replacing the previously-specified value.

6 Sequence formats

The SAM system understands several alphabets and many sequence formats.

6.1 Alphabets

The SAM system currently supports three predefined alphabets: 'DNA', 'RNA', and 'protein', as well as user-defined alphabets of up to 25 letters. The predefined alphabets can be specified by setting the `alphabet` variable. If no alphabet is chosen, the first sequence in a specified file will be examined using `readseq` (discussed below) to determine the alphabet. If this method does not work, the protein alphabet is the default. The SAM software includes several warning messages if it appears that an incorrect alphabet has been chosen.

The alphabets use standard characters. DNA sequences are composed of the characters "AGC-TRYN" and RNA of "AGCURYN," where 'R' is a purine ('G' or 'A'), 'Y' is a pyrimidine ('C,' or 'T' or 'U,' as appropriate), and 'N' is a wildcard character that could be any of the four normal characters. SAM's sequence I/O routines can convert between DNA and RNA alphabets if the alphabet is specified incorrectly.

The protein alphabet is "ACDEFGHIKLMNPQRSTVWYBZX." In addition to the twenty amino acids, 'X' is the general wildcard character, 'B' matches 'N' or 'D', and 'Z' matches 'Q' or 'E.'

In all alphabets, unknown characters are converted to wildcards and a warning message is printed.

When a model is created, a wildcard character's probability is the sum of the probabilities of the component letters. Thus, the 'X' character will have unity probability, giving it no preference to one state over another. During the training process, wildcard character frequency counts are proportioned among the appropriate true characters according to the relative probabilities of those characters.

6.1.1 User-defined alphabets

SAM also supports user-defined alphabets of 2 to 25 user-selected letters ('A'–'Z') and one (required) wildcard letter. The restriction to alphabetic characters is a result of the need for both uppercase and lowercase letters in the sequence alignment format. As the system always requires an all-matching wildcard, only 25 letters are allowed.

User-defined alphabets are specified with the `alphabet_def` variable. As with the standard alphabets, the definition will be included in all resulting models, so future specification of the alphabet on the command line is not required.

For example, performing the commands

```
buildmodel text -train text.seq -alphabetdef "text QWERTYUIOPASDFGHJKLZCVBNMX"
align2model text -i text.mod -db text.seq
```

results in the alignment file:

```
>sentence1, 47 bases, 687E946B checksum.
THEQUICKBRoW...NFOXJUMPEDOVERTHESLOWL.AZYDOG...
>sentence1, 47 bases, 687E946B checksum.
THEQUICKBRoW...NFOXJUMPEDOVERTHESLOWL.AZYDOG...
>sentence2, 47 bases, EC040EB7 checksum.
THEQUICKER.GreeNFOXHOPPEDOVERTHESLOWLuCKYPIG...
>sentence3, 47 bases, CB5CB7A1 checksum.
THESLOWLAZ.Y...PIGWADDLEDINTOTHEQUICK.PURPLEfox
>sentence4, 47 bases, EBB0DD62 checksum.
THEFASTBRO.W...NFOXHOPPEDINTOTHEQUICKlAZYDOG...
```

Note that the above example does not model the letter 'X' because it is a wildcard: the 'X' character was not trained and does not have a preference for any state over any other state.

A minimum of three characters, 2 normal and one wildcard, is required to define an alphabet. Default flat regularizers are created automatically, but users may wish to create their own alphabet-specific regularizers with `regularizer_file`.

As with alphabets, models are tagged with the `alphabet_def` line, for example

```
MODEL --  Final model for run text
alphabet_def text QWERTYUIOPASDFGHJKLZCVBNMx
GENERIC
1.886984 0.254944 0.376488
.....
```

See Section 7.4 on page 31.

6.2 Sequences

SAM uses a modified version of the `readseq` package D. G. Gilbert of the Indiana University. The code is based on the February 1, 1993 release, and is included as a subdirectory of the SAM source directory. We are grateful that Gilbert has provided this useful package that may be used by anyone.

The `readseq` package can read most common formats: examples of all these formats are included in the `readseq` directory. The formats include:

- IG/Stanford, used by Intelligenetics and others
- GenBank/GB, genbank flatfile format
- NBRF format
- EMBL, EMBL flatfile format
- GCG, single sequence format of GCG software
- DNAStrider, for common Mac program
- Fitch format, limited use
- Pearson/Fasta, a common format used by Fasta programs and others
- Zuker format, limited use. Input only.
- Olsen, format printed by Olsen VMS sequence editor. Input only.
- Phylip3.2, sequential format for Phylip programs
- Plain/Raw, sequence data only (no name, document, numbering)
- MSF multi sequence format used by GCG software
- PAUP's multiple sequence (NEXUS) format
- PIR/CODATA format used by PIR

We often use the IG/Stanford format, which looks like this:

```
; All lines beginning with a ‘;’ are comments
; Now follows the identifier for sequence 1, and the sequence,
; and the digit ’1’
SEQ1
LMLDQQTINI IKATVPVLKE HGVTITTTFY KNLFAKHPEV
RPLFDMGRQE SLEQPKALAM T1
;
;
SEQ2       - Comments after the identifier are ignored
AKHPEVRPLFDMGRQESLEQPKALAMT1
```

The IG/Stanford format has been slightly changed from the `readseq` package's default: a semicolon starting a line will end a sequence whether or not the number '1' or the number '2' occurred at the end of the previous line.

For information on other formats, please look through the test files and the `Formats` file in the `readseq` directory.

Sequence output by `hmmscore` will be in whatever format the last input file had.

Sequence output by `align2model` and `hmmscore` is in a FASTA-compatible format in which uppercase letters indicate match states and lowercase letters indicate insertion states and hyphens indicate deletion states (model positions for which the given sequence has no corresponding character). The prettyalign program can be used to line up the match columns of an a2m-format alignment file.

Additionally, `align2model` can includes periods so that its sequence outputs can be visually aligned without the use of `prettyalign`. If, for example, the longest sequence in a collection is 2000 characters long, all sequences will be filled (using periods) to that longest sequence's alignment length, which will be more than 2000 if any deletion states are used. Thus, allowing the periods to be printed can greatly expand the size of the alignment file. If periods are not desired, the paramater `a2mdots` can be set to 0. The `prettyalign` program will work whether or not the `a2m` format alignment has periods.

SAM can also read HSSP files.

6.3 Training and test sets

The `buildmodel` program uses two sets of sequences: the training and the test set. Training is performed exclusively on the training set, and at the end of the model creation, all sequences in the test set are checked against the model, and the average NLL distance is reported for both the training and the test set.

Training and test sets can be specified in up to two files each: `train, train2, test,` and `test2`. At most `Nseq` sequences will be read from any one file, so that at most `4Nseq` sequences will be read in if four files are specified.

The system can also randomly partition sequences into the training and the test set. If `Ntrain` is set, the system will randomly pick `Ntrain` sequences from all files specified (training and testing) using the random seed `trainseed`, and reserve the rest for the test set. By default, the seed is set to the process ID number, which is printed on the output file so that the partition can be reproduced. Sequence partitioning and model training use different random seeds, though both default to the process ID.

Several other programs, such as `hmmscore` and `align2model`, take an arbitrary number of sequence database files specified as `db`. Unlike most variables, repeating the `db` declaration adds a new file to the list, rather than replacing the previous database file.

7 Regularizers and models

The SAM system handles a type of hidden Markov model that was developed specifically for biological sequences. It consists of a chain of 'nodes', each of which consists of a 'match' state, an 'insert' state, and a 'delete' state (Figure 1 on page 12). The only way the structure can be varied is in the length, *i.e.*, how many nodes the model has. There are three transitions out of each state, which can be taken with some probability. One of these transitions leads to the insert state in the same node, whereas the others lead to the match and delete states in the next node. Two states are special: the begin state (numbered 0) and the end state (numbered $L + 1$ for a model of length L). The model

A	0.08713	C	0.03347	D	0.04687	E	0.04953	F	0.03977
G	0.08861	H	0.03362	I	0.03689	K	0.08048	L	0.08536
M	0.01475	N	0.04043	P	0.05068	Q	0.03826	R	0.04090
S	0.06958	T	0.05854	V	0.06472	W	0.01049	Y	0.02992

Table 1: Default amino acid match-state frequencies for protein regularizer.

is completely specified when all the probabilities are given for all transitions and all the letters in the match and insert states.

7.1 Regularizers

The word *regularizer* is often used in (Bayesian) estimation for a method to keep estimates from over-fitting the data, and in Bayesian statistics it is tightly connected with the so-called *prior* distribution. We use a Bayesian method of model estimation, and we have chosen to let the regularizer play several important roles in the program. The regularizer should reflect your prior expectations about how a model will look like for the family you are about to model. For instance, one may not think a model that only uses inserts and deletes is a good one, and that expectation can be built into the regularizer.

The regularizer has three functions:

Regularizer: During model estimation the regularizer should make sure that the model does not diverge too much from your expectations. It is done by adding 'fake' observations to the real ones. The model is reestimated by 'counting' how many times each probability parameter is used by the data, and then normalizing these counts. Before the normalization the 'fake' counts in the regularizer are added.

Initial model: By normalizing the regularizer, a valid model is obtained. If no initial model is specified to the program it will use this normalized regularizer as a starting point, but usually some noise is added first (see below).

Noise: The normalized regularizer also determines the noise added both initially and during learning (if annealing is used). See below.

Therefore, to run the program, you always have to specify a regularizer. Some good default ones are shipped with the program, so you need not worry about it in the beginning. With some experience however, it can be used as a powerfull tool for guiding the learning process.

The default RNA and DNA regularizer assumes a uniform distribution over the four letters, while the default protein regularizer uses the amino acid background frequencies in Table 1. Often it is a good idea to use the actual letter frequencies in the training data instead of the default distribution. This can be achived by setting `Insert_method_train` to 1.

Without editing all the numbers in the regularizer, one can change the strength of it by changing some parameters called 'confidences'. All regularizer numbers corresponding to transitions leaving the delete states are multiplied by the parameter `del_jump_conf` before being used. Similarly for the parameters `ins_jump_conf` and `match_jump_conf`. The numbers corresponding to the letter probabilities in match states and delete states are multiplied by `matchconf` and `insconf` respectively.

7.1.1 Regularizer alternatives

For training protein sequences, we always recommend the use of a Dirichlet mixture prior, which is enabled by setting `prior_library` to the name of a prior library. The prior library (discussed in the Sjölander et. al. paper mentioned in Section 1) encapsulates information about what distributions are expected to be found in match states. That is, columns in a multiple alignment are not all drawn from the same background distribution: some columns are highly conserved, others are primarily hydrophobic, and so on. The SAM distribution includes both the mixture from the CABIOS paper as well as several other prior libraries created by Kevin Karplus (karplus@cse.ucsc.edu):

uprior9.plib The 9-component library discussed in the aforementioned paper. Optimized for unweighted blocks data.

mall-opt.9comp Library re-optimized for unweighted data from an HSSP subset.

opt-weight1.9comp Library reoptimized for weighted version of same HSSP subset.

recode1.20comp A 20-component Dirichlet mixture trained on (realigned) HSSP alignments that have a diverse set of sequences. Intended for use in recoding inputs to neural net, but also useful as a standard regularizer.

null.1comp A one-component regularizer with tiny alpha, to get effectively no regularization.

The 20-component library is Kevin Karplus' current favorite for general use.

The `insert_protein` command, in conjunction with another file that specifies the prior library, can be used to ensure that a prior library is used whenever protein analysis is performed.

The distributions of nucleic acids do not lend themselves to effective use of Dirichlet mixture priors.

When a prior library is used, it overrides the match-state character emission values of the regularizer. Similarly, the insert-state character emission values of the regularizer are by default overridden to be the geometric average of the match state probabilities. Thus, as a result of the historical development of this code, for protein sequence analysis, only the transistion probabilities of the regularizer are actually used in training. Again, the distribution contains several different transistion regularizers optimized for different purposes, all created by Kevin Karplus. With Version 2.0, the default protein transition regularizer has been changed to trained.regularizer, good general regularizer. The old values are in the `sam1.3.regularizer` file of the `lib` directory.

trained.regularizer Regularizer optimized for unweighted transition counts on some set of re-estimated HSSP alignments

cheap_gap.regularizer Makes gap opening and closing very cheap allowing exploration of many different alignments, but giving too high a cost to long matches

long_match.regularizer Assigns somewhat reasonable gap costs for unweighted data, useful for sweeping away "chatter" into big matches and big gaps, by making gap opening expensive but gap extension fairly cheap.

We intend to futher evaluate nucleotide regularizers in the future.

Prior libraries and regularizers can be specified by their path name. If the `$PRIOR_LIBRARY` environment variable is set to a path name including a trailing '/' (or if it was not set but the proper directory was specified at compile time), SAM will check that directory for prior libraries and regularizers.

Mixtures and regularizers make the biggest difference for small training sets. The file `globins50.seq` contains 50 globins. To test this, generate and score two models from four sequences:

```
buildmodel train4 -train globins50.seq -seed 0 -trainseed 0 -ntrain 4
           -priorlibrary 0
buildmodel train4reg -train globins50.seq
           -priorlibrary recode1.20comp
           -regularizerfile weak-gap.regularizer
           -seed 0 -trainseed 0 -ntrain 4
hmmscore train4  -i train4.mod  -db globins50.seq -sw 2
hmmscore train4reg -i train4reg.mod -db globins50.seq -sw 2
```

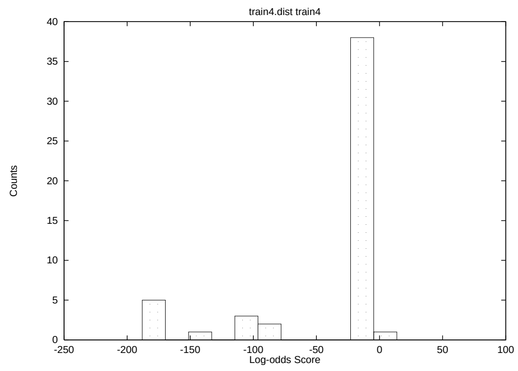
Note here that the four training sequences are also in the test set.

The results of these two runs, as well as two similar ones with 10 training sequences, are shown in in Figure 4, in the form of score histograms. See Section 9.6.1 on page 78. Note how in both cases, the use of the optimized regularizer improves scores of the globin sequences, and also that in both cases, the jump from 4 sequences to 10 sequences greatly improves model scores.

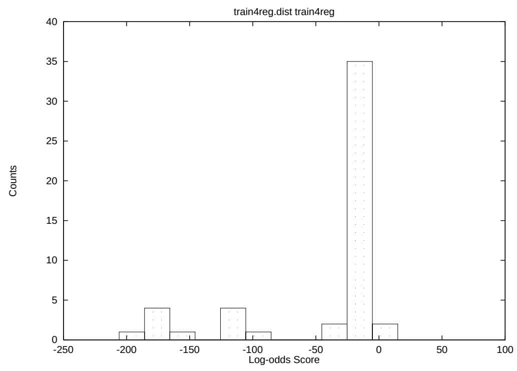
7.1.2 Transition regularizers with structural information

Just as the Dirichlet mixtures were used to incorporate prior information about amino acid distributions in the match states of an HMM, one can now use analogous information concerning the transition probabilities in various structural environments. To derive this information, we built HMMs from about 1050 HSSP database files and aligned the sequences that made up the file back to the HMM. Using sequence weighting and noting the structural environment, we generated weighted counts for transitions in every structural environment. Structural environment was defined in terms of secondary structure and accessibility. From these weighted counts we derive pseudocounts and incorporate them when building an HMM. The net effect of this is to impose general structural information, such as the low probability of an insert into the middle of a helix, into the HMM estimation process.

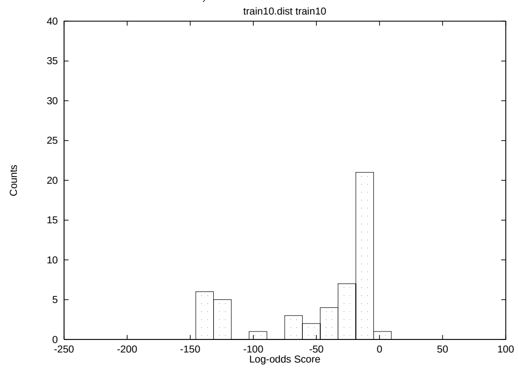
There are three relevant parameters. The first is the specification of the structural transition prior library, which one specifies with `trans_priors`. The library incorporated into the current SAM suite is TransFromRev15.plib. In order to use this library, one must specify a template file with `template`. This is a three-column file: animo acid sequence, secondary structure, and accessibility (as defined by HSSP). During model estimation, the sequence in the template file is aligned to the HMM. The alignment of the template sequence to the HMM dictates the assignment of the values in the second and third columns of the template file to each model node. These values in the last two columns



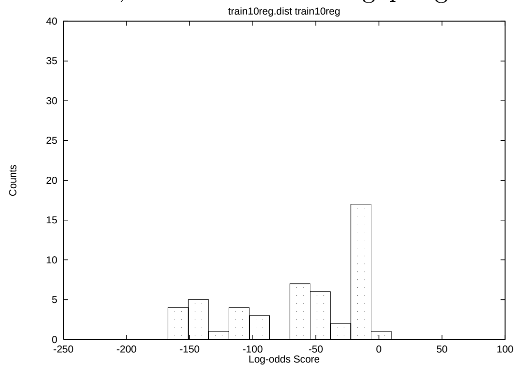
Ntrain = 4, no Dirichlet mixture



Ntrain = 4, Dirichlet and weak-gap regularizer



Ntrain = 10, no Dirichlet mixture



Ntrain = 10, Dirichlet and weak-gap regularizer

Figure 4: Regularizer performance

specify a structural environment, whose pseudocounts are used to reestimate the node's transition parameters. One may change the influence of pseudocounts with a real-valued multiplier using the parameter `transweight`.

The program `make_template` is included with the SAM distribution for generating template files from HSSP files.

The following is a command line example involving the use of the transition prior library TransFromRev15.plib, the template file for the PDB structure 2prd, and the weight multiplier.

```
    buildmodel 2prd -train 2prd.training.seqs -priorlibary recode1.20comp      -transpriors
    TransFromRev15.plib -template 2prd.tplate  -transweight 2.5
```

The 2prd template file 2prd.tplate was generated with the command line

```
    make_template 2prd -alignfile 2prd.hssp
```

and the first few lines are:

```
TEMPLATE
%
%
%       Template sequence from 2prd
%
%       File generated Mon Aug 11 19:20:51 1997
%
%
SEQLENGTH   174
ID  2prd
%
%
%AA    STRUCTURE    ACCESSIBILITY
%--   -----------   -------------
 A         *          128
 N         *          60
 L         G          30
 K         G          60
 S         G          61
 L         S          44
 P         *          88
 V         *          23
 G         *          19
 D         T          172
 K         T          115
 A         T          27
 P         T          11
 E         T          94
 V         E          27
 V         E          1
 H         E          53
 M         E          1
 V         E          0
 I         E          0
 E         E          12
 V         E          0
 P         *          18
```

This method of incorporating structural information into the transition probabilities of an HMM leaves open the option of experimentation with protein HMMs for which structure may only be predicted or assumed.

7.2 Initial model

Sometimes one would like to change a model that was already found earlier, and then restart build-model from that model. Thus, the initial model should not be made from the regularizer as described above. That is done by specifying the model explicitly in the init file, by using the heading 'MODEL' instead of 'REGULARIZER' that starts the regularizer specifications. See section 7.4, below. Most other programs in the SAM package also take already-formed models as input.

If desired, the first model in some other file (which might have a keyword other than 'MODEL') can be read using the `model_file` directive. See Section 5 on page 21.

7.3 Initial alignment

One of the best ways to train a hidden Markov model is to use an existing rough alignment to get the process started. There are two equivalent ways to do this. First, a model could be generated using the `modelfromalign` program (Section 9.4). Second, an alignment file can be specified on the `buildmodel` command line using the `alignfile` directive. In this case, any initial models are ignored in favor of this starting alignment.

The format of the alignment file is determined automatically as follows. First, if the key letters 'HSSP' begin the files first line, it is read in as an HSSP file. Second, the `align2model` format is checked. In this case, lowercase letters are treated as insertions, periods are ignored, and uppercase letters and hyphens refer to match columns. If all sequences do not have the same number of match columns under these asumptions, the sequences are checked for a general alignment format, in which all upper and lower case letters count as match columns, and all periods and hyphens count as deletions. If this fails as well, SAM will continue on using this last format, but will print error messages about the sequences with non-matching lengths.

The `alignment_weights` parameter can be used to weight the sequences in the initial alignment. See Section 8.4 on page 44.

As a subcase of an initial alignment, `buildmodel` can be instructed to create models from randomly chosen single sequences in the training set. This is done by setting the `sequence_models` to a value greater than 0. For each of the initial models required by `buildmodel`, a random sequence will be chosen and a model created based on that sequence regularized with a weight equal to the value of `sequence_models`. As long as fewer models are created than sequences in the training set, a different sequence will be chosen for each model. Noise will be reduced according to `retrain_noise_scale`.

7.4 Model format

Regularizers and models are specified by one set of numbers for each node in the structure. One can also specify a generic node for nodes not specified explicitly (internal nodes or the special Start and End states). The simplest model (for DNA) looks like this:

```
MODEL
alphabet DNA
Generic
dd md id
dm mm im
di mi ii
mA mG mC mT
iA iG iC iT
ENDMODEL
```

where `dd`, `md`, and `id` are numbers specifying probabilities of transitions INTO the delete state from delete, match and insert respectively. Similarly, `dm`, `mm`, and `im` are probabilities for the transitions INTO a match state and `di`, `mi`, and `ii` into insert. (The states come in the order: delete (`d`), match (`m`), and insert (`i`)). In Figure 1 on page 12, $T(m_3|d_2)$, for example, corresponds to the `dm` entry of node 3. The next four numbers (`mA`, `mG`, `mC`, `mT`) are the letter probabilities in the match state, *i.e.*, probabilities for producing the letters A, G, C, and T. Similarly the last four are the letter probabilities for the insert state. Here DNA was assumed; there would be 20 probabilities for proteins in each of the last two groups, in the alphabetical order of the single-letter amino acid abbreviations given in Section 6.1. Wildcards do not have corresponding entries in either the match or insert tables: their probabilities are calculated by SAM.

A model of length 4, in which all nodes are different, looks like this:

```
MODEL
alphabet DNA
0 0  0  0   0  0  0   0  mi ii   0  0  0  0    iA iG iC iT
1 0  md id  0  mm im  di mi ii   mA mG mC mT   iA iG iC iT
2 dd md id  dm mm im  di mi ii   mA mG mC mT   iA iG iC iT
3 dd md id  dm mm im  di mi ii   mA mG mC mT   iA iG iC iT
4 dd md id  dm mm im  di mi ii   mA mG mC mT   iA iG iC iT
5 0  0  0   dm mm im  0  0  0    mA mG mC mT   0  0  0  0
ENDMODEL
```

The first number in each line is the model position (the node number). Position 0 is the begin state, and position length+1 (5 in the example) is the end state.

In the two first positions (0 and 1) and the last (5) some probabilities are zero. These will always be set to zero by the program, whether or not a number different from zero is specified. Referring to Figure 1, the begin and end states look like match states, but really only match beginning-of-sequence and end-of-sequence, rather than real characters. In the case of position 0, initial insertions are allowed (the `mi` and `ii` transitions), as are transitions to the next position's match or delete states. Since position 0 has no delete state, the `dd`, `dm` transitions for position 1 are zero (the `di` transition is between d_1 and i_1 in the figure).

At position 5, all sequences are required to match the implicit end-of-sequence. Because the end position has no insert or delete states, all transitions into node 5's insert or delete state are zero.

The use of regularizers is discussed in section 7.1. A regularizer specification looks exactly the same as a model specification, except that it starts with 'REGULARIZER' instead of 'MODEL'. Frequency count output (`print_frequencies`) is similarly formatted, but with the starting word 'FREQUENCIES'. A trained model can be turned into a user-specified NULL model (see Section 9.2) by replacing 'MODEL' with 'NULLMODEL'. Any text on the same line as the initial word is ignored — `buildmodel` places a brief comment after the word 'MODEL'.

When specifying regularizers and models, it is sometimes convenient to specify the first and last node differently than the remainder. Since the length of the model can vary, the final node cannot be specified as being, for example, node 100. Instead, one can use *negative* numbers to specify nodes relative to the end, rather than the beginning. For example,

```
REGULARIZER
Generic .........
Begin   ......
1      ......
3      .......
-2      ......
-1      ......
End     .....
ENDMODEL
```

‘Begin’ (or anything beginning with ‘B’) is synonymous with node number 0, and ‘End’ with the end node. If this regularizer is used with a model of length 100, node number 0, 1, 3, 99(-2), 100(-1), and 101 (End) will be specified individually, and for all the rest of the nodes the Generic specification would be used.

The `buildmodel` program adds two informational nodes to models it produces. The first, called ‘LETTCOUNT’, has the distribution of characters in the set of training sequences. The letter counting procedure adds a small offset to avoid zero counts. Wildcard counts are proportioned among the appropriate letters according to the distribution of non-wildcard letters. The second, ’FREQAVE’, has the average frequency of each letter in the match states. If the match states are only modelling a portion of the training sequences, these averages may be different from the ‘LETTCOUNT’ values. These nodes can be used as null models during the scoring procedure, and during future buildmodel runs. See Section 9.2 on page 56.

It is often easiest to specify regularizers by changing an exisiting regularizer. For example, the default protein regularizer can be printed out to the model file by setting `dump_parameters` to 1.

```
buildmodel params -a protein -train trna10.seq -dump_parameters 1 -reestimates 0
```

This command writes all parameter values to the file `params.mod` using the protein alphabet (several alphabet warning messages will be printed because the sequences are not protein sequences). The last argument is required to ensure that `buildmodel` constructs a regularizer.

The `params.mod` file contains among other lines, the following regularizer specification (several digits have been truncated):

```
REGULARIZER:   Initial setting
alphabet protein
GENERIC 1.89 0.25 0.38 1.82 15.52 3.76 0.23 0.27 4.01
0.16 0.04 0.11 0.12 0.07 0.12 0.07 0.11 0.13 0.14
0.06 0.11 0.07 0.10 0.11 0.17 0.15 0.14 0.03 0.07
0.16 0.04 0.11 0.12 0.07 0.12 0.07 0.11 0.13 0.14
0.06 0.11 0.07 0.10 0.11 0.17 0.15 0.14 0.03 0.07
ENDMODEL
```

The numbers are in order, the transition probabilities, the 20 match state values, and the 20 insert state probabilities. The match an insert state values correspond to those in Table 1. Versions of SAM before Version 1.1 had a uniform distribution in the insert states, rather than a background distribution. To change to a uniform distribution for insert states, but maintaining the default transition regularization, the following could be placed in a parameter file (or the `insert_method_train` variable could be used, as discussed in Section 7.1):

```
REGULARIZER:  Background in match, 1/20 in insert
alphabet protein
GENERIC
1.886984 0.254944 0.376488
1.819972 15.521340 3.764209
0.225758 0.265967 4.006562
0.162339 0.037220 0.107508 0.123557 0.074544
0.122092 0.072662 0.112151 0.128548 0.138534
0.063912 0.113368 0.074824 0.103722 0.110612
0.170739 0.154307 0.143584 0.028017 0.069302
0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05
0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05
ENDMODEL
```

Alphabet (or `alphabet_def`) specification within a model or regularizer is optional, though whenever `buildmodel` prints one of these structures, the name of the alphabet will be included.

7.4.1 Model length

The length of the model(s) can be determined by the program in several ways (listed in order of importance):

- If there is an initial alignment (`alignfile`), the length of the alignment will be used.
- If there is an initial model of fixed length (no GENERIC) or `modellength` is 0, the initial model's length will be used.
- If there is an initial regularizer of fixed length (no GENERIC), the regularizer's length will be used.
- If the value of `modellength` is greater than 0, all models will be of that length.
- If instead `maxmodlen` is set to greater than 0, model lengths will be chosen randomly between `maxmodlen` and `minmodlen`.
- If `maxmodlength` is left at its default 0 and `modellength` is set to 0, and no initial model is specified, all model lengths will be set to the average length of the training sequences.
- If `maxmodlength` is left at its default 0 and `modellength` is left at its default value of -1, then model lengths will be randomly chosen between 90% and 110% of either the initial model length (if present) or the average training sequence length (if no initial model is present).

When model lengths are randomly selected, it is done with the same random number generator that creates model noise (distinct from the random number generator used to divide sequences into the training and the test set).

The surgery heuristic (Section 8.2) may lengthen or shorten a model.

7.4.2 Special nodes

There are several special node types that can be used to hand-tune a model. These are indicated with type declarations within the model description, such as:

```
TYPE 29 NO_SURGERY
TYPE 12 KEEP
TYPE 1  FIM
TYPE -1 F
```

Here, the two parameters are the node number (a negative node number indexes from the end of the model, as above) and a type. Type declaractions may appear anywhere within the model specification, and in any order. If more than one specification for a node appears, the last one is used. Only the first character of the type matters, and it must be one of 'N', 'K', 'T', 'A', or 'F'.

To tie a type declaration to a specific node (rather than a generic node), its number must match that of the node declaration. That is, if a model consists of node declarations for nodes 1...5, a '`TYPE 6 F`' statement will either generate an error if there is no generic node declaration, or create a FIM node number 6 using the generic node specification.

Model output from `buildmodel` is fully specified: the model will include begin and end nodes and a sequence of positively-numbered nodes. If you wish to change change node types of a node, you must specify the exact same positive node number as that node. Specifying, for example `TYPE -1 FIM`, will result in an error because there is no generic node, and node -1 has not been specified. To achieve this effect, you will need to add either a generic or a new node description (for node -1) in addition to the type statement. Alternatively (and preferably), you can use the `addfims` program or in some cases the `auto_fim` variable to add a FIM at the start and end of the model.

Type 'N' nodes are no-surgery nodes. During the surgery heuristic, these nodes will neither be deleted (if they are used by too few sequences) nor expanded (if their insert states are used by too many sequences). The parameters of the node will be trained as normal. No-surgery nodes are usually not used explicitly: they are a building block upon which keep and FIM nodes are based.

Type 'K' nodes are keep nodes. The match and insert probabilities of these states will not be trained, however their transition probabilities will. Keep nodes are also immune to surgery.

Type 'T' nodes are transition keep notes. The *outgoing* transitions associated with that node are not trained but the match and insert tables are. Note that the outgoing transitions for a node include the incoming transitions to that node's insert state as well as the incoming transitions to the next node's delete and match states. Transition keep nodes are immune to surgery.

Type 'A' nodes are all keep nodes. This has the attributes of both 'K' and 'T', though of course one could not specify both 'K' and 'T' because nodes can only have one type. The outgoing transitions and the character distributions are not trained. All keep nodes are immune to surgery. In general, if you do not wish to train a node (for example, it is a conserved region from an existing alignment that is known to be correct), the nodes of that region should be of type 'A'. The 'K' and 'T' nodes are for more specialized use, such as learning transition probabilities for an existing profile.

Keep nodes can be particularly useful when training from an existing model or alignment (using the `modelfromalign` program, Section 9.4). If a region is identified as being particularly important to preserve during training, its nodes can be identified as keep nodes (determining which node numbers correspond to the region can be done using `drawmodel` or `align2model`). For example, if node numbers 10–12 are identified as being 'correct' (and thus should not be trained), the lines

```
    TYPE 10 K
    TYPE 11 K
    TYPE 12 K
```

should be added between the MODEL and matching ENDMODEL statements. After training, the kept nodes may no longer be numbered 10–12 because of model surgery, however the nodes will still be part of the model, and will be identified as kept nodes in the model output. The program `modifymodel` can be used to change node types.

Type 'F' nodes are free-insertion modules (FIMs) which are discussed below.

If the program is being run without an initial model, node types are taken from the regularizer. If there is an initial model, types in the regularizer are ignored preference to any types present in the model.

7.4.3 Binary models

Files containing models can be written in either text (human readable) or binary format. You can recognize a binary model by the keyword BINARY which appears on the line directly after the model declaration. The advantages of using binary formatted models are decreased file size/disk space usage and faster model reading and writing. The disadvantage is that you can't read or modify your model files.

By selecting binary format, you can reduce the size of an 81-node model file from 16,240 bytes to 8726 bytes. A file containing 249 nodes shrinks from 116,218 bytes in text format to 51,647 bytes. We achieve this striking savings by taking advantage of the fact that nodes of a given type frequently have the same letter probabilities for the insert state.

Before a binary model is written to file, the program scans all the nodes of each type. If it finds the nodes all have the same letter probabilities for insert state, the data is stored in a table. The probabilities are considered 'identical' if their difference is less than .000002, therefore they are rounded to this magnitude in the binary model.

If you run buildmodel and use binary output you are likely to shave a few seconds from the program's run time.

When generating model files, use the command-line option `-binary_output 1` for binary format. The default for this option is currently set to 0 or off.

The program `hmmconvert` (Section 9.5.2) is available to switch models from one format to another. It will read your model, determine its format and then write the model to a new file in the opposite format.

7.5 Free-insertion modules

It is often desirable to be able to model motifs that occour in long sequences. This can be done by a little 'hack' in the HMMs. The idea is to have a model of the motif flanked by insert states with particular character distributions. The cost of aligning a sequence to such a model does not strongly depend on the position of the motif in the sequence, and thus it will pick up the piece of the sequence that fits the motif model the best.

These insert states are added to the model by the so-called free-insertion modules (FIMs), in which only the delete state and the insert states are used. All the transitions from the previous node go into the delete state, which is ensured by setting the other probabilities to zero. From the delete state there is a transition to the insert state with the probability set to one. In the insert state, character probabilities are set according to `FIM_method_train` (for `buildmodel`) or `FIM_method_score` (for `hmmscore`). Use of these parameters allows you to embody information particular to the problem domain and can significantly affect performance. Discussion on the their use appears in Section 9.2. From the delete and insert states there are transitions to the next node which have the same probabilities (delete to match and insert to match have the same probability, and so on). Note that the probabilities do not sum up to one properly in such a module. Since all sequences must pass through a FIM's delete node (excepting the case of when the Begin node is a FIM), a delete (or dash) will be present in any alignment to the model.

The FIM is also used as the null model during scoring. (Section 9.2)

These special nodes can be used to learn, align, or discriminate motifs that occur once per sequence. Typically, FIMs are used at the beginning and the end of a model to allow an arbitrary number of insertions at either end. For example, if a model for a motif has been learned from truncated sequences, adding FIMs to the resulting model will enable detection of that motif within longer sequences. Before trimming sequences by hand, one should try learning with FIMs, as in:

```
REGULARIZER
GENERIC .....
TYPE 0 FIM
TYPE -1 FIM
ENDMODEL
```

The begin node (number 0) can be used as a FIM as shown above. However, the end node has no insert state, so the FIM is put just before the end, which is specified as node -1. (See Section 9.3.1 for some important comments on adding FIMs to models without the `addfims` program, as in the regularizer example above.)

Given a sequence, an HMM will only identify one occurence of the domain or motif on which it was trained even if there are multiple copies. Multiple occurrences can be found with `multdomain`.

To train a model to find several (different) motifs, one can add FIMs at different positions in the model. For instance to model sequences with two motifs of lengths 5 and 10 one should specify the model as

```
REGULARIZER
GENERIC .....
TYPE 0 FIM
TYPE 6 FIM
TYPE -1 FIM
ENDMODEL
```

and set the model length equal to 17 (5+10+2FIMs).

If domains are clipped from an alignment, converted to a model using `modelfromalign`, and then to a FIM model using `addfims`, it is best to model several positions on either side of the domain to prevent the FIMs from eating up the ends of the domain.

The FIM state's insert table (or the generic node's, if the FIM is not fully specified) has the distribution over characters to be used.

The program `addfims` will add free insertion models to both ends of a model, and is further discussed in Section 9.3. Because the correct addition of a FIM requires changes in the transition probabilities to and from the FIM, it is recommended that users only add fims by hand for unspecified regularizers and models: those that only have a generic node and one or more type declarations. To add free insertion modules to the ends of an existing model, for training or for motif searching, be sure to use the `addfims` or `modifymodel` program, or that `auto_fim` is set to its default value of 1.

Sometimes, in alignment or training the model may not be using FIMs as much as desired when, for example, there is a reasonably strong probability of using a specific internal insert state. The probability of a FIM modeling n characters is by default the product of the insert table probabilities for those characters. If this probability is too low, meaning that sequences are not using the FIM enough, it can be adjusted with the `fimstrength` parameter. Changing this parameter from its default 1.0 to 2.0, for example, make use of the FIM twice as likely as before. The value of `fimstrength` is also applied to simple null models, and if less than zero, the absolute value of fimstrength is applied to both FIMs and normal insertion states.

7.6 FIM, insert and match tables

To aid in the manipulation of insert tables, SAM provides several options to globally change the regularizer's match and insert states, and the initial model if it is generated from the regularizer. These options are controlled by `FIM_method_train` and `insert_method_train`. The default is to use the residue counts of the training sequences in both the insert and FIM states, as well as the match states of the GENERIC node.

0 Use the tables present in the model.

1 The relative frequencies of residues in the training sequences (from the LETTCOUNT node or the training sequences).

2 The relative frequencies of residues in model match states (from the FREQAVE node).

3 Uniform (flat) probability over all residues.

5 Amino acid background frequencies over all proteins (from the Generic node).

6 Geometric average of the match state probabilities.

The match state frequency average is only available when an existing model (with a FREQAVE node) is being trained.

If the method number is negative, the change only occurs if a model is being created by the program. For example, if `buildmodel` is run with an initial model, a negative FIM or insert method will have no effect on the model.

If `insert_method_train` is set, and a generic node exists in the regularizer or model, then both the match and the insert states of that generic node are changed. This is important because it means that match states in the initial model (before training commences) will be based on, for example if `insert_method_train` is 1, the letter counts of the current training set, assuming that that there is no initial model or alignment.

If the `train_reset_inserts` variable is set, then after each reestimation cycle in the training process, the insertion and FIM tables will be set according its value: 0, 1, 2, 3, 5 as above, or 6 (the default) to set to the geometric average of the match table probabilities in the newly trained model. See Section 9.2.1 on page 59.

8 The `buildmodel` estimation process

A detailed discussion of the estimation process can be found in "Hidden Markov models in computational biology: Applications to protein modeling," mentioned in the Introduction. This section provides an overview of the mechanics of model estimation.

After the sequences have been divided into training and test sets, and the initial model or models have been created, `buildmodel` will iteratively train the model using expectation-maximiation (EM). For each iteration, a comment line (beginning with a percent sign '`%`') is written to the output file (e.g., `test.mod`) that includes the iteration number and the average NLL distance between the set of training sequences and the model. Iterations continue until either an iteration gains less improvement than the `stopcriterion` (and noise is less than 10% of its starting value) or `reestimates` iterations have been performed. When multiple models are being trained (but not multiple subfamily models, see Section 8.4.1.1), training on each model is stopped individually when that model reaches the `stopcriterion` (provided noise is less than one tenth its initial value).

If surgery and multiple initial models are used, one model is selected for the surgery procedure, which will attempt to prune and grow the model as appropriate. After each surgery procedure (up to `nsurgery`), the reestimation process is repeated. Once either the limit on the number of surgeries is reached, or the surgery parameters produce no model modifications, the training procedure is complete.

After the model has been trained, the NLL scores for the test set are computed and reported, and the final model is written to the output file. This model file may be used as an input file to further refine the model, perhaps by setting the `stopcriterion` to a smaller value.

8.1 Noise and annealing

It is possible to add noise to the initial model(s). By setting `initial_noise` to a positive number that amount of noise is added to a model in the beginning of the program. It serves the important purpose to make models differ, if the program runs many models simultaneously — each model will have a different noise added.

To try to avoid local minima, one can add noise to the models during their estimation, and decrease the noise level gradually in a technique similar to the general optimization method called simulated annealing. The initial level of the noise in this annealing process is called `anneal_noise`. If `anneal_noise` is greater than 0, annealing is performed. (If `initial_noise` is also given, that will determine the noise for the first iteration, and `anneal_noise` the noise in the following iterations.) During the estimation process the annealing noise is decreased by a speed determined by `anneal_length`. There are two ways it can be done:

Linearly: If `anneal_length` is greater than or equal to 1, the noise is decreased linearly to zero in `anneal_length` iterations by the formula

$$\text{noise} = \texttt{anneal\_noise}(1 - \text{number\_of\_iterations}/\texttt{anneal\_length})$$

Exponentially: If `anneal_length` is less than 1, the noise is decreased exponentially by multiplying the noise with `anneal_length` in each iteration, which gives the noise

$$\text{noise} = \texttt{anneal\_noise} * \texttt{anneal\_length}^{\text{number\_of\_iterations}}.$$

In the exponential schedule, noise injection is halted when the amount of noise reaches 10% of its initial value.

Once the noise level has been calculated, there are three possible ways noise can be added, as controlled by whether `randomize` is positive, negative, or zero.

positive A set of `randomize` random paths are calculated through the model according the the regularizer probabilities. Each of these sequences is weighted by the amount of noise. These sequences are added to the counts generated by the real frequencies, thus the noise setting is somewhat dependent on the number of sequences being trained.

negative A set of $-$`randomize` random paths are calculated through the model. With a weight of $-$noise/`randomize`, these counts are added to the *normalized* (probability, rather frequency) model, and than the model is renormalized. Thus, the noise generation is similar to that of the first case, but total noise added is independent of both the number of sequences and the `randomize` setting.

zero For each probability parameter in the model, a random number between 0 and the corresponding parameter in the normalized regularizer is found. This number is scaled by the level of the noise, given for instance by `initial_noise`, and added to the probability in the model. After doing this for the whole model, all the probabilities are normalized. For example, if the noise is a random number between 0 and 2, random pseudo counts corresponding to up to two sequences will be added to each transition and each match state. This is the fastest means of noise generation.

Note that the annealing schedules are *ad hoc*. Still, according to our experience even fast and crude annealing generally improves performance. By default, exponential noise at a ration of 0.8 is used with no `initial_noise`, an `anneal_noise` of 5, and a `randomize` setting of 50 (corresponding to 50 random sequences). These values were chosen experimentally (see the Hughey and Krogh paper mentioned in the introduction).

After a model has been created, adding too much noise to the model may eliminate all the trained information. Therefore, if an initial model or an initial alignment is specified, noise (`initial_noise` or `anneal_noise`) is reduced from the default setting by a factor of `retrain_noise_scale`, which has a default of 0.1. Thus, the effective noise during a retraining would be 0.5 rather than 5. The same is true of surgery iterations, discussed in the next section. In this case, the starting noise of the reestimation process after a surgery, whether or not an initial model is specified, is the `anneal_noise` scaled by `surgery_noise_scale` parameter, which also has a default of 0.1.

A second annealing option, one based on slowing increasing the weights of the sequences being trained is discussed in Section 8.4.2.

8.2 Surgery

It is often the case that during the course of learning, some match states in the model are used by few sequences, while some insertion states are used by many sequences. *Model Surgery* is a means of dynamically adjusting the model during training.

Surgery will be `nsurgery` times: a full reestimation process is performed including `reestimates` reestimations, or until the `stopcriterion` is reached. By default, as in the tRNA example above, surgery is performed up to two times.

The basic operation of surgery is to delete unused match states and to insert match states in place of over-used insert states (the special node types described in Section 7.4.2 are never subjected to surgery modification). In the default case, any match state used by less than one half of the sequences is removed, forcing those sequences to use an insert state or to significantly change their alignment to the model.[2] Similarly, any insert state used by more than half sequences is replaced with a number of match states approximating the average number of characters inserted by that insert state.

The surgery heuristic can be adjusted with one parameter or with three. In the first case, setting `mainline_cutoff` to a number other than the default 0.5 will indicate how much non-match, or main line, activity will be accepted. For example, a setting of 0.25 indicates that any match state used by less than one quarter of the sequences should be removed, while any insert state used by more than one quarter of the sequences should be expanded into a number of match states approximately equal to the average number of characters emitted by that state.

For finer tuning of the surgery process, the parameters `cutmatch`, `cutinsert`, and `fracinsert`, can be used. During surgery, any match state with a smaller portion of sequences than `cutmatch` is removed, and any insert state with a higher portion of sequences than `cutinsert` is replaced by the average number of characters emitted by that insert state multiplied by `fracinsert`. By default, `fracinsert` is 1.0, and `cutmatch` and `cutinsert` are both equal to `mainline_cutoff`.

[2]To be more precise, any node that has a frequency count of less than one half the number of sequences is removed.

These parameters can be set in ways that cause large amounts of surgery. For example, setting `cutmatch` to 0.5 and `cutinsert` to 0.25 will delete any match state used by fewer than half the sequences, and insert match states for any insert node used by greater than one quarter of the sequences. Typically, this will result in an oscillating model in several positions — those positions used by more than one quarter and less than one half of the sequences. Such excessive surgery can sometimes aid in forming a good model.

8.3 Training statistics

In addition to the trained model, a report of the training procedure is included in `buildmodel`'s output. The comment sections of this file for the training example in the introduction is reproduced below.

```
%  SAM: buildmodel v2.1.1  (Apr 24, 1998) compiled 04/27/98_12:31:54.
%  SAM:  Sequence Alignment and Modeling Software System
%  (c) 1992-1998 Regents of the University of California, Santa Cruz
%  http://www.cse.ucsc.edu/research/compbio/sam.html
%
% ------ Citations (HMMs, SAM) ------
% A. Krogh et al., Hidden Markov models in computational biology:
%   Applications to protein modeling, JMB 235:1501-1531, Feb 1994.
% R. Hughey, A. Krogh, Hidden Markov models for sequence analysis:
%   Extension and analysis of the basic method, CABIOS 12:95-107, 1996.
% ----------------------
%  Run start: Mon Apr 27 12:49:46 1998
%  Run name:  test
%  On host:   alpha
%  In dir:    /auto/projects/compbio3/samtmp/sam/SAMBUILD/alpha/demos
%  By user:   rph
% ------------------------------------------------------------
%  Regularizer FIM_method_train training letter frequencies (1).
%  Regularizer Insert_method_train training letter frequencies (1).
%  Model initial FIM_method_train training letter frequencies (1).
%  Model initial Insert_method_train training letter frequencies (1).
%  Generic, Insert, and FIM character tables dynamically reset to
%   train_reset_inserts geometric mean of match probabilities (6).
%  All models generated from regularizer.
%
%  Model lengths:    80  76  68
% Itera-   Average
% tion     distance
%     1   108.535   106.841   105.750
%     2   105.199   104.495   105.751
%     3   104.916   103.681   105.910
%     4   105.510   103.122   106.658
%     5   105.418   102.466   104.947
%     6   104.099   102.609   105.159
%     7   103.643   101.179   103.770
%     8   103.262   102.086   103.935
%     9   102.124    99.454   103.989
%    10   100.864    99.758   102.458
%    11   101.231    99.296   102.804
%    12   100.591    98.433   101.124
%    13   100.067    95.583    99.759
%    14    97.499    92.888    98.344
%    15    82.019    70.605    79.457
%    16    78.085    66.075    75.398
%    17    72.760    65.473    73.220
%    18    70.477    64.637    71.556
%    19    69.273    64.283    70.661
%    20    68.444    64.169    70.313
%    21    68.079    64.120    70.182
%    22    67.826    64.120    70.132
%    23    67.625    64.120    70.132
%    24    67.519    64.120    70.132
%    25    67.443    64.120    70.132
% Model 1 (counting from 0) wins!!
% -----------------------------------------------------------
%  TRAIN 10 sequences (average length 74).  Distance statistics...
%     Min:-32.21 (  4) Max:-26.93 (  3) Ave:-29.72 SampDev:  1.95
%     MinSDs:  -1.28  MaxSDs:   1.44
%
%  Total CPU time:  user   0: 0: 7    system   0: 0: 0
%  Finished at:  Mon Apr 27 12:49:54 1998
% -----------------------------------------------------------
%
% Parameters from command line and insert files:
%
% trainseed 8541
% seed 0
train /projects/compbio3/samtmp/sam/demos/trna10.seq
a RNA
print_frequencies 1
%
% -----------------------------------------------------------
% -----------------------------------------------------------
MODEL --  Final model for run test
```

Here, the initial information includes program version and run information. In this case, no initial models were specified, so `buildmodel` created 3 models (the default value of `Nmodels`) from the regularizer of the specified lengths. Next, all three models are trained, and the one with the best score is selected. In this case, the best model did not require any surgery, so the process was complete. If the best model did need surgery, that single model would be further refined. Next, statistics on the scores of the training sequences (and test sequences, if present) and CPU time are presented, followed by non-default parameter settings. The `seed` entry is commented out to prevent any future training iterations from reusing the old seed. Finally, the model, along with its generic, letter count, and frequency average nodes is printed. The model has not been included in the example.

If the `many_files` variable is set, then the results of `buildmodel` are broken up into three files: the `.stat` file contains the run statistics and parameter settings, the `.mod` file contains the final model, and the `.freq` file contains the frequency model if `print_frequencies` is set.

8.4 Weighted training

Beginning with Version 1.3, SAM is able to perform a variety of weighted training options, including support for multiple sub-family training. Sequence weighting is particularly important when, as normal, the sequence data given to SAM is biased toward some type or subfamily of sequences (for example, from those organisms that have been most studied). Prior to Version 2.0, the SAM software system did not include any internal sequence weighting schemes, but could use weights generated by some other program. Version 2.0 includes two internal weighting methods described below, the first of which is turned on by default when external weighting is not used.

8.4.1 External weighting

For all external sequence weighting options, a sequence weights file is specified with either the `sequence_weights` variable (for `buildmodel` training sets) or the `alignment_weights` variable (for `buildmodel` or `modelfromalign` initial alignments). In this file, any line starting with a percent sign (`%`) is ignored as a comment. The first non-comment line is presumed to be a description of the weights file, for example including the program that generated the sequence weights. The next non-comment line contains two integers, the number of weighted sequences and the number of weighted subfamilies. Remaining uncommented lines consist of a sequence identifier, white-space, and floating-point sequence weights, one per family. Weights can be positive, negative, or zero, and need not sum to one. If a sequence does not have a corresponding weight, its weight is set to 1.0 and a message is printed. If a weight does not have a corresponding sequence, a message is printed. Sequences and weights do not need to be in the same order within their respective files.

For plain sequence weighting, the number of families is set to 1, and each sequence is assigned a single weight in the `sequence_weights` file. During the reestimation cycle, the frequency counts for each sequence will be multiplied by its weight.

Sequence weighting is particularly important in database discrimination: without sequence weighting, the model may specialize to an over-represented subset of the sequences, meaning that family members that do not happen to be in that sub-family will receive low scores.

The file `gseg4.seq` contains the initial 70-character segments of each of 4 globins. The last three are quit similar.

```
;
BAHG$VITSP
mldqqtiniikatvpvlkehgvtitttfyknlfakhpevrplfdmgrqesleqpkalamtvlaaaqnien
;
GLB$APLJU
alsaadagllaqswapvfansdangasflvalftqfpesanffndfkgksladiqaspklrdvssrifar
;
GLB$APLKU
slsaaeadlvgkswapvyankdadganfllslfekfpnnanyfadfkgksiadikaspklrdvssriftr
;
GLB$APLLI
slsaaeadlagkswapvfanknangadflvalfekfpdsanffadfkgksvadikaspklrdvssriftr
;
```

When a model is trained on this file without weighted training, the model is overspecialized to the latter group of sequences, resulting in the following scores:

```
GLB$APLKU          70      -138.28        53.50
GLB$APLLI          70      -138.04        50.91
GLB$APLJU          70      -131.19        56.49
BAHG$VITSP         70       -97.16       101.29
```

The following simple weight file is an attempt to correct this bias:

```
% gseg4.weights
Weight file for the simple globin segment example.
% 4 sequences and 1 family
4 1
BAHG$VITSP  2.0
GLB$APLJU   0.66
GLB$APLKU   0.66
GLB$APLLI   0.66
```

Note that in this weight file, to make the results of the two examples comparable, the weights were made to sum to 4. The reason for this is that in addition to sequences, the regularizer (provided the various confidence parameters are non-zero) shapes the model. Setting all sequence weights uniformly high (e.g., 100.0) will have a similar effect to setting all the regularizer confidences to 0.

With the simple weight file, the following scores are produced.

```
BAHG$VITSP        70      -130.50       67.12
GLB$APLLI         70      -114.13       75.53
GLB$APLKU         70      -113.77       78.55
GLB$APLJU         70      -107.92       80.61
```

Here, with the three similar sequences weighted less, the model better matches (perhaps too much) the dissimilar sequence.

The above example is definitely a toy problem: weights must be set using statistically and biologically-valid means.

8.4.1.1 Multi-subfamily weighting

Warning: This feature is not completely available or completely debugged.

A particularly interesting use of weighting schemes is when a family of sequences can be divided into several subfamilies. This special type of training is used whenever the number of families in a weights file is greater than one.

In this case, SAM will train one model per family in parallel so that each model can specialize to its subfamily. Although this sounds just like training each subfamily separately, there is an important difference. During the regularization procedure, the counts across all subfamily models are taken into account when reestimating each subfamily's model. This means that, in the case of multiple alignments, a full-family multiple alignment can be generated by aligning each sequence to its appropriate subfamily model and combining the results. At the moment, subfamily modeling is not fully implemented and not recommended for use.

There are a few changes in the functionality of `buildmodel` when subfamily modeling is used. First, only a single suite of subfamily models is trained at a time, so multiple runs must be performed to match the functionality of starting with more than one model and selecting the best. Second, prior libraries must be used. Third, a more conservative approach to model surgery is taken. The subfamily models are always modified in parallel, and only if all the subfamily models agree on the surgery procedure (if all subfamily models believe inserting new model positions is appropriate, the minimum of all proposed insertion lengths is used). To encourage more surgery, users may wish to lower the surgery thresholds when training with multiple models. See Section 8.2 on page 41.

Also, model files are treated somewhat differently. The `many_files` option is always turned on. The subfamily models are writing to files named, for example, `runname.3.mod`, where the number indicates which subfamily (starting from zero) that model is for. It is possible to retrain a suite of subfamily models by setting `family_base_file` to the root name of the suite of models (i.e., `runname` in the above example).

The `hmmscore` program does not yet score against multiple models: to perform database search against a suite of models, `hmmscore` must be run independently for each subfamily, and then the results combined by, for example, classifying each sequence as a member of the subfamily with which it scored best.

8.4.2 Annealing with Weights

Sequence weights can also be an effective means of annealing (Section 8.1) during the training process. When using this option, the sequence weights are slowly increased over the first several reestimation cycles. Thus, at first, the sequences will have little effect on the model for the next training iteration: the regularizer and prior library will dominate, though particularly strong signals in the sequences, such as strongly conserved regions, will show throw. As the training continues, the sequence weight multiplier is brought up to its final value, giving full weight to the sequences.

The annealing schedule options are similar to that available with noise generation. The relevant parameters are `weight_length`, which indicates how long the annealing should last, and `weight_final`, indicating the final sequence weight multiplier (the default is 1.0). The sequence weight multiplier found be the formulas below is multiplier by the sequence weight (which is 1.0 if no weight file is used in training) to find each sequence's weight during the given reestimation iteration.

Linearly: If `weight_length` is greater than or equal to 1, the weight multiplier is increased linearly to `weight_final` in `anneal_length` iterations by the formula

$$\text{multiplier} = \texttt{weight\_final} * \text{number\_of\_iterations}/\texttt{weight\_length}$$

Exponentially: If `weight_length` is less than 1, the multiplier is increased exponentially until the multiplier reaches 90% of its final value as follows:

$$\text{multiplier} = \texttt{weight\_final} * (1.0 - \texttt{weight\_length}^{\text{number\_of\_iterations}}).$$

Sequence weights are never zero: the first reestimation cycle uses the first non-zero value of the weight formula.

8.4.3 Internal weighting options

Version 2.0 of SAM offers two methods of standalone weighting (i.e. without the extra steps for the file-based weighting). These methods are based entirely on the log-odds score of the sequence against the model being trained. Their invention was motivated by HMMer's maximum-discrimination training method.

Given a linear hidden Markov model M, a dynamic programming calculation can be used to calculate, for a given sequence, the probablity that sequence a was generated by the model, $P(a|M)$. The question of interest is, however, does that model match the sequence? That is, is the sequence more likely to be generated by the hidden Markov model than some other, less structured null model, ϕ. Making the assumption that the models M and ϕ are *a priori* equally likely, this reduces to the log-odds probability of

$$P(M|a) = \frac{p(a|M)}{p(a|M) + p(a|\phi)}.$$

Typically, a log-odds score S is used instead (Altschule 91):

$$\begin{aligned} S_a &= \ln \frac{P(a|M)}{P(\phi|a)} \\ P(M|a) &= \frac{1}{1+e^{-S}} \end{aligned}$$

This score measures whether the probability that the sequence was generated by the model is greater than the probability it was generated by a null model. This log-odds score is used to calculate the weight of a sequence. As model training iterations proceed, sequences that poorly match the model (i.e. with poor log-odds scores) are given higher weight.

Internal weighting method 1 uses the following equation to calculate a sequence weight.

$$W = e^{(w-S)*(\frac{\ln K}{(w-b)})}$$

S is the log-odds score of the sequence. The program keeps track of the current worst score w and the current best score b and these are used to decide on the two extreme weights. The worst scoring sequence will have a weight of 1, while the best scoring sequence will have a weight of K, typically in the 0.01 to 0.1 range. K is a user-controlled parameter entered on the command line as *iweight*.

To use method 1 with K=.1, run buildmodel with the following arguments:

```
buildmodel runname -train train.seq -internal_weight 1 -iweight .1
```

Internal weighting method 2 is a variation of method 1. When using method 1, sequences with very poor scores may get excessively large weights. Method 2 modifies the weights of such outlier sequences. If a sequence scores so badly that it exceeds the median score by three standard deviations, it is weighted with a decreasing linear weight function, reaching a minimum of 1.0 for the sequence with the worst score.

Method 2:

$$W = e^{(w-S*\frac{\ln K}{(w-b)})}$$

is applied to scores no more than 3 deviations below the median, while

$$W = 1 - \frac{S-w}{b-w}$$

is applied to the remaining sequences.

To use method 1 with K=0.05, run buildmodel with the following arguments:

```
buildmodel runname -train train.seq -internal_weight 2 -iweight 0.05
```

The current SAM default is to not use internal weighting. If internal weighting is selected and no iweight parameter entered, SAM defaults to an iweight of 0.1.

Looking again at the toy problem example demonstrated in the previous section, we saw the following scores when a model was trained on 4 globins without weighting.

```
GLB$APLKU        70     -138.28      53.50
GLB$APLLI        70     -138.04      50.91
GLB$APLJU        70     -131.19      56.49
BAHG$VITSP       70      -97.16     101.29
```

The unweighted model is overspecialized.

Internal weighting method 1 produces these scores:

```
GLB$APLLI        70      -99.47      91.42
GLB$APLJU        70      -98.70      91.61
GLB$APLKU        70      -97.10      95.98
BAHG$VITSP       70      -93.66     103.20
```

Internal weighting method 2 produces these scores:

```
GLB$APLLI        70      -95.28      95.93
GLB$APLJU        70      -93.43      97.25
GLB$APLKU        70      -91.59     101.63
BAHG$VITSP       70      -84.48     112.20
```

When using internal weighting, you can inspect all sequence weights generated during each iteration of the model building process.

```
buildmodel runname -train train.seq -internal_weight 2 -iweight 0.05 -print_all_weights
1
```

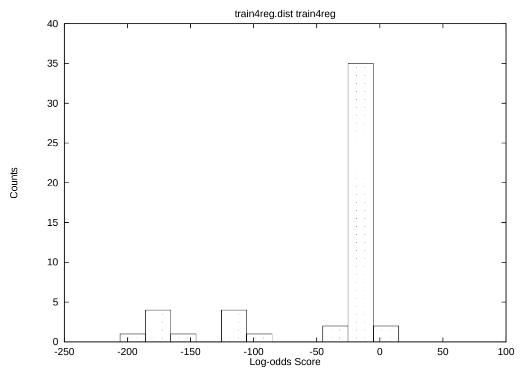
The `print_all_weights` option when set to 1 will produce a weight output file once per iteration. The files are named `runname1.weightoutput`, where 1 is the iteration number.

By default, `print_all_weights` is set to off.

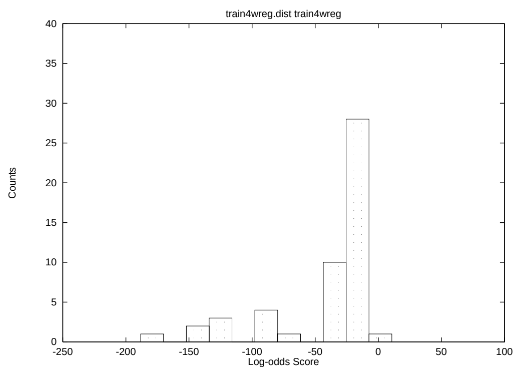
Continuing the example of Figure 4 on page 29, performing the two commands:

```
buildmodel train4w -train globins50.seq
            -seed 0 -trainseed 0 -ntrain 4 -internalweight 2
buildmodel train4wreg -train globins50.seq
            -priorlibrary recode1.20comp
            -regularizerfile weak-gap.regularizer
            -seed 0 -trainseed 0 -ntrain 4 -internalweight 2
hmmscore train4w  -i train4w.mod  -db globins50.seq -sw 2
hmmscore train4wreg -i train4wreg.mod -db globins50.seq -sw 2
```

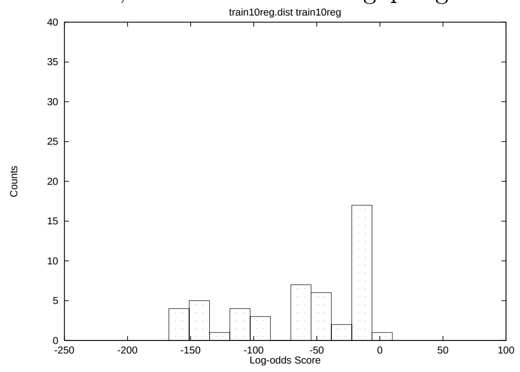
results in the score histograms of Figure 5, in which the scores improve even further from the use of regularizers and Dirichlet mixtures.



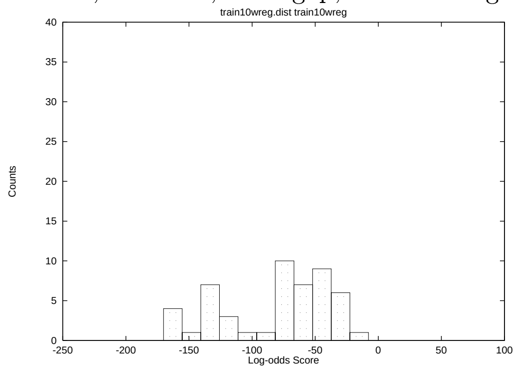
Ntrain = 4, Dirichlet and weak-gap regularizer



Ntrain = 4, Dirichlet, weak-gap, internal weighting



Ntrain = 10, Dirichlet and weak-gap regularizer



Ntrain = 10, Dirichlet, weak-gap, internal weighting

Figure 5: Weighting performance

8.5 Viterbi training

For increased speed and performance (and possible worse results), Viterbi training is now possible (though in some instances not robust). The `viterbi` parameter should be set to 1 for Viterbi training. Future research will help determine the usefulness of this option. Fragment training (Section 9.1.1) can also be used, but is not recommended because the jump transitions are not currently trained; instead jumps into the model are replaced with sequences of delete states. Thus, training with fragments will over-emphasize the delete states in the model.

9 Related programs

9.1 `align2model` and `prettyalign`

After you have obtained a model of your sequence family, `align2model` can be used to give a multiple alignment of sequences. Often one is just interested in aligning the sequences that were also used to train the model, but in principle any sequence can be included in the alignment.

The multiple alignment is obtained by aligning each of the sequences to the model by the Viterbi algorithm. This has the advantage that it can be done for each sequence independently, and therefore it is very simple to add new sequences to the multiple alignment. Also, once the model is found, the multiple alignment is very fast and easy to produce.

The program to produce the basic alignment is called `align2model`. Calling it with no arguments gives a brief explanation. To align all the sequences in `trna10.seq` use the command:

```
align2model trna10 -i test.mod -db trna10.seq
```

This will put an intermediate form of the alignment in the file `trna10.a2m`. In this FASTA-compatible intermediate format deletions are shown as dashes ('-') and insertions (produced in the insert states of the model) are shown as lower case characters, while periods ('.') are used to fill in the sequences that did not have any insertions if `a2mdots` is set to 1, the default.

In case you only want to align a few sequences in a large file, you can specify the identifiers of these sequences on the command line. For instance

```
align2model trna2 -i test.mod -db trna10.seq -id TRNA1 -id TRNA9
```

will only align the two specified sequences. The output (in `trna2.a2m`) would look like this:

```
>TRNA1, 77 bases, ADEFE3D4 checksum.
GGGGAUGUAGCUCAG-.UGGUAGAGCGCAUGCUUCGCAUGUAUGAGGCCC
CGGGUUCGAUCCCCGGCAUCU---CCA
>TRNA9, 77 bases, 71415595 checksum.
CGGCACGUAGCGCAGCcUGGUAGCGCACCGUCCUGGGGUUGCGGGGGUCG
GAGGUUCAAAUCCUCUCGUGCCGACCA
```

To get a nice display of the alignment produced by `align2model`, you can use the program `prettyalign`, which has several display options. The program reads from a file like the one made in the example above:

```
prettyalign trna10.a2m > trna10.pretty
```

which would give you an alignment similar to the one shown in the Section 3

`Prettyalign` does not follow SAM's normal commandline format. To see an explanation of the various options, run the program with some invalid option (like `prettyalign -h`). Some of the most useful options are:

-f Print in a FASTA-like format.

-i Do not include sequence identifiers in front of each line.

-l *num* Set the output line length equal to *num*.

-n Toggle indexing the sequences, as well as labeling them.

-c Toggle column numbering.

-m Set maximum insertion length (longer insertions are printed as their length).

-I I-G style alignment. Also sets maximum insertion length very high.

The commands

```
align2model  trna3 -i test.mod -db trna10.seq -id TRNA1 -id TRNA2 -id TRNA9
prettyalign trna3.a2m -l 50 > trna3.pretty
```

gives the following output

```
            10            20        30        40
             |             |         |         |
TRNA1 GGGGAUGUAGCUCAG-.UGG...UAGAGCGCAUGCUUCGCAUGU
TRNA2 GCGGCCGUCGUCUAGU.CUGgauUAGGACGCUGGCCUCCCAAGC
TRNA9 CGGCACGUAGCGCAGCcUGG...UAGCGCACCGUCCUGGGGUUG
            50        60        70
             |         |         |
TRNA1 AUGAGGCCCCGGGUUCGAUCCCCGGCAUCU---CCA
TRNA2 CAGCAAUCCCGGGUUCGAAUCCCGGCGGCCG---CA
TRNA9 CGGGGGUCGGAGGUUCAAAUCCUCUCGUGCCGACCA
```

The `prettyalign` program can compress long insertions to only the initial segment of bases in the insertion plus digits representing the *total* length of the insertion. For example, the sequence `GacguacguG` could be printed out as `Ga8guG` if 4 was the largest number of insertions that was to be allowed (note that the character 8 is using up one of the positions). By default, insertions of up to length ten thousand are fully printed. This can be changed with the `-m` flag to `prettyalign`, which sets the maximum number of insertions that are printed. If set to zero, no insertions are printed, and no indication of the lack is given. If less than zero, insertion characters are not printed, and that number of digits is used to indicate the length of each insertion.

For example, the alignment at the end of Section 9.3.1 could alternatively be created using the command

```
prettyalign ftrain.mod -m 8 > ftrain2.pretty
```

to produce the alignment:

```
                       10          20        30        40        50
                        |           |         |         |         |
TRNA1  ........GGGGAUGUAGCUCAGU-GG..UAGAGCGCAUGCUUCGCAUGUAUGAGGCCCCGGG
TRNA2X gcg18ugc-GGCCGUCGUCUAGUCUGGauUAGGACGCUGGCCUCCCAAGCCAGCAAUCCCGGG
TRNA3X ccc.....GGCCCUGUGGCUAGCUGGU..CAAAGCGCCUGUCUAGUAAACAGGAGAUCCUGGG
TRNA4X ggg16cagGGCGAAUAGUGUCAGCGGG..-AGCACACCAGACUUGCAAUCUGGUAGGGA-GGG
TRNA5X ........GCCGGGAUAGCUCAGUUGG..UAGAGCAGAGGACUGAAAAUCCUCGUGUCACCAG
TRNA6  ........GGGGCCUUAGCUCAGCUGG..GAGAGCGCCUGCUUUGCACGCAGGAGGUCAGCGG
TRNA7  ........GGGCACAUGGCGCAGUUGG..UAGCGCGCUUCCCUUGCAAGGAAGAGGUCAUCGG
TRNA8  g.......GGCCCGUGGCCUAGUCUGGa.UACGGCACCGGCCUUCUAAGCCGGGGAUCGGGGG
TRNA9  c.......GGCACGUAGCGCAGCCUGG..UAGCGCACCGUCCUGGGGUUGCGGGGGUCGGAGG
TRNA10 uc......-CGUCGUAGUCUAGGUGGU..UAGGAUACUCGGCUUUCACCCGAGAGA-CCCGGG
            60        70
             |         |
TRNA1  UUCGAUCCCCGGCAUCUCC-a........
TRNA2X UUCGAAUCCCGGCGGCCGC-acg12gca.
TRNA3X UUCGAAUCCCAGCGGGGCC-uccagggg.
TRNA4X UUCGAGUCCCUCUUUGUCC-acca.....
TRNA5X UUCAAAUCUGGUUCCUGGC-aug13gca.
TRNA6  U-CGA-CCCGCUAGGCUCC-acca.....
TRNA7  UUCGAUUCCGGUUGCGUCC-a........
TRNA8  UUCAAAUCCCUCCGGGUCC-g........
TRNA9  UUCAAAUCCUCUCGUGCCG-acca.....
TRNA10 UUCAAGUCCCGGCGACGGA-acca.....
```

The `-I` will create a compatible IG-style alignment file which may be converted to other formats using the `readseq` package included as a subdirectory of SAM. The `-I` option automatically sets a high value for the insertion length parameter.

9.1.1 Aligning fragments

Consider a model of 100 nodes and a fragment of 25 that very closely matches some contiguous section of the model. Even though that section would align very well, the overall alignment of the fragment could be quite poor because of its need to use 75 delete states in the model. The problem here is that in addition to modelling conserved regions, the model also models the length of the conserved region.

SAM has two alignment options for dealing with fragments. The first option allows a sequence to start matching the model at any location (rather than only the begin node) and end at any location (rather than only the end node). This will improve alignment for short sequences that match a segment of the model. This option can be turned on by setting the `SW` variable to one.

The second option is similar to Smith and Waterman method of sequence comparison, which will find the best alignment for any pair of subsequences within two sequences. The same can be done with models, allowing a submodel to match a subsequence. This type of dynamic programming can be specified by setting the `SW` variable to 2. When this is done, sequences can jump from the initial module (presumably a FIM, automatically added when `auto_fim` is set) into the delete state of any module in the model, and can also jump out of the delete state of any module within the model to the delete state of the next-to-last node. The first and next-to-last module are assumed to be FIMs, hence the rational is that a sequence will use the FIM for some period of time to consume characters that do not match the model, then the sequence will jump to the model node corresponding to the start of the fragment, use several model nodes, and then jump to the ending FIM to consume the rest of the sequence.

The probability of these jumps is set by the variables `jump_in_prob` and `jump_out_prob`, both of which have a default value of unity. That is, as in the sequence-to-sequence Smith and Waterman, there is no cost associated with jumping in and out of the model.

The file `trna1frag.seq` contains several sequences that contain part or all of TRNA1. The sequence include TRNA1 (72 bases), TRNA1Long (the complete TRNA with additional characters), Long (58 base segment of TRNA1), Medium (34 base segment), Short (6 base segment TRNA1), and AAMediumA, an embedding of Medium within several segments of As to bring it to 176 characters. Additionally, the file contains several (obviously) non-TRNAs of various lengths, all of whose IDs begin with the word 'Not'.

When this file is aligned to the model `test.mod`, created above, the alignment of the sequence and fragments is reasonable, but the non-tRNAs still align the entire model and may even use internal insertion states. (As shall be seen in Section 9.2.4, the scoring of these fragments with the SW option off is not nearly so good as their alignments).

```
                               10                20        30        40
                                |                 |         |         |
TRNA1          ........GGGGAUGUAGCUCAG-UGG........UAGAGCGCAUGCUUCGCAUGUAUGA
TRNA1Long      aaa11aaaGGGGAUGUAGCUCAG-UGG........UAGAGCGCAUGCUUCGCAUGUAUGA
Short          ........-------------------........-------------------------
ShortReverse   ........UCGUAC-------------........-------------------------
Medium         ........-------------------........--GAGCGCAUGCUUCGCAUGUAUGA
MediumReverse  ........UUGGGC---------CCCG........GAGUAUGUACGCUUCGUACGCGAG-
AAMediumAA     ........AAAAAAAAAAAAAAAAAAAaaa50aaaAAGAGCGCAUGCUUCGCAUGUAUGA
Long           ........---------GCUCAG-UGG........UAGAGCGCAUGCUUCGCAUGUAUGA
NotShort       ........-------------------........---------------AAAAAAAAAA
Not            ........AAAAAAAAAAAAAAA---A........AAAAAAAAAAAAAAAAAAAAAAAAA
NotLong        ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
NotExtraLong   ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
NotExExtraLong ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
NotExExExLong  ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
NotExExExExL   ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
                   50        60        70
                    |         |         |
TRNA1          GGCCCCGGGUUCGAUCCCCGGCAUCU---CCA.........
TRNA1Long      GGCCCCGGGUUCGAUCCCCGGCAUCUCCAAAAaaaaaaaa.
Short          --------------------------CAUGCU.........
ShortReverse   --------------------------------.........
Medium         GGCCCCGGGUU---------------------.........
MediumReverse  --------------------------------.........
AAMediumAA     GGCCCCGGGUUAAAAAAAAAAAAAAAAAAAAAaaa50aaa.
Long           GGCCCCGGGUUCGAUCCCCGGCAU--------.........
NotShort       AA-----------------------------A.........
Not            AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.........
NotLong        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaaa33aaa.
NotExtraLong   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa362aaa.
NotExExtraLong AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa800aaa.
NotExExExLong  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa1676aa.
NotExExExExL   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa3428aa.
```

Alignment with the SW option set to 1 is much the same (again, scoring will be improved), though the alignment procedure has managed to better isolate the AAMediumAA sequence's TRNA core, modeled by match states, from its prefix and postfix, modeling by internal insertion nodes.

```
                                10                 20         30         40
                                 |                  |          |          |
TRNA1          ........GGGGAUGUAGCUCAG-UGG........UAGAGCGCAUGCUUCGCAUGUAUGA
TRNA1Long      aaa11aaaGGGGAUGUAGCUCAG-UGG........UAGAGCGCAUGCUUCGCAUGUAUGA
Short          ........CAUGCU-------------........-------------------------
ShortReverse   ........UCGUAC-------------........-------------------------
Medium         g.......-------------------........---AGCGCAUGCUUCGCAUGUAUGA
MediumReverse  u.......----------------UGGgccccg..GAGUAUGUACGCUUCGUACGCGAG-
AAMediumAA     ........AAAAAAAAAAAAAAAAAAAaaa50aaaAAGAGCGCAUGCUUCGCAUGUAUGA
Long           g.......-----------CUCAGUGG........UAGAGCGCAUGCUUCGCAUGUAUGA
NotShort       a.......-------------------........---------------AAAAAAAAAA
Not            ........AAAAAAAAAAAAAAA---A........AAAAAAAAAAAAAAAAAAAAAAAAA
NotLong        ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
NotExtraLong   ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
NotExExtraLong ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
NotExExExLong  ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
NotExExExExL   ........AAAAAAAAAAAAAAAAAAA........AAAAAAAAAAAAAAAAAAAAAAAAA
                    50        60        70
                     |         |         |
TRNA1          GGCCCCGGGUUCGAUCCCCGGCAUCUCCA---.........
TRNA1Long      GGCCCCGGGUUCGAUCCCCGGCAUCUCCAAAAaaaaaaaa.
Short          --------------------------------.........
ShortReverse   --------------------------------.........
Medium         GGCCCCGGGUU---------------------.........
MediumReverse  --------------------------------.........
AAMediumAA     GGCCCCGGGUUAAAAAAAAAAAAAAAAAAAAAaaa50aaa.
Long           GGCCCCGGGUUCGAUCCCCGGCAU--------.........
NotShort       AA------------------------------.........
Not            AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.........
NotLong        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaaa33aaa.
NotExtraLong   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa362aaa.
NotExExtraLong AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa800aaa.
NotExExExLong  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa1676aa.
NotExExExExL   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAaa3428aa.
```

When alignment is performed using the SW_score option set to 2, only the core TRNA segments are aligned: the non-TRNA's, as well as the prfix and postfix of AAMediumAA are aligned to the FIMs that have been automatically added to the model. The one problem is that the Short sequence has made some use of the end FIM because it is not long enough to make a really significant hit to the model's internal nodes.

```
                         10         20         30         40         50
                          |          |          |          |          |
TRNA1           ........GGGGAUGUAGCUCAG-UGGUAGAGCGCAUGCUUCGCAUGUAUGAGGCCCCGG
TRNA1Long       aaa11aaaGGGGAUGUAGCUCAG-UGGUAGAGCGCAUGCUUCGCAUGUAUGAGGCCCCGG
Short           c.......----------------------------------------------------
ShortReverse    ucgu....----------------------------------------------------
Medium          g.......----------------------AGCGCAUGCUUCGCAUGUAUGAGGCCCCGG
MediumReverse   uug11cgg----------------------------------------------------
AAMediumAA      aaa70aaa--------------------AGAGCGCAUGCUUCGCAUGUAUGAGGCCCCGG
Long            gcucag..----------------UGGUAGAGCGCAUGCUUCGCAUGUAUGAGGCCCCGG
NotShort        a.......----------------------------------------------------
Not             a.......----------------------------------------------------
NotLong         a.......----------------------------------------------------
NotExtraLong    a.......----------------------------------------------------
NotExExtraLong  a.......----------------------------------------------------
NotExExExLong   a.......----------------------------------------------------
NotExExExExL    a.......----------------------------------------------------
                       60         70
                        |          |
TRNA1           GUUCGAUCCCCGGCAUCUCCA----.........
TRNA1Long       GUUCGAUCCCCGGCAUCUCCAAAA-aaaaaaaa.
Short           -----------------------A-ugcu.....
ShortReverse    -----------------------A-c........
Medium          GUU----------------------.........
MediumReverse   -----------------------A-gua22gag.
AAMediumAA      GUUAAAA------------------aaa67aaa.
Long            GUUCGAUCCCCGGCAU---------.........
NotShort        -----------------------A-aaa11aaa.
Not             -----------------------A-aaa71aaa.
NotLong         -----------------------A-aa107aaa.
NotExtraLong    -----------------------A-aa436aaa.
NotExExtraLong  -----------------------A-aa874aaa.
NotExExExLong   -----------------------A-aa1750aa.
NotExExExExL    -----------------------A-aa3502aa.
```

9.2 `hmmscore`

Any sequence can be compared to a model by calculating the probability that the sequence was generated by that model. Taking the negative (natural) logarithm of this probability gives the NLL score. For sequences of equal length the NLL scores measures how 'far' they are from the model, and it can be used to select sequences that are from the same family. However, the NLL score has a strong dependence on sequence length and model length. A less biased score is the NLL divided by the length of the sequence, although that is still not perfect. `Hmmscore` provides two less biased means of scoring sequences. The default way is by reporting NLL scores as the difference between a null model and trained model NLL score (a log-odds score, as used in HMMER). The second way is by its Z-score, that is, the number of standard deviations the NLL is away from the average NLL of sequences of the same length.

Null model scoring is discussed in more detail in the Barrett, Karplus, and Hughey paper mentioned in the introduction and available from the SAM WWW page. (`http://www.cse.ucsc.edu/research/compbio/papers/nullmod/nullmod.html`).

The program hmmscore can find NLL, NLL−NULL (log-odds), and Z-scores. The most common operation is to calculate NLL−NULL scores for a large number of sequences. This can be done by supplying the name of the model file and one or more sequence database files on the command line, optionally followed by hmmscore parameter specifications. For instance, for the example files described earlier the NLL scores are found the following way

```
hmmscore test -insert test.mod -db trna10.seq -sw 2
```

Produces the file `test.dist` already displayed:

```
%  SAM: ../src/hmmscore v2.1.1  (Apr 24, 1998) compiled 04/27/98_12:32:02.
%  SAM:  Sequence Alignment and Modeling Software System
%  (c) 1992-1998 Regents of the University of California, Santa Cruz
%  http://www.cse.ucsc.edu/research/compbio/sam.html
%
% ------ Citations (HMMs, SAM) ------
% A. Krogh et al., Hidden Markov models in computational biology:
%   Applications to protein modeling, JMB 235:1501-1531, Feb 1994.
% R. Hughey, A. Krogh, Hidden Markov models for sequence analysis:
%   Extension and analysis of the basic method, CABIOS 12:95-107, 1996.
% ----------------------
%  Run start: Mon Apr 27 12:50:01 1998
%  Run name:  test
%  On host:   alpha
%  In dir:    /auto/projects/compbio3/samtmp/sam/SAMBUILD/alpha/demos
%  By user:   rph
% -------------------------------------------------------------
% Inserted Files:  test.mod
% Database Files:  /projects/compbio3/samtmp/sam/demos/trna10.seq
%
% FIMs automatically added (auto_fim = 1).
% Subsequence-submodel (local) (SW = 2).
% S&W simple NULL scores adjusted by ln(seqlen) (adjust_score=2).
% 10 sequences with 747 residues.
% This run used EM scoring.
% The model has 77 positions.
%
% Using total residues as number of starting possibilities,
%  0.01 significance at <= ln(0.01)-ln(747)= -4.6 -  6.6 = -11.2
% Adjusting for model length gives 10 starting points,
%  0.01 significance at <= ln(0.01)-ln(10) = -4.6 -  2.3 = -6.9
% If entire sequences are modeled (i.e., no FIMs),
%  0.01 significance at <= ln(0.01)-ln(10) = -4.6 -  2.3 = -6.9
% Values for 10.0 significance are  -4.3,  -0.0, and  -0.0.
% Significance level is higher for multiple scoring runs.
% Sequence scores selected :  All  (select_score=8)
%
% Column 1: NLL-NULL using simple FIM (node 0) insert probabilities
% Column 2: the raw NLL score
% Scores sorted by column 1, best first
%
% Sequence ID   Length      Simple         Raw      X count
TRNA7              73       -36.52       58.72
TRNA3              76       -35.59       64.57
TRNA8              75       -34.61       59.74
TRNA2              76       -33.78       63.24
TRNA9              77       -33.54       63.72
TRNA1              72       -33.33       60.42
TRNA5              73       -32.77       67.51
TRNA6              74       -32.76       61.07
TRNA4              75       -32.51       67.67
TRNA10             76       -31.78       69.03
```

As discussed in Section refsec:score, the score file contains five columns. The first is the sequence identifier, followed by sequence length, the 'NLL-NULL' score using a simple null model. The next score column is either the raw NLL score if only the simple null model is calculated (the default), or the complex null model's 'NLL-NULL' score if one of the more time-consuming null models is used, as discussed below. If Z-scoring is used, Z-scores are listed after the two score columns. Last, the number of all-character wildcards in each sequence is listed for those that have any wildcards.

By default, `hmmscore` uses the EM scoring method, just as is used to train a model. If desired, scores can be based on exact alignment to the model, multiplying the probabilities along the best path rather than all paths. This method, which corresponds to the forward half of `align2model`, can be turned on by setting `viterbi` to 1. Viterbi scoring is appropriate for finding out how good a sequence's best alignment to a model is.

The `hmmscore` program can also be used to select sequences according to various criteria.

Plots of the NLL–NULL scores or Zscores can be used to visually look for a break between significant and insignificant matches. See Section 9.6 on page 78.

If any all-character wildcards are present in a sequence (e.g., 'X' in proteins or 'N' in nucleic acids), the number of wildcards is reported after the score.

9.2.1 NLL–NULL scoring

SAM includes several possibilities for NULL model scoring. First, the null model can be a simple probability distribution, effectively a model with a single FIM. Second, the NULL model can be a regularizer of similar structure to the model being scored. Third, the null model can be any model specified in SAM format, with the key word 'NULLMODEL' (rather than, for example, 'MODEL' or 'REGULARIZER'), or the first model in a file specified with the `nullmodel_file` parameter.

To report differences between the model NLL score and the simple null model score (possibly modified by `FIM_method_score`, see below), set the `subtract_null` variable to 1 (the default). To report differences between the model and a similarly structured null model (a complex null model, which uses the transition and insert probabilities of the model and the geometric average of the model's match states for its own match states), set `subtract_null` to 2. To report differences between two models (for example, one trained on positive family examples and one trained on negative examples of a family), set `subtract_null` to 3. To report differences between the score of the sequence and the score of the reversed sequence, which provides an automatic adjustment for compositional bias, set `subtract_null` to 4.

The simple null model score (`subtract_null`=1) is actually calculated in all cases. If the other null model calculations were to be calculated all the time, it would double the running time of `hmmscore`. The `simple_threshold` variable can be used to control when the more time consuming score should be calculated: the complex null model score will only be calculated when the simple null model score is less than `simple_threshold`. Sequences for which the more time consuming null model was not calculated will have the score 10000 in the second score column of the distance file. The default value of `simple_threshold` is 0.

Our current favorites are the simple null model (geometric average) mixed with the reverse sequence null model for well-scoring sequences. The command:

```
    hmmscore testrev -i test.mod -db trna10.seq -subtract_null 2 -sw 2
```

produces a score file that includes both simple and reverse-sequence null model scores:

```
%  SAM: ../src/hmmscore v2.1.1  (Apr 24, 1998) compiled 04/27/98_12:32:02.
%  SAM:  Sequence Alignment and Modeling Software System
%  (c) 1992-1998 Regents of the University of California, Santa Cruz
%  http://www.cse.ucsc.edu/research/compbio/sam.html
%
% ------ Citations (HMMs, SAM) ------
% A. Krogh et al., Hidden Markov models in computational biology:
%   Applications to protein modeling, JMB 235:1501-1531, Feb 1994.
% R. Hughey, A. Krogh, Hidden Markov models for sequence analysis:
%   Extension and analysis of the basic method, CABIOS 12:95-107, 1996.
% -----------------------
%  Run start: Mon Apr 27 12:50:07 1998
%  Run name:  testrev
%  On host:   alpha
%  In dir:    /auto/projects/compbio3/samtmp/sam/SAMBUILD/alpha/demos
%  By user:   rph
% --------------------------------------------------------------
% Inserted Files:  test.mod
% Database Files:  /projects/compbio3/samtmp/sam/demos/trna10.seq
%
% FIMs automatically added (auto_fim = 1).
% Subsequence-submodel (local) (SW = 2).
% S&W simple NULL scores adjusted by ln(seqlen) (adjust_score=2).
% 10 sequences with 747 residues.
% This run used EM scoring.
% The model has 77 positions.
%
% Using total residues as number of starting possibilities,
%  0.01 significance at <= ln(0.01)-ln(747)= -4.6 -  6.6 = -11.2
% Adjusting for model length gives 10 starting points,
%  0.01 significance at <= ln(0.01)-ln(10) = -4.6 -  2.3 = -6.9
% If entire sequences are modeled (i.e., no FIMs),
%  0.01 significance at <= ln(0.01)-ln(10) = -4.6 -  2.3 = -6.9
% Values for 10.0 significance are  -4.3,  -0.0, and  -0.0.
% Significance level is higher for multiple scoring runs.
% Sequence scores selected :  All  (select_score=8)
%
% Column 1: NLL-NULL using simple FIM (node 0) insert probabilities
% Column 2: NLL-NULL for the reverse sequence NULL model
% Scores sorted by column 1, best first
%
% Sequence ID   Length      Simple     Reverse      X count
TRNA7              73       -36.52      -32.65
TRNA3              76       -35.59      -31.70
TRNA8              75       -34.61      -30.41
TRNA2              76       -33.78      -29.88
TRNA9              77       -33.54      -29.65
TRNA1              72       -33.33      -29.50
TRNA5              73       -32.77      -28.61
TRNA6              74       -32.76      -28.22
TRNA4              75       -32.51      -28.62
TRNA10             76       -31.78      -27.91
```

The NLL–NULL scores, especially for the simple null model, are most useful when the model has had free insertion modules (Section 7.5) added to it. Then, the null model and the FIMs will cancel out, and the score will be based primarily on the section of the sequence that matches the region that has been modeled. By default, `hmmscore` automatically adds FIMs to any model that does not already contain them when the simple or complex null model is used. To change this, if for example you want to ensure that entire sequences are modeled, rather than simply subregions, change `auto_fim` from its default value of 1 to 0. The `auto_fim` variable has no effect when a user-specified null model is used.

Since NLL–NULL scores are negative logs, the lower the better. In the case above, all of the tRNA's have been positively identified as tRNAs. (Not surprising as they were all in the training set!)

The question of what scores are significant can be addressed mathematically. For aligning a sequence to a model with a 0.01 significance, one must score better than $\ln(0.01) = -4.6$. At this setting, the expected number of false positives (sequences incorrectly labeled as belonging to the family) is 0.01. For searching an entire database, however, the effect of the vast number of places the model will be fitted against be taken into account: For fitting a model against D locations $-4.6 - \ln(D)$ must be scored to be significant. The number of locations can be decided in several ways. First, if the model is viewed as matching entire sequences (for example, if free insertion modules have not been added to the model), then the number of locations is equal to the number of sequences in the database. If the model does have FIMs, it can match arbitrary regions of any sequence. If the model is regarded as fitting a relatively contiguous section of a sequence, then the number of characters in the database roughly corresponds to the number of starting positions, so the log of the database size should be used. For the most conservative significance estimate, if the model is regarded as matching any subsequence of the model, the number of positions is squared, or the log multiplied by two, to arrive at the significance cutoff. Interested readers may wish to refer to the work of Milosavljević and Jurka (CABIOS 9(4):407–411) for more information.

The BLAST program's E parameter is the same as `hmmscore`'s significance level. BLAST has a default significance of 10, indicating an expected 10 false positives during a discrimination experiment.

The `hmmscore` program will report all of these significance tests for running against the sequences it is given. However, if a model is used in multiple `hmmscore` runs against different (non-redundant) databases, the significance level must be calculated based on the entire experiment, not an individual `hmmscore` run.

New to Version 1.2 is the ability to adjust the null model scoring. Since this determines the probability that a sequence was randomly generated according to the residue insertion probabilities, these values should reflect knowledge of the problem domain. Five possibilities are offered. The flat distribution or the background distribution of amino acids over all proteins can be used. Both of these distributions are invariant over all families, and are thus a simplistic assumption. The distribution can also be the distribution of the residues in the training set or the average residue distribution over all columns (match states) modeled by the HMM. The advantage of these two, especially the latter, is the ability to correct for compositional bias in the sequences. Lastly, the insertion probabilities can reflect the residue distribution of the sequence currently being scored. This is the most pessimistic null model, as it demands not that the HMM model the sequence better than fixed background frequencies, but that is model the sequence significantly better than frequencies exactly matching the sequence's composition.

So the options available for `FIM_method_score` are

0 Use the tables present in the model.

1 The relative frequencies of residues in the training sequences (from the Lettercounts node or the training sequences).

2 The relative frequencies of residues in model match states (from the Frequency node).

3 Uniform (flat) probability over all residues.

5 Amino acid background frequencies over all proteins (from the Generic node).

6 Geometric average of the match state probabilities in the model.

The default setting, for experimental and statistical reasons, is the geometric average of the model match states (6). The insertion tables can be similarly modified with `Insert_method_score`, the default of which is no change (0).

As with the training methods, if the method value is negative instead, the FIMs and insert tables will only be modified if there is no initial model read in (an unlikely occurrence for `hmmscore`).

A more detailed discussion of these issues can be found in (Barrett, Hughey & Karplus 1996), mentioned in the Introduction.

9.2.2 Z-score scoring

The SAM system's second scoring method is the use of Z-scores for database sequences of similar length. This ability has two parts:

Finding a smooth curve: If there are sufficiently many sequences in the file, a smooth interpolation through the data is found (a curve in a score vs. NLL plot). This is used to calculate Z-scores, or for each sequence, the number of standard deviations the NLL score is away from the smooth curve. The specification of that smooth curve can be found in a file with the same name as the model file, except that the extension is changed to ".smooth" (test.smooth in the above example). See below for more details. A smooth curve can also be calculated (or recalculated) from existing NLL scores.

Calculating Z-scores: If a smooth curve is calculated, the Z-score for each sequence is also found. If a file with the extension ".smooth" already exists, the Z-score is also calculated based on this. The use of Z-scores assumes that scores form a normal distribution at each length, a condition which often does not hold.

9.2.2.1 Calculating the smooth curve When searching a database, one would like to know if the NLL score of a sequence is significantly lower than it is for other sequences of the same length. This is what the Z-score tells you. To calculate Z-scores one has to find the average and standard deviation of the scores for the bulk of the sequences (excluding 'outliers', which are usually the interesting sequences). The program `hmmscore` does it the following iterative way:

1. For all possible length intervals containing a certain number of sequences (usually 1000) the average NLL score and average sequence length is calculated. These numbers define the smooth curve. The standard deviation around this curve is found also.
2. All sequences with an NLL score more than a certain number of standard deviations from the smooth curve are considered outliers and excluded in the next iteration.
3. If there are no new sequences excluded, the process is stopped. Otherwise it is repeated (unless it has ran for some maximum number of iterations).

This procedure often produces good results, but there is **no guarantee that it works**. To be sure to obtain a nice smooth curve, one can take out all sequences that are known to be outliers (like the training set) and then run the program on all the rest. It is also a good idea to throw out all sequences that contain wildcard characters before the smooth curve is calculated. This can be done by editing the file containing all the NLL scores calculated by `hmmscore` in the first place, removing the lines which include a wildcard count, such as the second line of:

```
RND_ECOLI          375    605.416     0.00000    37.551
CA21_CHICK        1362   3975.208     0.00000    33.049  601 X
```

9.2.3 Selecting sequences, scores, and alignments

Sequences can be selected by `hmmscore` and placed in a `.sel` file.

A selection mode is chosen by setting `select_seq`. If 0, no sequences are selected; if 1, sequences are selected according to their simple null model scores and `NLLNull`; if 2, sequences are selected according to their column 2 score (complex, user, or reverse sequence null, or raw NLL score if `subtract_null` is 0 or 1) and `NLLcomplex`; if 4, sequences are selected according to their Z-scores and `Zmax`; if 8, all sequences are selected. Selection criteria can be combined: 3 requires sequences to score better than `NLLnull` with the simple null model and `NLLcomplex` with the complex null model. Negative numbers indicate that sequences that do not pass the corresponding positive test should be selected.

The following will place labeled copies of all sequences scoring lower than -35 into `test.sel`.

```
hmmscore tests -i test.mod -db trna10.seq -select_seq 1 -NLLNull -35 -sw 2
```

Selected sequences are written out in the same order they are encountered in the database files, which may be different from the order they are listed in the score file if scores are sorted. The `sortseq` program can be used to write out the sequences in the same order they are listed in the score file. See Section 9.7.4 on page 83. The `sort` variable controls whether sequence scores are unsorted (0); sorted by Z-score (4), column 2 of the distance file (2), or column 1 of the distance file (1). Sequences in the select file will be selected by other than the sorting criteria unless `sort` and `select_seq` are set to corresponding values.

The `select_score` variable can be set in the same manner as `select_seq`, in which case only scores of those sequences that match the specified criteria will be recorded in the distance file. This is particularly useful for database searches in which only sequence IDs are of interest.

The `select_align` variable can be used in a similar way to cause selected sequence alignments to be placed in the `runname.a2m` file. Note that all selection variables use the same `NLLnull`, `NLLcomplex`, and `Zmax` thresholds, though different combinations of the thresholds can be used by the different parameters.

9.2.4 Scoring Fragments

Consider a model of 100 nodes and a fragment of 25 that very closely matches some contiguous section of the model. Even though that section would score very well, the overall score of the fragment could be quite poor because of its need to use 75 delete states in the model. The problem here is that in addition to modelling conserved regions, the model also models the length of the conserved region.

SAM has two scoring options for dealing with fragments. The first option allows a sequence to start matching the model at any location (rather than only the begin node) and end at any location (rather than only the end node). This will improve scoring for short sequences that match a segment of the model. This option can be turned on by setting `hmmscore`'s `SW` variable to one.

The second option is similar to Smith and Waterman method of sequence comparison, which will find the best score for any pair of subsequences within two sequences. The same can be done with models, allowing a submodel to match a subsequence. In `hmmscore`, this type of scoring can be specified by setting the `SW` variable to 2. When this is done, sequences can jump from the initial module (presumably a FIM) into the delete state of any module in the model, and can also jump out of the delete state of any module within the model to the delete state of the next-to-last node. The first and next-to-last module are assumed to be FIMs, hence the rational is that a sequence will use the FIM for some period of time to consume characters that do not match the model, then the sequence will jump to the model node corresponding to the start of the fragment, use several model nodes, and then jump to the ending FIM to consume the rest of the sequence.

The probability of these jumps is set by the variables `jump_in_prob` and `jump_out_prob`, both of which have a default value of unity. That is, as in the sequence-to-sequence Smith and Waterman, there is no cost associated with jumping in and out of the model.

The file `trna1frag.seq` contains several sequences that contain part or all of TRNA1. The sequence include TRNA1 (72 bases, TRNA1Long (the complete TRNA with additional characters), Long (58 base segment) TRNA), Medium (34 base segment), Short (6 base segment TRNA1), and AAMediumA, an embedding of Medium within several segments of As to bring it to 176 characters. Additionally, the file contains several (obviously) non-TRNAs of various lengths, all of whose IDs begin with the word 'Not'.

When this file is scored using `hmmscore` with `SW` set to 0 for global alignment, the scores of the full sequence and the long fragment place them clearly as tRNAs. However, the score of the short and medium fragments are greatly penalized by the large number of delete states they must use.

```
TRNA1Long           94       -38.41      101.77
TRNA1               72       -36.23       61.80
Long                58       -17.07       61.53
AAMediumAA         176        -1.97      316.01
NotExExExExL      3504        14.64     6728.30
NotExExExLong     1752        14.69     3371.52
NotExExtraLong     876        15.57     1693.98
Medium              34        16.10       62.00
NotExtraLong       438        16.29      855.49
NotLong            109        18.23      227.07
Not                 73        19.27      159.14
MediumReverse       34        27.15       73.06
NotShort            13        40.50       65.40
ShortReverse         6        48.86       57.52
Short                6        49.14       57.80
```

When scoring is performed using the `SW` option set to 1, the following score file is generated, which places even the short fragment in the possible tRNA range.

```
TRNA1Long           94       -33.87      101.77
TRNA1               72       -32.56       61.19
Long                58       -22.43       52.11
Medium              34        -9.28       33.10
Short                6         0.84        7.72
NotShort            13         1.14       23.48
ShortReverse         6         1.28        8.15
MediumReverse       34         2.78       45.16
AAMediumAA         176         2.83      315.63
Not                 73         3.11      138.68
NotLong            109         3.51      207.66
NotExtraLong       438         4.90      838.02
NotExExtraLong     876         5.59     1677.23
NotExExExLong     1752         6.28     3355.65
NotExExExExL      3504         6.98     6712.48
```

When scoring is performed using the `SW` option set to 2, the following score file is generated, which also picks up the sequence AAMediumAA, which is a segment of a tRNA embedded within a longer sequence.

```
TRNA1Long           94       -34.48      101.16
TRNA1               72       -33.33       60.42
Long                58       -24.94       49.60
AAMediumAA         176       -13.87      298.94
Medium              34       -10.37       32.01
NotExExExLong     1752        -4.08     3345.28
NotExtraLong       438        -3.94      829.19
NotLong            109        -3.93      200.22
Not                 73        -3.92      131.66
NotExExtraLong     876        -3.90     1667.74
NotShort            13        -3.77       18.58
MediumReverse       34        -3.77       38.62
ShortReverse         6        -3.65        3.23
Short                6        -3.64        3.23
NotExExExExL      3504        -3.39     6702.11
```

Significance levels, especially for the second option, change greatly. In the first option, because sequences can start at any location (e.g., the initial FIM) and jump out of any location (e.g., also the initial FIM), no sequence will have scores worse than zero — even non-family members will have a negative NLL−NULL score. However, the significance level will be similar to that of standard scoring.

In the second option, the number of placements of a sequence to the model is essentially the number of starting points of the seqeunce plus the number of exit points once the sequence has started using the core of the model. That is, squences start in the initial FIM, may at any time jump anywhere into the model, and then jump out again latter. The effect seen in the significance level depends on both the sequence length and the model length, and for comparison between different models, must be done to the scores themselves as they are being generated. If the `adjust_score` variable is set to 1 (the default) and `SW`=2, all scores will have added to them the log of the sum of sequence and model length. If `adjust_score` is set to 2 (the default) and `SW`=1 or 2, then all scores will have added to them the log of the sequence length. In the future, the `adjust_score` parameter may be refined as we further explore the length dependence of scores. The adjustment is not sufficient for models that have internal FIMs. See Section 9.2.1 on page 59.

9.2.5 Selecting multiple domain alignments

The `hmmscore` program also can create multiple-domain alignments and score files from selected sequences. Prior to Version 2.1, this feature was called the `multdomain` program. To enable this option, the `select_mdalign` parameter is set in a manner similar to other selection parameters. See Section 9.2.3 on page 63.

For each selected sequence, the multiple domain seach procedure will locate copies of a single motif within each selected sequence. A user specified `mdNLLnull` is the criterion by which a subsequence is judged to be a match to the model; whenever an `mdNLLnull` simple null model score or lower is achieved, a match to the model has been found. Once this match is found, it is cut from the sequence and another match is looked for. The process terminates when no matches scoring better than `mdNLLnull` are found. Note that the multiple domain scoring procedure always uses the simple null model and always uses viterbi scoring. Thus, it is theoretically possible for a sequence to be selected by `hmmscore` for multiple domain seach but for no domain to be found even if `NLLnull` and `mdNLLnull` are set to the same value and `select_mdalign` is set to 1. The `mdNLLnull` threshold can most reliably be set by examining the output of `hmmscore` (using Viterbi scoring) and deciding where the cutoff between hits and misses should be.

The output is similar to that of `align2model`, except that for each match the sequence ID is modified to indicate where in the sequence the match occurred. Additionally, all letters in the sequence that are part of the match are capitalized. Unlike `align2model`, multiple domain seach sequence output does not include periods ('.') as spacers: `prettyalign` must be used to correctly space the multiple alignment.

As an example, the file `multtrna.seq` contains two sequences, each of which contains two trna motifs. By invoking `multdomain` in the following way (selecting all sequences for a multiple domain):

```
hmmscore multtrna -i testf.mod -db multtrna.seq -select_mdalign 8 -sw 2
prettyalign multtrna.mult -l90 > multtrna.pretty
```

the file `multtrna.pretty` generated is:

```
;  SAM: ../src/prettyalign v2.1.1  (Apr 24, 1998) compiled 04/27/98_12:32:27.
;  SAM:  Sequence Alignment and Modeling Software System
;  (c) 1992-1998 Regents of the University of California, Santa Cruz
;  http://www.cse.ucsc.edu/research/compbio/sam.html
;
; ------ Citations (HMMs, SAM) ------
; A. Krogh et al., Hidden Markov models in computational biology:
;   Applications to protein modeling, JMB 235:1501-1531, Feb 1994.
; R. Hughey, A. Krogh, Hidden Markov models for sequence analysis:
;   Extension and analysis of the basic method, CABIOS 12:95-107, 1996.
; ----------------------


TRNA12_6:80   ugcua...................................................
TRNA12_88:166 ugcuaggggauguagcucagugguagagcgcaugcuucgcauguaugaggccccgg
TRNA34_10:85  gcuagcgua...............................................
TRNA34_98:172 gcuagcguaggcccuguggcuagcuggucaaagcgccugucuaguaaacaggagau
                                                               10
                                                                |
TRNA12_6:80   .........................................GGGGAUGUAGCUCAG
TRNA12_88:166 guucgauccccggcaucuccaguacugcguu..........GCGGCCGUCGUCUAG
TRNA34_10:85  .........................................GGCCCUGUGGC-UAG
TRNA34_98:172 ccuggguucgaaucccagcggggccuccagcauaguuugacGGGCGAAUAGUGUCA
                    20         30        40        50        60
                     |          |         |         |         |
TRNA12_6:80   -UGG...U.AGAGCGCAUGCUUCGCAUGUAUGAGGCCCCGGGUUCGAUCCCCGGCA
TRNA12_88:166 UCUGgauU.AGGACGCUGGCCUCCCAAGCCAGCAAUCCCGGGUUCGAAUCCCGGCG
TRNA34_10:85  CUGG...UcAAAGCGCCUGUCUAGUAAACAGGAGAUCCUGGGUUCGAAUCCCAGCG
TRNA34_98:172 GCGG...G.AGCACACCAGACUUGCAAUCUGGUAGGGA-GGGUUCGAGUCCCUCUU
               70
                |
TRNA12_6:80   UCUCCAGUAcugcguugcggccgucgucuagucuggauuaggacgcuggccuccca
TRNA12_88:166 GCCGCAUCGcuu............................................
TRNA34_10:85  GGGCCUCCAgcauaguuugacgggcgaauagugucagcgggagcacaccagacuug
TRNA34_98:172 UGUCCACCAguacguagauccgcggc..............................


TRNA12_6:80   agccagcaaucccggguucgaaucccggcggccgcaucgcuu..............
TRNA12_88:166 ........................................................
TRNA34_10:85  caaucugguagggaggguucgagucccucuuuguccaccaguacguagauccgcgg
TRNA34_98:172 ........................................................


TRNA12_6:80   ..
TRNA12_88:166 ..
TRNA34_10:85  c.
TRNA34_98:172 ..
```

This file shows the matching area of the sequence within several copies of the sequence. If the variable `alignshort` is set to zero or higher, matching segments of the sequence are clipped, with `alignshort` positions shown on either side. The IDs are the same as if complete sequences are printed, corresponding to the starting and ending points of the motif within the original sequence. Depending on how large `alignshort` is, the subsequences may overlap. For example, the commands:

```
multdomain multtrnas -i testf.mod -db multtrna.seq -alignshort 3
prettyalign multtrnas.mult -l90 > multtrnas.pretty
```

produce the alignment:

```
;  SAM: ../src/prettyalign v2.1.1  (Apr 24, 1998) compiled 04/27/98_12:32:27.
;  SAM:  Sequence Alignment and Modeling Software System
;  (c) 1992-1998 Regents of the University of California, Santa Cruz
;  http://www.cse.ucsc.edu/research/compbio/sam.html
;
; ------ Citations (HMMs, SAM) ------
; A. Krogh et al., Hidden Markov models in computational biology:
;   Applications to protein modeling, JMB 235:1501-1531, Feb 1994.
; R. Hughey, A. Krogh, Hidden Markov models for sequence analysis:
;   Extension and analysis of the basic method, CABIOS 12:95-107, 1996.
; -----------------------
                        10           20          30        40        5
                         |            |           |         |
TRNA12_6:80    cuaGGGGAUGUAGCUCAG-UGG...U.AGAGCGCAUGCUUCGCAUGUAUGAGGCCC
TRNA12_88:166  guuGCGGCCGUCGUCUAGUCUGgauU.AGGACGCUGGCCUCCCAAGCCAGCAAUCC
TRNA34_10:85   guaGGCCCUGUGGC-UAGCUGG...UcAAAGCGCCUGUCUAGUAAACAGGAGAUCC
TRNA34_98:172  gacGGGCGAAUAGUGUCAGCGG...G.AGCACACCAGACUUGCAAUCUGGUAGGGA
               0        60        70
               |         |         |
TRNA12_6:80    CGGGUUCGAUCCCCGGCAUCUCCAGUAcug.
TRNA12_88:166  CGGGUUCGAAUCCCGGCGGCCGCAUCGcuu.
TRNA34_10:85   UGGGUUCGAAUCCCAGCGGGGCCUCCAgca.
TRNA34_98:172  -GGGUUCGAGUCCCUCUUUGUCCACCAgua.
```

In addition to `multtrna.mult`, the file `multtrna.mstat` is produced. It reports the NLL-NULL score for each of the motifs listed in `multtrna.mult`. In this case, `multtrna.mstat` (or `multtrnas.mstat`) looks like:

```
%  SAM: ../src/hmmscore v2.1.1  (Apr 24, 1998) compiled 04/27/98_12:32:02.
%  SAM:  Sequence Alignment and Modeling Software System
%  (c) 1992-1998 Regents of the University of California, Santa Cruz
%  http://www.cse.ucsc.edu/research/compbio/sam.html
%
% ------ Citations (HMMs, SAM) ------
% A. Krogh et al., Hidden Markov models in computational biology:
%   Applications to protein modeling, JMB 235:1501-1531, Feb 1994.
% R. Hughey, A. Krogh, Hidden Markov models for sequence analysis:
%   Extension and analysis of the basic method, CABIOS 12:95-107, 1996.
% ----------------------
%  Run start: Mon Apr 27 12:50:26 1998
%  Run name:  multtrna
%  On host:   alpha
%  In dir:    /auto/projects/compbio3/samtmp/sam/SAMBUILD/alpha/demos
%  By user:   rph
% ------------------------------------------------------------
% Motifcutoff 0.500000 mdNLLminusNULL -10.000000
% See related information in multtrna.dist
%
% Column 1: NLL-NULL using simple FIM (node 0) insert probabilities
% Column 2: the raw NLL score
% Scores sorted by column 1, best first
%
% Sequence ID    Length      Simple         Raw      X count
TRNA12_6:80         75       -32.53      191.65
TRNA34_10:85        76       -32.17      224.91
TRNA12_88:166       79       -32.13       82.59
TRNA34_98:172       75       -29.83      110.74
```

Because reliable results are only obtained if FIMs are added to the model, multiple domain searchers are best performed when `auto_fims` is set to 1.

9.2.6 Distributed scoring

Scoring a large database with `hmmscore` can take several hours on even the fastest workstation. The scoring program includes primitive support for distributed scoring. If the `segments` variable is set to an integer larger than 1, `hmmscore` assumes that that many runs of `hmmscore` are being used to score a complete database. The `segment_number` variable is used to label each segment.

These parameters might be used as follows:

```
hmmscore  test1 -i test.mod -db bigdatabase -segments 2 -segment_number 1 -sw 2&
rsh othermachine hmmscore test2 -i test.mod -db bigdatabase -segments 2 -segment_number
2 -sw 2
wait cat test1.dist test2.dist > test.dist
```

The associated parameter, `segment_size`, specifies that number of sequences that are read in at a time. At its default value of 100, two segments would produce the effect that the first 100 sequences are processed by segment 1, the next 100 be segment 2, the next 100 by segment 1, and so on. Note that workload is partitioned according to the number of sequences rather than the number of residues, some segments may take longer to complete than other segments.

9.3 addfims

The `addfims` program can be used to add Free Insertion Modules (Section 7.5) to the beginning and end of a model. The program `modifymodel` can also be used for this purpose. This is particularly useful if a model has been trained on a clipped sequence motif, and is to be used in analyzing full sequences.

For example, the file `trna10f.seq` is the same as `trna10.seq`, except that sequences 2–5 have had extraneous characters appended to one or both ends. Using the `test.mod` file previously generated, an alignment of the first 5 sequences looks like this:

```
                  10                                     20         30
                   |                                      |          |
TRNA1  ...GGGGAUGUAGCU..........CAG-.................UGG..U.AGAGCGCAUG
TRNA2X ...GCGGCCGUCGUC..........UAGUgcggccgucgucuagucUGGauU.AGGACGCUGG
TRNA3X cccGGCCCUGUGGC-..........UAGC.................UGG..UcAAAGCGCCUG
TRNA4X ...GGGCGAAUAGUGucagggcgaaUAGUgucag............CGG..G.AGCACACCAG
TRNA5X ...GCCGGGAUAGCU..........CAGU.................UGG..U.AGAGCAGAGG
               40        50        60                 70
                |         |         |                  |
TRNA1  CUUCGCAUGUAUGAGGCCCCGGGUUCGAUCCCCGGC.........AUCU---C........CA
TRNA2X CCUCCCAAGCCAGCAAUCCCGGGUUCGAAUCCCGGC.........GGCCGCACggcggccgCA
TRNA3X UCUAGUAAACAGGAGAUCCUGGGUUCGAAUCCCAGC.........GGGGCCUC........CA
TRNA4X ACUUGCAAUCUGGUAGGGA-GGGUUCGAGUCCCUCU.........UUGUCCAC........CA
TRNA5X ACUGAAAAUCCUCGUGUCACCAGUUCAAAUCUGGUUccuggcaugGUUCCUGG........CA


TRNA1  .....
TRNA2X .....
TRNA3X gggg.
TRNA4X .....
TRNA5X .....
```

This alignment is incorrect: the extra end characters do not all use end insert states. Rather, internal insert states are found to minimize the alignment cost for sequences 2X, 4X, and 5X.

The `addfims` program has an interface identical to that of `buildmodel`, though only the runname, model file (and its model, or if not present, its regularizer), and the alphabet are used:

```
addfims testf -insert test.mod
align2model testf -i testf.mod -db trna10f.seq
prettyalign testf.a2m -l90 > testf.align
```

will produce the following alignment, which correctly places the extra characters at the ends.

```
                               10          20         30       40
                                |           |          |        |
TRNA1  ................GGGGAUGUAGCUCAG-UGG...U.AGAGCGCAUGCUUCGCAUGUAG
TRNA2X gcggccgucgucuaguGCGGCCGUCGUCUAGUCUGgauU.AGGACGCUGGCCUCCCAAGCCAG
TRNA3X ccc.............GGCCCUGUGGC-UAGCUGG...UcAAAGCGCCUGUCUAGUAAACAGG
TRNA4X gggcgaauaguguca.GGGCGAAUAGUGUCAGCGG...G.AGCACACCAGACUUGCAAUCUGG
TRNA5X ................GCCGGGAUAGCUCAGUUGG...U.AGAGCAGAGGACUGAAAAUCCUC
            50        60        70
             |         |         |
TRNA1  AGGCCCCGGGUUCGAUCCCCGGCAUCU---CCA-.............
TRNA2X CAAUCCCGGGUUCGAAUCCCGGCGGCCGCACGG-cggccgca.....
TRNA3X AGAUCCUGGGUUCGAAUCCCAGCGGGGCCUCCA-gggg.........
TRNA4X UAGGGA-GGGUUCGAGUCCCUCUUUGUCCACCA-.............
TRNA5X GUGUCACCAGUUCAAAUC---UGGUUCCUGGCA-ugguuccuggca.
```

9.3.1 Training with FIMs

A similar effect can be achieved by training with free insertion modules. Suppose the file `ftrain.init` contains:

```
alphabet RNA
train trna10f.seq
seed 0
REGULARIZER
alphabet RNA
GENERIC 1.88 0.25 0.37 1.81 15.52 3.76 0.22 0.26 4.00 0.25 0.25 0.25 0.25 0.25 0.25
0.25 0.25
TYPE 0 FIM
TYPE -1 FIM
ENDMODEL
```

This file specifies a default regularizer and that both the BEGIN node and the last node are free insertion modules. Training and aligning on this file with the commands:

```
buildmodel ftrain -i ftrain.init
align2model ftrain -i ftrain.mod -db trna10f.seq
prettyalign ftrain.a2m -l90 > ftrain.align
```

will produce the alignment:

```
                                 10          20        30        40
                                  |           |         |         |
TRNA1  ..................GGGGAUGUAGCUCAGU-GG..UAGAGCGCAUGCUUCGCAUGUAUG
TRNA2X gcggccgucgucuagugc-GGCCGUCGUCUAGUCUGGauUAGGACGCUGGCCUCCCAAGCCAG
TRNA3X ccc...............GGCCCUGUGGCUAGCUGGU..CAAAGCGCCUGUCUAGUAAACAGG
TRNA4X gggcgaauagugucag..GGCGAAUAGUGUCAGCGGG..-AGCACACCAGACUUGCAAUCUGG
TRNA5X ..................GCCGGGAUAGCUCAGUUGG..UAGAGCAGAGGACUGAAAAUCCUC
TRNA6  ..................GGGGCCUUAGCUCAGCUGG..GAGAGCGCCUGCUUUGCACGCAGG
TRNA7  ..................GGGCACAUGGCGCAGUUGG..UAGCGCGCUUCCCUUGCAAGGAAG
TRNA8  g.................GGCCCGUGGCCUAGUCUGGa.UACGGCACCGGCCUUCUAAGCCGG
TRNA9  c.................GGCACGUAGCGCAGCCUGG..UAGCGCACCGUCCUGGGGUUGCGG
TRNA10 uc................-CGUCGUAGUCUAGGUGGU..UAGGAUACUCGGCUUUCACCCGAG
            50        60        70
             |         |         |
TRNA1  AGGCCCCGGGUUCGAUCCCCGGCAUCUCC-a.............
TRNA2X CAAUCCCGGGUUCGAAUCCCGGCGGCCGC-acggcggccgca..
TRNA3X AGAUCCUGGGUUCGAAUCCCAGCGGGGCC-uccagggg......
TRNA4X UAGGGA-GGGUUCGAGUCCCUCUUUGUCC-acca..........
TRNA5X GUGUCACCAGUUCAAAUCUGGUUCCUGGC-augguuccuggca.
TRNA6  AGGUCAGCGGU-CGA-CCCGCUAGGCUCC-acca..........
TRNA7  AGGUCAUCGGUUCGAUUCCGGUUGCGUCC-a.............
TRNA8  GGAUCGGGGGUUCAAAUCCCUCCGGGUCC-g.............
TRNA9  GGGUCGGAGGUUCAAAUCCUCUCGUGCCG-acca..........
TRNA10 AGA-CCCGGGUUCAAGUCCCGGCGACGGA-acca..........
```

The initial `buildmodel` run in the above case will produce an warning message along the lines of:

```
Some FIMs have non-zero match states (e.g., node 0),
a natural result of turning an existing or GENERIC node into a FIM.
These nodes are being renormalized as discussed in the SAM manual.
To avoid this message, use addfims or hmmedit when adding FIMs.
```

The purpose of this message is to apprise the user of a frequent occurrence when using FIMs. Effectively, the FIM has no match state, and therefor transitions into the FIM nodes match state, as well as the character generating probability table itself, should have zero probability. If they don't, SAM assumes that the node has not been completely turned into a FIM, and renormalizes the node for you, in the same manner in which `addfims` works. SAM will do the following:

1. Zero the probabilities in the match table.
2. Move any probability assigned to jumps into the FIMs match table to jumps into the FIMs delete node.
3. Assign both the insert to insert and the delete to insert transitions unity probability (in a FIM, probabilities exiting the insert state and exiting the delete state will sum to two rather than one).
4. To make the FIM independent of whether or not characters are generated (ie, whether or not just the delete node is used or the delete and the insert node are used), the outgoing probabilities for the FIM's insert to match and delete to match transitions are averaged, as are the FIM's insert to delete and delete to delete transitions.
5. Once the transitions have been averaged, they are normalized so that the delete to match and delete to delete transitions, as well as the insert to match and insert to delete transitions, sum to one.

Users who do not like this default behavior, in particular the averaging that ensures the FIM has the same outgoing transition cost whether or not it generates characters, may generate their own FIM with a zeroed match table, in which case only the following will be done during model normalization:

1. Zero any probability assigned to jumps into the FIMs match table.
2. Assign both the insert to insert and the delete to insert transitions unity probability.
3. Normalized the delete to match and delete to delete transitions, as well as the insert to match and insert to delete transitions, to sum to one.

Because (for historical reasons), the begin node (node 0) is composed of a non-character-producing match state and an insert state, rather than a delete state and an insert state, a node zero FIM is normalized with these two states swapped.

In all cases, a FIM is never trained.

9.4 `modelfromalign`

`Modelfromalign` takes a multiple alignment and converts it to a model. As a base model, the program starts with the default regularizer (or a specified regularizer, as for `buildmodel`), and then calculates node frequencies according to the given multiple alignment. Sequences in the alignment can be weighted according to the `alignment_weights` file, discussed in Section 8.4.

If a trustworthy hand alignment is available, this is often the best way to build a model: create one from an alignment, and then refine it using `buildmodel`. If some sections of the alignment are particularly important, it may be desirable to make them fixed nodes, as described in Section 7.4.2.

The `alignfile` parameter can also be used with `buildmodel` to specify a seed alignment. See Section 7.3 on page 31.

The `modelfromalign` program will read any `readseq` format, but has a few special interpretations. It follows the `align2model` convention that lowercase letters are insertions, hyphens are deletions, and dots are simply filler for insertions in other sequences. Additionally, the letter 'O' is converted into a FIM, following a convention used in some multiple alignment formats. If all sequences do not have the same number of uppercase letters and hyphens, then `modelfromalign` will try treating all characters as uppercase and all periods as hyphens (i.e., it will try modeling each character as a match column).

The program has one required parameter that must be set on the command line or in an inserted or `.samrc` file: the `alignfile` is the name of the file with the alignment. For example,

```
    modelfromalign trna2 -alignfile trna2.align
```

will produce a model from the `trna2.align` alignment. All `buildmodel` parameters dealing with regularization and prior libraries are used in the conversion from column frequencies to a model. Prior libraries are particularly helpful when converting a small protein alignment to a model.

If the alignment is of a motif, setting the `align_fim` variable to 1 will cause FIMs to be added to the model before printing it out.

9.5 Model manipulation

9.5.1 `drawmodel`

The `drawmodel` program is a means of generating a postscript drawing from a model, regularizer, or frequency count data (created with the `print_frequencies` option). The program is run with two arguments, the first being a model file, and the second the output file.

```
drawmodel model.mod drawing.ps
```

The program will scan the file, looking for models, regularizers, and frequency counts, and query whether or not each one should be printed, after presenting a line from the file.

There are two drawing options: overall and local. Overall is the correct option for frequency counts — the outgoing transitions of each state are drawn in different styles depending on the fraction of all sequences that use that transition. The circular delete states show the node number, while the diamond insert states show average number of characters, rounded up, inserted by each sequence that used the insert state.

In the local option, suitable for models and regularizers, transitions from a given state are drawn according to what percentage of sequences in that given state take each transition. Also, the diamond insert states have the percent of sequences which, once within the insert state, remain in the insert state. (See Figure 3 on page 16.)

The drawmodel program has several command-line options: `-landscape` to draw models in landscape format, and `-scale num`, to change the scale of the drawing to an arbitrary floating-point number. The default is portrait mode with a scale 0.235, which fits six row of 19 protein model nodes (plus one ghost node, the first model of the next row) on each page. Larger scale settings increase the size of the model nodes, and cause fewer to be placed on each line. For additional customization, the postscript file, which is readable, can be modified (e.g., to change print on a different size of paper).

If `-mod n`, `-freq n`, or `-reg n` is specified on the command line, the nth model, frequency count, or regularizer will be selected for printing, and the interactive queries on which model to print will not occur.

The `drawmodel` program requires a postscript header called `sam_header.ps`. If the program is installed correctly, the path of this header will be compiled into the `drawmodel` program. If this is not the case, or you prefer a modified header, set the `SAM_PS` environment variable to the directory name, without a trailing slash ('/'), that contains the `sam_header.ps` file.

Europeans and other people who like the A4 paper size can change '`/A4 false def`' to '`/A4 true def`' near the bottom of the header file `sam_header.ps`.

9.5.2 hmmconvert

The hmmconvert program takes a model file as input and outputs a new model file in the 'opposite' format (binary or text) from the original.

The command syntax is as follows:

```
hmmconvert runname -model_file test.mod
```

If test.mod is in text format, runname.mod will be in binary format and vice versa. If you wish to destroy your original model file, use a runname of the file's name without the .mod extension.

```
hmmconvert test -model_file test.mod
```

If test.mod is in text format, this command will create a new file called test.mod in binary format. If test.mod is binary, the new test.mod will be text.

To preserve your original modelfile, enter a new runname.

```
hmmconvert new -model_file test.mod
```

This will create a file called new.mod in the opposite format of test.mod. test.mod will be left alone.

9.5.3 modifymodel

modifymodel is a utility program for modifying SAM models. It has an interactive type in interface as well as command line arguments. This program will grow over time to meet the needs of users. It contains extensive built in help with examples and a question mark (?) will list the commands:

```
AddNodes nodeNumber count  #adds generic nodes after specified
Change subCommand...       #has sub commands, change ? to see
DeleteNodes nodeSpec       #deletes the specified nodes
DuplicateNode nodeNumber count  #copies the specified node
NewModel modelLength       #make new model from default generic
PrependToCurrentModel fileName  #reads model from file, pastes on front
PostpendToCurrentModel fileName #reads model from file, puts on end
ReadModel fileName
Show subCommand...
WriteModel [fileName]   #defaults to name of file last read in
Quit
Help
```

Most (at the moment all) operations act on the current model: which is the last read in model. The basic usage is to read a model in, do what you want to it (add nodes, change types, paste models together, etc.) and write it back out.

The currently supported functions are:

- Changing node types.
- Adding/duplicating/deleting nodes.
- Cutting models into smaller parts.
- Pasting models together.

To print all the types of the nodes (use this to see how long the model is):

```
show type all
```

To change all nodes that do not have a type into KEEP nodes,

```
change type _ K ALL
```

To change node 9 into a FIM:

```
change type + F 9
```

To shorten a model by deleting nodes 4 through the end:

```
delete 4,end
```

Below is an example of reading a model, deleting the first 20 nodes, adding a new node to the front and changing it to a FIM, then pasting it on to the end of a second model, and finally writing the modified model out:

```
read one.mod
delete 1,20
add 0 1
change type + F 1
prepend two.mod
writemodel both.mod
quit
```

9.5.4 Conversion between SAM and HMMer

SAM and HMMer are the two most widely used HMM sequences alignment and modeling systems. Even though they both employ hidden Markov models, their file formats are vastly different. Provided with this release are the conversion programs `sam2hmmer` and `hmmer2sam` that make almost complete conversion between SAM and HMMer v1.7 possible.

The conversion is "almost" because there is some information loss, and this information loss is due to the structural differences between the two systems. For those users who wish to use the conversion utilities, we discuss these issues below.

9.5.4.1 Internal HMM Structure Both SAM and HMMer use the same linear model of nodes that contain 3 states – match, delete, insert. The number of transitions per node is the same, 9. The internal difference is in how transitions between nodes are viewed. SAM views transitions INTO a node, thus each node contains the transitions FROM the previous node. HMMer is the opposite, transitions are TO the next node. HMMer's convention makes much more sense because each triple of transitions stored in a node sums to one. SAM's convention maintained only for historical reasons. This becomes a problem at the beginning and end of the model.

The begin node of HMMer has a valid transition probability distribution out of the MATCH state (in that the transition probabilities sum to one), but has the null distribution for matching characters. This means HMMer always starts in the insert state, and the begin node match state is not used. The begin node of SAM can start in either non-character-generating match or an insert. So the SAM to HMMer conversion is fine, but HMMer to SAM conversion leaves SAM's begin node match state character table zero. The net result is that SAM is forced to start in the insert state.

The end node of HMMer also has a zero match state character table.

The two programs are used as follows:

```
    hmmer2sam hmmerfile.hmmer  newsamfile.mod
    sam2hmmer newhmmerfile -i samfile.mod
```

Above, the output of `sam2hmmer` is stored in the file `newhmmerfile.hmmer`.

9.5.4.2 Extra Information in HMMer

HMMer allows other information on a per node basis. This information seems to be solvent access and consensus information. This information cannot be used in SAM at this time, and is silently dropped. A note of the global presence of this extra information is noted as a comment in the SAM model, solely for documentation purposes.

9.5.4.3 Extra Information in SAM

SAM has constructs in the model that are not supported in HMMer, and these constitute the greatest difference in the two systems. SAM has per node "types" that control learning and evaluation parameters. The problematic type is the Free Insertion Module (or FIM). This special type allows SAM to have a single node give equal cost for all characters by effectively matching one or more characters. These FIMs can be positioned anywhere in a model, but are most commonly positioned at the ends (the begin node and the node before the end node). HMMer has no such concept, and handling FIMs in a conversion is a problem. FIMs at the begin and end are are not too critical, because HMMer's system always treats the begin and end just like they are free inserts (also, HMMer re-normalizes all nodes so any strange distributions are "fixed"). FIMs in the middle of a model can NOT be converted in an exact manner. Due to the structure in SAM, some probabilities in a FIM node do not sum to one and the match state transition probabilities are all zero. During conversion, these nodes have to be "fixed" so that valid (sum to one) probability distributions are given to HMMer.

One way to fix up a FIM is to remove it: it maps to nothing in a HMMer model. This method is not too bad if the FIM does not usually match many characters, but is very bad if the FIM is used a lot. A second way is fabricate some normal cheap insert states and hope that they match the average length of the FIM. This method is hard, since there is not enough information to guess the number of insert nodes to add. (If the frequencies are present in the SAM model, then this can be done. However this information is not always present.). `sam2hmmer` utilizes the former option. The transition probabilities for the match state are computed from the exit probabilities of the FIM (a chain of FIMs in a row are removed as they were one). A SAM FIM has exits from the insert and delete states, and the previous non-FIM node has a transition from match to insert. The new transitions are computed from these numbers by breaking the match to FIM transition into two parts that have the same ratio as the exit probabilities of the FIM. This same computation is done to compute the new transitions from delete and insert.

It is not possible to record any extra information from SAM into a HMMer model as HMMer does not support comments in the file.

9.6 Plotting Programs

SAM has several plotting programs to assist in data analysis. The programs require `gnuplot` http://www.cs.dartmouth.edu/gnuplot_info.html. In general, the programs create one or more `.data` files, a `.plt` file of `gnuplot` commands, and a postscript file.

Options common to the programs include:

`plotps` — Creates a postscript file `runname.ps` using `gnuplot` if set to 1, 2, or 3. When set to 0, only a `.plt` file and one or two `.data` files are generated. A setting of 1 generates a plot in `gnuplot`'s default rectangular shape, while a setting of 2 generates a square plot. For options 1 and 2, the `.data` and `.plt` files used to create the postscript file are deleted. When set to 3, the postscript file is generated and the `.data` and `.plt` files are retained. The default setting is 1.

`plotleft` — Lowest X axis value on a graph generated by `gnuplot`. The X axis is calculated internally if `plotleft=plotright`. The default setting is 0.0.

`plotright` — Highest X axis value on a graph generated by `gnuplot`. The X axis is calculated internally if `plotleft=plotright`. The default setting is 0.0.

`plotmax` — Highest Y axis value on a graph generated by `gnuplot`. The Y axis is calculated internally if `plotmax=plotmin`. The default setting is 0.0

`plotmin` — Lowest Y axis value on a graph generated by `gnuplot`. The Y axis is calculated internally if `plotmax=plotmin`. The default setting is 0.0.

`plotline` — Creates a vertical line at this value in a graph generated by `gnuplot` if plotline is nonzero. The default setting is 0.0.

`plotnegate` — Negates the scores on a graph generated by `gnuplot` if set to 1. The default setting is 0 (off).

9.6.1 `makehist`

The `makehist` program will generate two or three files from one or two `.dist` distance file created by `hmmscore` or a `.mstat` distance file created by `multdomain`. The files can be used with the popular `gnuplot` plotting package. Histograms are most useful to discover to examine the separation between family members and non-family members during a database search. The `makehist` program requires a `NLLfile` as an argument, in addition to the runname. For example, to make a histogram from

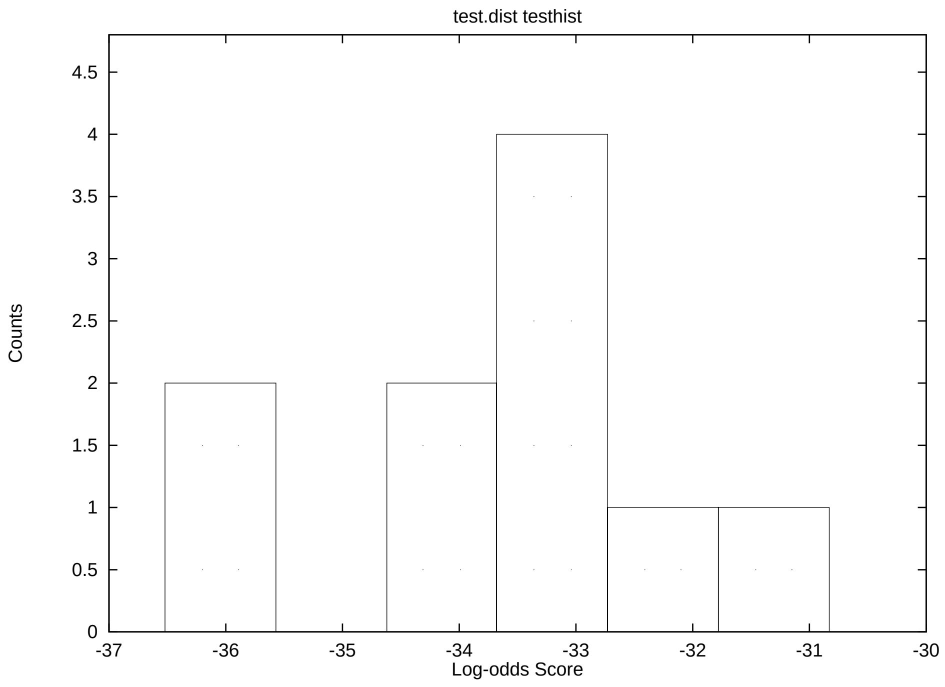


Figure 6: A histogram generated by `makehist` and `gnuplot`.

`test.dist`, the following command would be used:

```
makehist test -Nllfile test.dist
```

Then, to view the histogram, run the `gnuplot` program and enter the command `load "test.plt"` to view the results. The `test.plt` file may be modified to change the graph. If the two marked lines in the file are uncommented, a postscript file will be generated.

If a second file is specified, using the `NLLfile2` option, a second histogram is placed above the first one. The `makehist` program has an optional argument, `histbins`, which can be used to set the number of bins between which scores should be divided. The histogram in Figure 6 was generated using five bins.

9.6.2 `makeroc`

The `makeroc` program generates a postscript graph file using the `gnuplot` plotting package. It takes two `.dist` distance files created by `hmmscore` as input and plots false negatives/ false positives on the axis Score vs. Counts where Score is the NLL-null score of a sequence and Counts is the number of sequences in the files with a particular score.

To plot false negatives vs. false positives in two distance files `globins.dist` and `nonglobins.dist`, the following command would be used:

```
makeroc test -Nllfile globins.dist -Nllfile2 nonglobins.dist
```

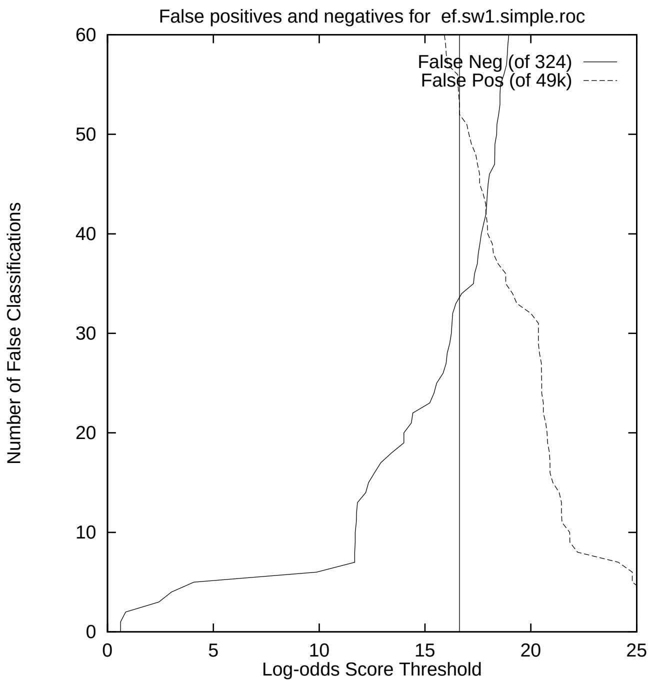


Figure 7: A plot generated by `makeroc` and `gnuplot`.

The program will output a file `test.ps` which you can view with `ghostview` or other postscript viewer.

9.6.3 `makeroc2`

The `makeroc2` program generates a postscript graph file using the `gnuplot` plotting package. It takes two `.dist` distance files created by `hmmscore` as input and plots false negative vs. false positive scores, each of which are a function of the setting of threshold score. At each integral point on, for example, the false positive axis, the value of the false negative coordinate is a linear interpolation between the two nearest integral false negative values based on the scores of the two false negative points and the false positive point in question. This means, for example, that if two negative examples score 10 and 20, and two positive examples score 19 and 21, the point (1,1.1) will be plotted for the score of 19, indicating that if the threshold is set at 19, there will be one false negative (the sequence that scores 21) and a little over one false positive (the sequence scoring 19 is 10two negative examples).

To plot false negatives vs. false positives in two distance files `globins.dist` and `nonglobins.dist`, the following command would be used:

```
makeroc2 test -Nllfile globins.dist -Nllfile2 nonglobins.dist
```

The program will output a file `test.ps` which you can view with `ghostview` or other postscript viewer.

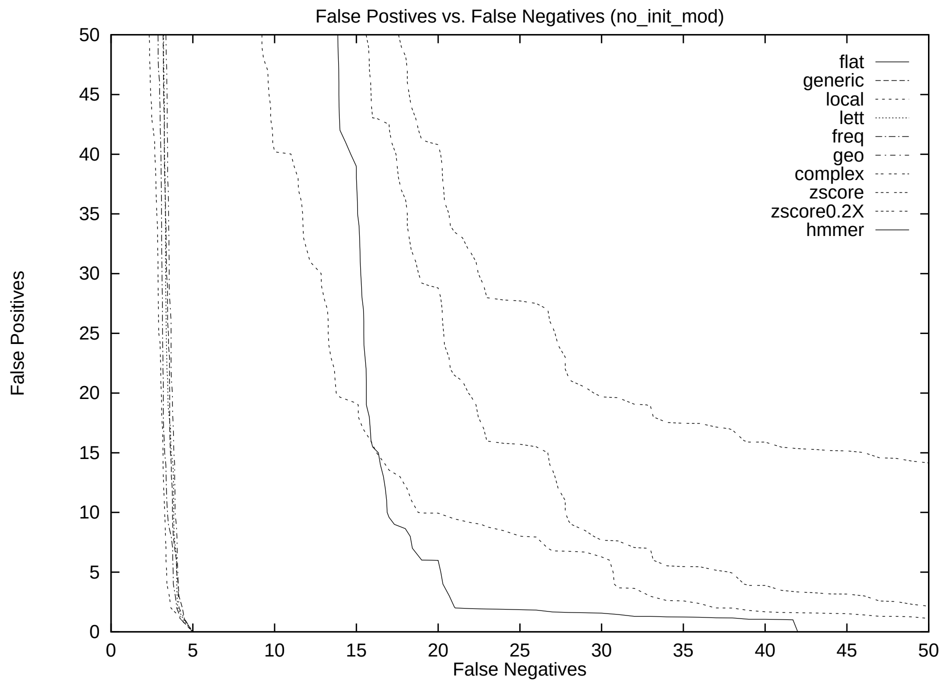


Figure 8: A plot generated by `multi_roc2.pl`, `makeroc2` and `gnuplot`.

To plot multiple pairs of distance files on the same axis, use the perl script `multi_roc2.pl`.

```
multi_roc2.pl true1.dist false1.dist title1 true2.dist false2.dist title2
true3.dist false3.dist title3 .  .  .
```

The titles appear in a legend in the upper right hand corner of the plot and identify the individual curves.

9.6.4 `makeroc3`

The third of the series is `makeroc3`. This program is given two data files, as with `makeroc2` and prints numerical information about a positive distance file and a negative distance file. With the command

```
makeroc2 test -Nllfile true.dist -Nllfile2 false.dist
```

the program will output a `test.dat` file of the form:

```
flips between x:2.000000, y:2.280622 score:-36.376999 and x:3.000000, y:2.263981
score:-36.743000
3 sequences at FP=FN
7 FN at FP=0
2 FP at FN=0
```

9.7 Sequence manipulation

The minor programs described in this section can be used to perform many helpful tasks.

9.7.1 checkseq

The checkseq program will read a sequence file and list various pieces of information about it. For example,

```
checkseq unused_runname -db trna10.seq
```

will produce the output:

```
Alphabet:  RNA
Input Format:  ig
Alignment:  no
AlignType:  unaligned
AlignColumns:  0
Num Sequences:  10
Average Length:  74.70
Max Length:  77
Min Length:  72
IDs Unique:  yes
Seq Unique:  yes
```

SAM internally recognizes several types of alignments. The check proceeds as follows: if the file is an HSSP file, it is considered an alignment. Otherwise, if the file is an a2m format file with the same number of upper case (match) and hyphen (delete) characters in each line, it is an a2m format alignment. Otherwise, if all characters have the same total number of upper case, lower case, hyphen, and period characters, it is an "all positions alignment" — if all positions are regarded as match columns, the file can be viewed as an alignment.

If the `sort` variable is set, `checkseq` will indicate whether or not all IDs are unique and whether or not all sequences are unique. This can be time consuming for large data files, in which case the user is advised to set `sort` to zero if this information is not needed.

9.7.2 permuteseq

The `permuteseq` program requires a run name and a database file. Its output will be a file of sequences that are permuted copies of the sequences in the database file. If `Nseq` is specified, the sequence file will be looped through multiple times until the requested total number of sequences are created. That is, in a file of 10 sequences, if `Nseq` is 15, the output `.seq` file will contain 10 random sequences, one for each original sequence, followed by 5 random sequences, one for each of the first 5 sequences in the file.

The `permuteseq` program can be used to verify scoring results. For example, the command:

```
permuteseq permuted -db trna10.seq -Nseq 8 -id TRNA1 -id TRNA2 -id TRNA3
hmmscore permuted -i test.mod -db permuted.seq -sw 2
```

produces the following distance file, which shows all the permuted tRNA sequences as having poor scores.

```
Rand6-TRNA3        76        -4.70       95.45
Rand8-TRNA2        76        -4.18       92.83
Rand2-TRNA2        76        -4.18       92.84
Rand5-TRNA2        76        -4.12       92.89
Rand4-TRNA1        72        -4.04       89.71
Rand3-TRNA3        76        -3.94       96.22
Rand1-TRNA1        72        -3.87       89.88
Rand7-TRNA1        72        -3.82       89.93
```

9.7.3 `sampleseqs`

The `sampleseqs` program will, given a model, randomly generate sequences according to the model's probabilities. For example,

```
sampleseqs sample -i test.mod -Nseq 5
```

will produce the following sample.seq file of synthetic tRNAs:

```
>SampleSeq0, 77 bases, EF8B285D checksum.
GCGGGGGGAGCUUACCCGUCAGACCGCUGGGUUAGCCCGGGCCGCACCCA
CCAGUUCUACUCCGGAUGGCGAUACCC
>SampleSeq1, 76 bases, B016E442 checksum.
GGGGGCAUGGCCCCGCUUGGAAGGCGCAGGCCCGCAACGGAAAGUGGUCC
CGGUUGUAAACCGCUCGACGCAACCA
>SampleSeq2, 72 bases, CF9F2C7F checksum.
GUAGAGGUAGCUCAGUUUGCAGCGCGGCAGCCUGCCAAUGUGGGGAUCAG
GGGUUCAUAUCGAAGUCCUCCA
>SampleSeq3, 70 bases, B077D3EB checksum.
GCAGAUCAGUUGGUAGGACACCUGAUCACAAUCGCCGGGACCGAUGGGUC
GAGUCCCCGCGGCGGGGGCA
>SampleSeq4, 78 bases, 5AC70EDF checksum.
UGCGCCAUAGUAUAGCUGGGUGCGCGGAUGCCUAGGACCUAGAGAGCUCA
CCGGUUCCAACCCUGGAACUCGGGACCA
```

Run with no arguments for a usage message.

9.7.4 `sortseq`

The `sortseq` program takes as input a database file(s) of sequences (`*.sel` or `*.seq`) and a distance file (`*.dist`) of NLL-NULL scores generated by `hmmscore`. It outputs a file of sequences in the same order as they occur in the distance file. `Sortseq` provides an enhancement to `hmmscore`. It outputs a `.seq` file which contains the actual sequences in sorted order while `hmmscore` outputs a file only containing sorted sequence id's and scores.

For example:

```
sortseq sort -db trna10.seq -NLLfile test.dist
```

will produce the following sort.seq file of sequences:

```
;TRNA7, 73 bases, 188CBC35 checksum.
TRNA7
GGGCACAUGGCGCAGUUGGUAGCGCGCUUCCCUUGCAAGGAAGAGGUCAU
CGGUUCGAUUCCGGUUGCGUCCA1
;TRNA3, 76 bases, E054B7BB checksum.
TRNA3
GGCCCUGUGGCUAGCUGGUCAAAGCGCCUGUCUAGUAAACAGGAGAUCCU
GGGUUCGAAUCCCAGCGGGGCCUCCA1
;TRNA8, 75 bases, A22B0856 checksum.
TRNA8
GGGCCCGUGGCCUAGUCUGGAUACGGCACCGGCCUUCUAAGCCGGGGAUC
GGGGGUUCAAAUCCCUCCGGGUCCG1
;TRNA2, 76 bases, 9DFDE3C0 checksum.
TRNA2
GCGGCCGUCGUCUAGUCUGGAUUAGGACGCUGGCCUCCCAAGCCAGCAAU
CCCGGGUUCGAAUCCCGGCGGCCGCA1
;TRNA9, 77 bases, 71415595 checksum.
TRNA9
CGGCACGUAGCGCAGCCUGGUAGCGCACCGUCCUGGGGUUGCGGGGGUCG
GAGGUUCAAAUCCUCUCGUGCCGACCA1
;TRNA1, 72 bases, D7704609 checksum.
TRNA1
GGGGAUGUAGCUCAGUGGUAGAGCGCAUGCUUCGCAUGUAUGAGGCCCCG
GGUUCGAUCCCCGGCAUCUCCA1
;TRNA5, 73 bases, C13872C8 checksum.
TRNA5
GCCGGGAUAGCUCAGUUGGUAGAGCAGAGGACUGAAAAUCCUCGUGUCAC
CAGUUCAAAUCUGGUUCCUGGCA1
;TRNA6, 74 bases, 2BEB10D6 checksum.
TRNA6
GGGGCCUUAGCUCAGCUGGGAGAGCGCCUGCUUUGCACGCAGGAGGUCAG
CGGUCGACCCGCUAGGCUCCACCA1
;TRNA4, 75 bases, B11509BB checksum.
TRNA4
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAG
GGUUCGAGUCCCUCUUUGUCCACCA1
;TRNA10, 76 bases, B2090954 checksum.
TRNA10
UCCGUCGUAGUCUAGGUGGUUAGGAUACUCGGCUUUCACCCGAGAGACCC
GGGUUCAAGUCCCGGCGACGGAACCA1
```

9.7.5 uniqueseq

The uniqueseq program takes as input a database file(s) of sequences (`*.sel` or `*.seq`). It sorts the sequences and outputs a file containing every sequence with a unique ID (i.e. no dupes are copied to the output file). It also outputs messages to the user informing of the following conditions in the database file: same id for different sequences; same sequence for different ids; and duplicate id found. If the optional `-train mytrainfile.seq` is used, the program checks the sequences in the `trainfile` and writes any sequence not in the database file to the output file. In this case, the following user information messages are output: training sequence (ID number) is in database with different ID (ID number); training sequence (ID number) is in database with same ID; training sequence (ID number) not in database (note: the program currently checks only one training file).

For example:

```
    uniqueseq unique -db testme.seq -train trna10.seq
```

will produce `unique.seq`.

10 System installation

The SAM system runs on a variety of Unix workstations (we have checked installation on workstations including DEC DECstation and Alpha, HP 715, IBM RS6000, SGI Onyx Reality Engine, Sun Sparc, Intel Pentium with the Linux operating system. The example `buildmodel` run required 50 seconds (user) on an RS6000/350, 45 seconds on a Sun 4/50 with 16 MBytes of memory, 42 seconds on a DECStation 5000/240 with 128 MBytes of memory, 33 seconds on the HP 715/50, 14 seconds on both a DEC Alpha 3000/400 and the Reality Engine.

The distribution includes an `INSTALL` file that discusses installation procedures.

If compilation did not work, you may need to try defining `-DDUM` as part of the command line. If that does not work, please send email to sam-info@cse.ucsc.edu for help.

10.1 Runtime statistics

At the end of each run of `buildmodel`, a line of statistics is printed out, such as the line

```
    -218.36  -217.00  -217.68   0.96  22 0 149
```

mentioned in Section 3. These numbers are quite useful for quick comparison of results when, for example, running the program many times using a shell script. The numbers are: minimum NLL-NULL score, maximum score, average score, sample deviation of scores, number of reestimates, number of surgeries, and the length of the final model. In the above case, the scores are for the training set: if a test set were specified (Section 6.3), the minimum, maximum, average, and sample deviation for the test set would be reported after the model length, followed by the ratio of the average test set score to the average training set score (ideally, this value should be close to unity — larger values may indicate overfitting of the model to the training set).

10.2 Reducing runtime

Training a model can be a be a time-consuming process. Each reestimation cycles through all sequences in the training set, performing a dynamic programming algorithm with operations proportional to the product of the total number of characters in the training set and the length of the model. Then, there can be large numbers of reestimations, making some runs take overnight.

Shorter execution times (and possibly worse models or alignments) can be had in several: a hard limit can be placed on the number of `reestimates`, or the `stopcriterion` can be increased, though both of these can decrease model quality. Similarly, the number of surgeries can be reduced. One of the most effective ways to reduce runtime is to simply reduce the number of sequences in the training set. A small, well-chosen training set, in which close homologues have been eliminated, can produce better models than a larger, random training set.

If a run seems to be taking too long, it is possible to tell SAM to save the next model as a prelude to killing the program. The two UNIX signals, SIGUSR1 and SIGUSR2, can be used to toggle the `print_surg_models` and `print_all_models` variables. In the first case, models are printed after each surgery procedure, and in the second, after each reestimation cycle.

10.3 Future Features

There are many future features we would like to include in SAM. The following list will also point out some of the things you currently cannot do using the system. The items are of varying difficulty.

- Elimination of the Zscoring options and reduction of the number of parameters.
- Graphical and command line tools for modifying models, including changing node types and probability tables.
- Position-specific regularizer strengths to extend the special node concepts between entirely fixed and entirely free.
- Model learning and combining using genetic algorithms.
- A coarse-grain parallel implementation.
- A version that can run on the Kestrel programmable parallel processor:

  http://www.cse.ucsc.edu/research/kestrel

11 Parameter descriptions

This section alphabetically explains all the parameters that can be specified in an init file. Where appropriate, the type of the parameter and any default value is listed. The default values are automatically used by the program if the user does not specify any alternative setting. The `dump_parameters` option can be used to verify the default values. See Section 5 on page 21.

The programs `drawmodel` and `prettyalign` do not use parameter files.

The parameter reading routines will accept variations in capitalization and the presence or absence of underscores.

SAM supports reading compressed input files. If any of the file name arguments to the options end in a `.gz` or `.Z` extension. SAM will read the files using the approriate decompression program. If an input file does not exist and does not have a `.gz` or `.Z` extension is not found, SAM will try to read from a compressed file with one of these extensions.

`a2mdots` <0 or 1> [1]: By default (1), `align2model` will place dots in the sequence alignment to fill space need for other sequences' insertions. If set to 0, these dots are not printed. See Section 9.1 on page 50.

`adjust_score` <0, 1 or 2> [2]: If set, scores are adjusted appropriately according to the `SW` method, and model and sequence length, so that final scores are somewhat independent of sequence length and model length. Currently, this only applies to fully local scoring, in which case, the log of the sum of the model and sequence lengths is added to each score. This parameter is used by `hmmscore` and by `multdomain`. See Section 9.2.4 on page 64.

`alignfile` <string>: A file containing an alignment of sequences for use with `modelfromalign` or as an initial model for `buildmodel`. See Section 9.4 on page 73. See Section 9.2 on page 56.

`align_fim` <0 or 1> [0]: Add FIMs to the ends of a model generated by `modelfromalign` or an alignfile in `buildmodel`. See Section 9.4 on page 73.

`alignment_weights` <string>: A file containing sequence weights for alignments used to form initial models with `buildmodel` or models with `modelfromalign`. See Section 8.4 on page 44.

`alignshort` <integer> [-1]: When less than 0 (default), multiple domain search produces an alignment file that copies the entire sequence for each copy of the domain occurring within the sequence. When 0, only the region matching the model is printed. When greater than zero, that many characters to the left and the right of the domain are also printed to the file. In both cases, sequence IDs in the new file can be used to locate where the `hmmscore` found copies of the model. See Section 9.2.5 on page 66.

`alphabet` <string> [protein]: This system supports 3 alphabets: DNA, RNA or protein. The protein alphabet is the default, and does not need to be specified. The abbreviation `a` may be used in place of `alphabet`. If unset, the first `train`, `test`, or `db` file is checked to see if the `alphabet` can be determined from the data. See Section 6.1 on page 22.

`alphabet_def` <string>: The `alphabet_def` variable can be used to define an alphabet of 2 to 25 letters plus a (require) all-matching wildcard character. In the quoted string argument, both an alphabet name and the list of characters, with the wildcard last, must be specified. See Section 6.1.1 on page 23.

`anneal_length` <float> [0.8]: Indicates the speed with which noise should be decreased to zero. If greater than 1, decrease linearly over `anneal_length` reestimates. If less than one, decrease exponentially. See Section 8.1 on page 40.

`anneal_noise` <float> [5]: Amount of noise to add to the model (decreased linearly or exponentially according to `anneal_length`. See Section 8.1 on page 40.

`auto_fim` <0 or 1> [1]: Cause `hmmscore` to automatically add FIMs to the model before scoring when simple or complex null model subtraction is used or fully local scoring (`SW` is 2) is used. See Section 9.2 on page 56.

`binary_output` <0 or 1> [0]: Tells `addfims`, `buildmodel`, `makecounts`, `modelfromalign`, and `modifymodel` to write models in text format if set to 0 or a binary format if set to 1. Default is currently text or 0. See Section 7.4.3 on page 36.

`calc_smooth` <0 or 1> [0]: Tells `hmmscore` whether or not to calculate a smooth curve and write it to `smooth_file`, or its default (`runname.smooth`). See Section 9.2 on page 56.

`cutinsert` <float> [0.5]: If this fraction of sequences use an insert state, surgery will replaced with one or more match states. See Section 8.2 on page 41.

`cutmatch` <float> [0.5]: When fewer than this fraction of sequences use a match state, surgery will delete the state. See Section 8.2 on page 41.

`db` <string>: A file containing sequences that are to be scored against a model in `hmmscore` or aligned to a model in `align2model`. Multiple instances of the `db` variable add to the list of database files, rather than replacing the previous `db` file name. See Section 9.2 on page 56.

`del_jump_conf` <float> [1.0]: Confidence in the regularizer for transitions leaving a delete state. The regularizer's transition values are multiplied by this number. See Section 7.1 on page 26.

`dump_parameters` <0, 1, or 2> [0]: Normally, only modified parameters are printed to the output file. If this is set to 1, all parameters are printed. If 2, and specified alone on the command line, `buildmodel` znd `align2model` will dump parameters and exit. Because in this case an alphabet is not specified and a regularizer not created, a setting of 2 will *not* reveal the default regularizer. See Section 7.4 on page 31.

`family_base_file` <string>: If non-null, and `sequence_weights` and `family_specific` are specified, initial models are read in from the files whose names are created by appending `.i.mod`, where `i` is an integer corresponding to the family number. For example, if there are three families and the base name is `test`, the family models will be read in from `test.0.mod`, `test.1.mod`, and `test.2.mod`. The first model in the file (of any type, including MODEL, REGULARIZER, NULLMODEL, and FREQUENCIES) is used. An error will result if the models are of different lengths. See Section 8.4 on page 44.

`FIM_method_train` <0, 1, 2, 3, 5, 6> [-1]: During the model building process, one may employ an initial model that contains FIMs. The table probabilities can readily be changed to reflect different distributions. Negative values only cause changes to the tables when models are created by the program, rather than being read in. The default setting of -1 uses the letter frequencies in the training set when generating new models. See Section 7.6 on page 38.

`FIM_method_score` <0, 1, 2, 3, 5, 6> [-6]: Similar to `FIM_method_train`, except that the insert probabilities in the FIMs are changed before sequences are scored against the model. Negative values only cause changes to the tables when models are created by the program, rather than being read in. The default method of -6 uses the geometric average of match state probabilities. See Section 9.2.1 on page 59.

`fimstrength` <float> [1.0]: A factor by which to multiply the FIM letter emission probabilities. If set to 2.0, for example, each letter will have twice the probability of being generated as in the normalized insert state. This can be used to encourage the use of FIMs. The value is also applied to simple null models. When set to a value less than 0, the absolute value of `fimstrength` is applied to all insert states, FIM or otherwise. See Section 7.5 on page 37.

`fracinsert` <float> [1.0]: When an insert state is being replaced, surgery will replace it with the average number or characters generated by the insert state multiplied by this number. See Section 8.2 on page 41.

`FREQUENCIES`: A model structure that has frequency counts rather than probabilities. Output by `buildmodel` if the `print_frequencies` parameter is set to 1. The `drawmodel` program is the only program that can use frequencies as input. See Section 7.4 on page 31..

`histbins` <integer> [10]: Number of bins used by the `makehist` program. See Section 9.6.1 on page 78.

`id` <string>: A sequence identifier, used to restrict `align2model` or `hmmscore` to only considering specific sequences. Multiple occurrences of the `id` parameter are added to the list of sequence identifiers, rather than replacing the value of `id`.

`initial_noise` <float> [-1.0]: When greater than zero, amount of noise to add for the first iteration. See Section 8.1 on page 40.

`ins_jump_conf` <float> [1.0]: Confidence in the regularizer for transitions leaving an insert state. The regularizer's transition values are multiplied by this number. See Section 7.1 on page 26.

`insconf` <float> [10000]: Confidence in the regularizer for character probabilities in an insert state. The high default means that the regularizer will overpower the actual counts determined by aligning sequences to the model. The regularizer's character insert values are multiplied by this number. See Section 7.1 on page 26.

`insert` <string>: Insert another parameter file. The single character `i` may be used in place of `insert`. See Section 5 on page 21.

`insert_file_dna` <string>: Insert another parameter file if the current alphabet has been set to DNA. This is particularly useful for alphabet-specific regularizers. See Section 5 on page 21.

`insert_file_protein` <string>: Insert another parameter file if the current alphabet has been set to protein. This is particularly useful for alphabet-specific regularizers. See Section 5 on page 21.

`insert_file_rna` <string>: Insert another parameter file if the current alphabet has been set to RNA. This is particularly useful for alphabet-specific regularizers. See Section 5 on page 21.

`Insert_method_train` <0, 1, 2, 3, 5> [-1]: Similar to `FIM_method_train` except that the insert probabilities are changed in the nodes that are not FIMs. Negative values only cause changes to the tables when models are created by the program, rather than being read in. The default method -1 uses the letter frequencies in the training set when generating models. If the model or regularizer includes a GENERIC node, then its match and insert tables are also filled in with these values. See Section 7.6 on page 38.

`Insert_method_score` <0, 1, 2, 3, 5, 6> [0]: Similar to `FIM_method_score` except that the insert probabilities are changed in the nodes that are not FIMs. Negative values only cause changes to the tables when models are created by the program, rather than being read in. The default method 0 is to not change the insert tables during scoring. See Section 9.2.1 on page 59.

`internal_weight` <0, 1, 2> [1]: Use internal maximum discrimination sequence weighting. Automatically turned off if not explicitly set and external weights are used. See Section 8.4.3 on page 47.

`jump_in_prob` <float> [1.0]: The probability cost of jumping into the center of the model when the `SW` option is set. See Section 9.2.4 on page 64.

`jump_out_prob` <float> [1.0]: The probability cost of jumping out of the center of the model when the `SW` option is set. See Section 9.2.4 on page 64.

`mainline_cutoff` <float> [0.5]: Changing this value will set both `cutmatch` and `cutinsert` to the new value. See Section 8.2 on page 41.

`many_files` <0,1> [0]: When zero, all the output of `buildmodel` is sent to the `.mod` file. When set, the probability model, frequency model, and run statistics are printed to different files. See Section 4 on page 19.

`match_jump_conf` <float> [1.0]: Confidence in the regularizer for transitions leaving a match state. The regularizer's transition values are multiplied by this number. See Section 7.1 on page 26.

`matchconf` <float> [1.0]: Confidence in the regularizer for character probabilities in a match state. The regularizer's character match values are multiplied by this number. This variable is ignored if a prior library is used. See Section 7.1 on page 26.

`maxinserts` <integer> [100]: In buildmodel it, the maximum number of states inserted after any node by the surgery. See Section 8.2 on page 41.

`maxmem` <integer> [0]: Maximum size of dynamic programming array to use for training and alignment. See Grice, Hughey, and Speck, and Tarnas and Hughey CABIOS papers for more information on the algorithm used. Depending on system configuration, performance may increase with higher values. If set to zero (the default), SAM will always use the smallest possible amount of space.

`maxmodlen` <integer> [0]: When starting with multiple, randomly generated models, the longest model to use. If set to 0 (the default), the value is calculated as 10% above the average sequence length when needed. See Section 7.4.1 on page 34.

`minmodlen` <integer> [0]: When starting with multiple, randomly generated models, the shortest model to use. If set to 0 (the default), the value is calculated as 10% below the average sequence length when needed. See Section 7.4.1 on page 34. See Section 7.4.1 on page 34.

`MODEL`: Specify an initial model. See Section 7.4 on page 31..

`model_abort_length` <integer> [10000]: In `buildmodel`, if the initial model length is greater than this number, an error message is printed and the program is aborted. This is to avoid giant models that will never complete training because of their memory or execution time requirements.

`model_file` <string>: If non-null, this file is read for an initial model. The first model in the file (of any type, including MODEL, REGULARIZER, NULLMODEL, and FREQUENCIES) is used. This will override any models present in inserted files. See Section 4 on page 19.

`modellength` <integer> [-1]: When greater than 0, sets the model length to a specific value in `buildmodel`. (overridden if a model or regularizer without a GENERIC node is present). If equal to 0 and `maxmodlen` is less than 1, all model lengths are set to the average length of the training sequences. If less than 0, model length(s) are set to a random value between `minmodlen` and `maxmodlen` according to `seed`. These two bounds will default to 90% and 110% of average sequence length if `maxmodlen` is less than 1. See Section 7.4.1 on page 34.

`Motifcutoff` <float> [0.5]: In mutiple motif search, fragments which are smaller than this fraction of the model length are not considered for further processing. Further, processing stops if a fragment of length less than the square of `Motifcutoff` is the best match (this is needed when using SW scoring with weak thresholds). See Section 9.2.5 on page 66.

`mdNLLnull` <float> [-10.0]: Criterion by which subsequences are judged to be matches to a single motif (model) during a multiple domain alignment. All occurrences for which NLL-NULL is better than the specified value are considered matches. See Section 9.2.5 on page 66.

`NLLnull` <float> [-10.0]**:** If a selection variable is odd, this value is checked against a sequence's simple null model score. See Section 9.2 on page 56.

`NLLcomplex` <float> [-10.0]**:** If a selection variable includes 2 in its binary representation, this value is checked against a sequence's complex, user, or reverse sequence null model score. See Section 9.2 on page 56.

`NLLFile` <string>**:** File with already-calculated sequence distances for use with `hmmscore`, `makehist`, `makeroc` or `makeroc2`. See Section 9.2 on page 56. and See Section 9.6 on page 78..

`NLLFile2` <string>**:** A second file with already-calculated sequence distances for use with `makehist`, `makeroc` or `makeroc2`. See Section 9.6 on page 78.

`Nmodels` <integer> [3]**:** Multiple initial models can be trained simultaneously, with the best one being used for surgery and further training. See Section 7.4.1 on page 34.

`NscoreSeq` <integer> [100000]**:** Maximum number of sequences to be read by the `hmmscore` or `align2model` program.

`Nseq` <integer> [10000]**:** Maximum number of sequences to be read from any of the up to four sequence files or a database files in `buildmodel`. See Section 6.3 on page 25.

`nsurgery` <integer> [3]**:** Maximum number of surgeries to perform. Each surgery will result in a full EM cycle until `stopcriterion` or reestimates is reached.

`Ntrain` <integer> [0]**:** Number of sequences to train on. If zero, all sequences that were read from the files `train` and `train2` (up to a limit of `Nseq` per file) form the training set. If `Ntrain` is greater than than the number of sequences read in from the files `train`, `train2`, `test`, and `test2`, all sequences are used for training. If `Ntrain` is less than the total number of sequences read in from the four files, all the sequences are randomly partitioned (using `trainseed`) into the training set with `Ntrain` sequences, and of the remaining sequences (i.e., whether or not a sequence occured in a training file or a test file is ignored). See Section 6.3 on page 25.

`nucleotide_prior` <string>**:** The prior library to use if the RNA or DNA sequences are being modelled and `prior_library` has not been set. See Section 7.1 on page 26.

`NULLMODEL`**:** Identifies a user defined null model in a model file. The parameter `subtract_null` must be set to 3 to use this null model. See Section 9.2 on page 56.

`nullmodel_file` <string>**:** If non-null, this file is read for a complex null model. The first model in the file (of any type, including MODEL, REGULARIZER, NULLMODEL, and FREQUENCIES) is used. This will override any null models present in inserted files. To use this null model, `subtract_null` must be set to 3. See Section 4 on page 19.

`plotleft` <float> [0.0]**:** Lowest X axis value on a graph generated by `gnuplot`. The X axis is calculated internally if `plotleft=plotright`. Used in conjunction with `makehist`, `makeroc` and `makeroc2`. See Section 9.6 on page 78.

`plotline` <float> [0.0]**:** Creates a vertical line at this value in a graph generated by `gnuplot` if plotline is nonzero. Used in conjunction with `makehist`, `makeroc` and `makeroc2`. See Section 9.6 on page 78.

`plotmax` <float> [0]**:** Highest Y axis value on a graph generated by `gnuplot`. The Y axis is calculated internally if `plotmax=plotmin`. Used in conjunction with `makehist`, `makeroc` and `makeroc2`. See Section 9.6 on page 78.

`plotmin` <float> [0]: Lowest Y axis value on a graph generated by `gnuplot`. The Y axis is calculated internally if `plotmax=plotmin`. Used in conjunction with `makehist`, `makeroc` and `makeroc2`. See Section 9.6 on page 78.

`plotnegate` <int> [0]: Negates the scores on a graph generated by `gnuplot` if set to 1. Used in conjunction with `makehist`, `makeroc` and `makeroc2`. See Section 9.6 on page 78.

`plotps` <int> [1]: Creates a postscript file runname.ps if set to 1. When set to 0, only a .plt file is generated. A square plot postscript file is generated for a setting of 2. For options 1 and 2, the .data and .plt files used to create the postscript file are deleted. When set to 3, the postscript file is generated and the .data and .plt files are retained. Used in conjunction with `makehist`, `makeroc` and `makeroc2`. See Section 9.6 on page 78.

`plotright` <float> [0.0]: Highest X axis value on a graph generated by `gnuplot`. The X axis is calculated internally if `plotleft=plotright`. Used in conjunction with `makehist`, `makeroc` and `makeroc2`. See Section 9.6 on page 78.

`print_all_models` <0 or 1> [0]: When set, models are printed after each iteration of the forward-backward procedure. Models are printed to files of the form `runname.a.mrrr.mod`, where 'mrrr' is the catenation of the number of the model (or 1 if only one model is being estimated at a time) and the reestimate number. This variable can be toggled at runtime by sending a `SIGUSR2` signal to the program, providing a means to look at intermediate results while the program is running or checkpointing a program run.

`print_all_weights` <0 or 1> [0]: When set, a weight output file is generated after each iteration of the forward-backward procedure. Weights are printed to files of the form runname1.weightoutput, where '1' is the number of the iteration.

`print_frequencies` <0 or 1> [0]: If this option is set, the frequency counts for each state will be printed as well as the model.

`print_surg_models` <0 or 1> [0]: When set, models are printed after each surgery (surgery occurs after a sequence of EM reestimates). Models are printed to files of the form `runname.s.rr.mod`, where 'rrr' is the reestimation index for the run. When surgery is used, a single winning model is automatically selected after the first EM reestimation loop if multiple initial models are used. This variable can be toggled at runtime by sending a `SIGUSR1` signal to the program.

`prior_library` <string>: When set, use Dirichlet mixture priors to regularizer the models. Transition costs and insert states are still regularized by the default (or specified) regularizer, but match states are regularized with Dirichlet mixtures. The `matchconf` variable is ignored if a prior library is used, in favor of the `prior_weight` variable. If `prior_library` is not set and `protein_prior` or `nucleotide_prior` is set, the indicated prior library is used. See Section 7.1 on page 26.

`prior_weight` <float> [1.0]: Weight of the prior library, if it is used. See Section 7.1 on page 26.

`protein_prior` <string> [recode1.20comp]: The prior library to use if the proteins are being modelled and `prior_library` has not been set. See Section 7.1 on page 26.

`randomize` <integer> [50]: Determines how noise is added to the model. See Section 8.1 on page 40.

`read_smooth` <0 or 1> [0]: Tells `hmmscore` whether or not to read a smooth curve from `smooth_file`, or its default (`runname.smooth`). See Section 9.2 on page 56.

`reestimates` <integer> [40]: Maximum number of reestimates to perform after a surgery. Generally, this should be set higher than the number of iterations that have noise. See Section 8 on page 39.

`reglength` <integer> [-1]: Similar to `modellength`, sets the length of the regularizer. Usually not needed. See Section 7.4.1 on page 34.

`REGULARIZER`: Specify an initial regularizer. See Section 7.4 on page 31.

`regularizer_file` <string>: If non-null, this file is read for a single-component regularizer. The first model in the file (of any type, including MODEL, REGULARIZER, NULLMODEL, and FREQUENCIES) is used. This will override any regularizers present in inserted files. See Section 4 on page 19.

`rerun` <integer> [-1]: The program optimizes Nmodels models until the first 'surgery', and then continues with the best one. Sometimes it is interesting to see how the second best would have done. If the second best is number 4 (starting from 0!), a setting this parameter to 4 would optimize that model. Models can also be accessed using one `print_all_models`.

`retrain_noise_scale` <float> [0.1]: If an initial model or alignment is passed to buildmodel, `initial_noise` (or `anneal_noise` if `initial_noise is unspecified`)is scaled by this multiplier, which must be between 0.0 and 1.0. See Section 8.1 on page 40.

`seed` <integer> [-1]: Random seed for noise generation and for selection of initial model lengths if `modellength` is less than one. The default value causes the process's pid to be used, which will then be printed to the output file to enable replication of results.

`segments` <integer> [1]: Number of segments `hmmscore` should logically split database into. Segmentation is based on number of sequences. See Section 9.2.6 on page 69.

`segment_number` <integer> [1]: Segment number among segments. See Section 9.2.6 on page 69.

`segment_size` <integer> [100]: Number of sequences read in at a time and given to one of the segments. See Section 9.2.6 on page 69.

`select_align` <integer> [0]: Tells `hmmscore` what selection criteria should be used for placing aligned sequences into the file `runname.a2m`. If 0, no sequences are selected; if 1, sequences are selected according to their simple null model scores and `NLLNull`; if 2, sequences are selected according to their complex, user, or reverse sequence null model score and `NLLcomplex`; if 4, sequences are selected according to their Z-scores and `Zmax`; if 8, all sequences are selected. Selection criteria can be combined: 3 requires sequences to score better than `NLLnull` with the simple null model and `NLLcomplex` with the complex null model. Negative numbers indicate that sequences that do not pass the corresponding positive test should be selected. See Section 9.2 on page 56.

`select_mdalign` <integer> [0]: Tells `hmmscore` what selection criteria should be used for placing a multiple domain sequence alignment in the file `runname.mult` with scores in `runname.mstat`. Functions as with `select_align`. Only sequences that pass the selection criteria and have a Viterbi alignment simple null model score that is better than `mdNLLnull` will appear in the files. See Section 9.2 on page 56.

`select_score` <integer> [8]: Tells `hmmscore` what selection criteria should be used for listing sequence scores in the file `runname.dist`. Functions as with `select_align`. See Section 9.2 on page 56.

`select_seq` <integer> [0]: Tells `hmmscore` what selection criteria should be used for placing sequences in the file `runname.sel`. Functions as with `select_align` See Section 9.2 on page 56.

`sequence_models` <float> [0.0]: Build initial models from randomly-selected sequences in the training set when greater than zero. Value indicates the weight the sequence should have when combined with the regularizer. See Section 7.3 on page 31.

`sequence_warning` <integer> [0]: Primarily for debugging. Set to −1 to print out all sequences in which a 'wrong' letter was found, or to −2 to print out all sequences.

`sequence_weights` <string>: File to read for sequence weights. See Section 8.4 on page 44.

`simple_threshold` <integer> [0]: Complex, user, and reverse sequence scores will not be calculated by `hmmscore` unless the simple null model score is less than this number. Set to 10000 to require all scores to be calculated. See Section 9.2.1 on page 59.

`smoothfile` <string>: Name of file for input or output of data used in calculation of Z-scores. See Section 9.2 on page 56.

`sort` <integer> [1]: Indicates whether or not sequence scores should be sorted by `hmmscore`. With a value of 1, sequences are sorted by column 1 (simple null model score). With a value of 2, sequences are sorted by column 2 (other null model selections; see `subtract_null`). With a value of 3, sequences are sorted by Z-score if available or by column 1. When negative, scores are sorted in reverse order, worst first. When 0, scores are not sorted. `Sort` also indicates whether or not `uniqueseq` should sort sequence IDs and sequences to check for uniqueness. See Section 9.2 on page 56 and Section 9.7.5 on page 84.

`stopcriterion` <float> [0.1]: The reestimation loop will stop whenever the improvement in the NLL score is less than this number (provided noise is less than 10% of its original value for that iteration), or when the maximum number of `reestimates` is reached. See Section 8 on page 39.

`subtract_null` <integer> [1]: In `hmmscore` and other programs, decides the type of null model to be used. In score files, this will be the second score column (the first is always the simple null model). When set to 0, raw scores are reported in the second column. Setting to 1 provides simple null model scores; to 2, complex null model scores; to 3, user's input null model; and to 4, the reverse sequence null model.

`surgery_noise_scale` <float> [0.1]: After the first surgery, `anneal_noise` is scaled by this multiplier, which must be between 0.0 and 1.0. See Section 8.1 on page 40.

`SW` <integer> [0]: When set to 1 , `hmmscore` uses submodel to sequence scoring. When set to 2 , `hmmscore` uses submodel to subsequence scoring. Can also be used with `align2model` but not currently with `buildmodel`. Similar to the Smith and Waterman method. See Section 9.2.4 on page 64.

`test` <string>: A file to read test sequences from. See Section 6.3 on page 25.

`test2` <string>: A second file to read test sequences from. See Section 6.3 on page 25.

`trainseed` <integer> [-1]: Random seed for partitioning the sequences into the test set and the training set. The default value causes the process's pid to be used, which will then be printed to the output file to enable replication of results. See Section 6.3 on page 25.

`train` <string>: A file to read training sequences from. See Section 6.3 on page 25.

`train2` <string>: A second file to read training sequences from. See Section 6.3 on page 25.

`train_reset_inserts` <0,1,2,3, or 6> [6]: At the end of `buildmodel` training, all insert and FIM character tables are set according to this variable, which takes on the same meanings as `FIM_method_train`. The default setting is to set all insert and FIM tables to the normalized geometric average of the match state costs. See Section 7.6 on page 38.

`trans_priors` <string>: The name of the structure-specific transition prior library to use when structural information for transition probability estimation is to be used for HMM estimation. See Section 7.1.2 on page 28.

`transweight` <float> [1.0]: A multiplier that affects the influence of the pseudocounts generated by the structure-specific transition priors. See Section 7.1.2 on page 28.

`template` <string>: For use with the structure-specific transition prior library. A three- column file (amino acid sequence, secondary structure, accessibility) that is used during HMM estimation to assign a structural environment to each model node. See Section 7.1.2 on page 28.

`viterbi` <0 or 1> [0]: If this is set, score or train using the Viterbi algorithm rather than EM. See Section 9.2 on page 56.

`weight_final` <float> [1.0]: The final (steady-state) multiplier of sequence weights. The default (1.0) means that, if no sequence weight file is used, each sequence is weighted as being one sequence. If a weight file is used, all values in that file are multiplied by this value. See Section 8.4 on page 44 and Section 8.1 on page 40.

`weight_length` <float> [0]: An annealing schedule for the sequence weight multiplier. If greater than 1.0, the weight multiplier is increased from zero linearly over `weight_length` reestimates. If less than one, increase exponentially. See Section 8.4 on page 44 and Section 8.1 on page 40.

`window_size` <integer> [1000]: Window size for use in Z-score calculation by `hmmscore`. See Section 9.2 on page 56.

`Zmax` <float> [4.0]: Z-score beyond which points are considered outliers during curve fitting in `hmmscore`. When a selection variable includes 4 in its binary representation, `Zmax` is used to determine what sequences are selected. Also, when `select_score/seq`= 4, sequences with a Z-score greater than `Zmax` are selected. See Section 9.2 on page 56.