MUSCLE User Guide

Multiple sequence comparison by log-expectation

by Robert C. Edgar

Version 2.0
November 2003

http://www.drive5.com/muscle
muscle (at) drive5.com

Table of Contents

1 Introduction

MUSCLE is a program for creating multiple alignments of amino acid sequences. A range of options is provided that give you the choice of optimizing accuracy, speed, or some compromise between the two. Default parameters are those that give the best average accuracy in our tests. Using versions current at the time of writing, my tests show that MUSCLE can achieve both better average accuracy and better speed than CLUSTALW or T-Coffee, depending on the chosen options.

2 Quick Start

The MUSCLE algorithm is delivered as a command-line program called *muscle*. If you are running under Linux or Unix that means you will need to be working at a shell prompt. If you are running under Windows, you should be in a command window (nostalgically known to us older people as a DOS prompt). If you don't know how to use command-line programs, you will need to get help—from now on, I will assume you know how to work with command-line programs.

2.1 Installation

Copy the *muscle* binary file to a directory that is accessible from your computer. That's it—there are no configuration files, libraries, environment variables or other settings to worry about. If you are using Windows, then the binary file is named *muscle.exe*. From now on *muscle* should be understood to mean "*muscle* if you are using Linux or Unix, *muscle.exe* if you are using Windows".

2.2 Making an alignment

Make a FASTA file containing some protein sequences. (If you are not familiar with FASTA format, it is described in detail later in this Guide.) For now, just to make things fast, limit the number of sequence in the file to no more than 50 and the sequence length to be no more than 500. Call the input file *seqs.fa*. (An example file named *seqs.fa* is distributed with the standard MUSCLE package). Make sure the directory containing the *muscle* binary is in your path. (If it isn't, you can run it by typing the full path name, and the following example command lines must be changed accordingly). Now type:

```
muscle -in seqs.fa -out seqs.afa
```

You should see some progress messages. If *muscle* completes successfully, it will create a file *seqs.afa* containing the alignment. By default, output is created in "aligned FASTA" format (hence the *.afa* extension). This is just like regular FASTA except that gaps are added in order to align the sequences. This is a nice format for computers but not very readable for people, so to look at the alignment you will want an alignment viewer such as Belvu, or a script that converts FASTA to a more readable format. You can also use the *–msf* command-line option to request output in MSF format, which is easier to understand for people. If *muscle* gives an error message and you don't know how to fix it, please read the Troubleshooting section.

The default settings are designed to give the best accuracy, so this may be all you need to know.

2.3 Large alignments

If you have a large number of sequences (thousands), or they are very long, then the default settings of may be too slow for practical use. A good compromise between speed and accuracy is to run just the first two iterations of the algorithm. On average, this gives significantly better accuracy than CLUSTALW but much better speed. This is done by the option *–maxiters 2*, as in the following example.

```
muscle -in seqs.fa -out seqs.afa -maxiters 2
```

There is a longer discussion later about what to do if you have a very large alignment—*muscle* is designed with this kind of challenge in mind, so there are several features that may help you with your particular situation.

2.4 Fastest speed

If you want the fastest possible speed, then the following example shows the applicable options.

```
muscle -in seqs.fa -out seqs.afa -maxiters 1 -diags -sp
```

At the time of writing, this is the faster than any other multiple sequence alignment program that I have tested. The alignments are not bad, especially when the sequences are closely related. However, as you might expect, this blazing speed comes at the cost of the lowest average accuracy of the options that *muscle* provides.

2.5 Accuracy: caveat emptor

Why do I keep using the clumsy phrase "average accuracy" instead of just saying "accuracy"? That's because the quality of alignments produced by MUSCLE varies, as do those produced other programs such as CLUSTALW and T-Coffee. The state of the art leaves plenty of room for improvement. Sometimes the fastest speed options to *muscle* give alignments that are better than T-Coffee, though the reverse will more often be the case. With challenging sets of sequences, it is a good idea to make several different alignments using different *muscle* options and to try other programs too. Regions where different alignments agree are more believable than regions where they disagree.

2.6 Pipelining

Input can be taken from standard input, and output can be written to standard output. This is the default, so our first example would also work like this:

```
muscle < seqs.fa > seqs.afa
```

2.7 Refining an existing alignment

You can ask *muscle* to try to improve an existing alignment by using the*–refine* option. The input file must then be a FASTA file containing an alignment. All sequences must be of equal length, gaps can be specified using dots "." or dashes "–". For example:

```
muscle -in seqs.afa -out refined.afa -refine
```

3 File Formats

MUSCLE uses FASTA format for both input and output. It does support a variant of MSF as a more readable output format; this is selected by using the *–msf* option.

3.1 Input files

Input files must be in FASTA format. These are plain text files (word processing files such as Word documents are not understood!). Unix, Windows and DOS text files are supported (end-of-line may be NL or CR NL). There is a maximum length of 16,000 characters per line in the current version (this limit is subject to change, and hopefully elimination, in future versions). There is no explicit limit on the length of a sequence, however if you are running a 32-bit version of *muscle* then the maximum will be very roughly 10,000 letters due to maximum addressable size of tables required in memory. Each sequence starts with an annotation line, which is recognized by having a greater-than symbol ">" as its first character. There is no limit on the length of an annotation line (other than the input line length limit), and there is no requirement that the annotation be unique. The sequence itself follows on one or more subsequent lines, and is terminated either by the next annotation line or by the end of the file. The standard single-letter amino acid alphabet is used. Upper and lower case is allowed, the case is not significant. The special characters X, B, Z and U are understood. X means "unknown amino acid", B is D or N, Z is E or Q. Nucleotide sequences (DNA and RNA) are not supported. If you give *muscle* a file containing letters AGCTU only, it will assume that they are amino acids, not nucleotides. U is understood to be the 21st amino acid Selenocysteine (three-letter abbreviation Sel; not to be confused with the RNA base Uracil which is represented by U in some alphabets). White space (spaces, tabs and the end-of-line characters CR and NL) is allowed inside sequence data. Dots "." and dashes "–" in sequences are allowed and are discarded unless the input is expected to be aligned (*–refine* option).

3.2 Output files

By default, output is also written in FASTA format. All letters are upper-case and gaps are represented by dashes "–". You can also request output in MSF format, which is more readable than FASTA, by using the *–msf* command-line option. It would be nice if more output formats were supported—please let me know what formats you would find useful.

4 Using MUSCLE

In this section we give more details of the MUSCLE algorithm and the more important options offered by the *muscle* implementation.

4.1 How the algorithm works

We won't give a complete description of the MUSCLE algorithm here—for that, you will have to read the paper (which at the time of writing will be quite challenging, given that I haven't written it yet). But hopefully a summary will help explain what some of the command-line options do and how they might be useful in your work.

The first step is to calculate a tree. In CLUSTALW, this is done as follows. Each pair of input sequences is aligned, and used to compute the pair-wise identity of the pair. Identities are converted to a measure of distance. Finally, the distance matrix is converted to a tree using a clustering method (CLUSTALW uses neighbor-joining). If you have 1,000 sequences, there are (1,000 × 999)/2 = 499,500 pairs, so aligning every pair can take a while. MUSCLE uses a much faster, but somewhat more approximate, method to compute distances: it counts the number of short sub-sequences (known as *k*-mers, *k*-tuples or words) that two sequences have in common, without constructing an alignment. This is typically around 3,000 times faster that CLUSTALW's method, but the trees will generally be less accurate. We call this step "*k*-mer clustering".

The second step is to use the tree to construct what is known as a progressive alignment. At each node of the binary tree, a pair-wise alignment is constructed, progressing from the leaves towards the root. The first alignment will be made from two sequences. Later alignments will be one of the three following types: sequence-sequence, profile-sequence or profile-profile, where "profile" means the multiple alignment of the sequences under a given internal node of the tree. This is very similar to what CLUSTALW does once it has built a tree.

Now we have a multiple alignment, which has been built very quickly compared with conventional methods, mainly because of the distance calculation using *k*-mers rather than alignments. The quality of this alignment is typically pretty good—it will often tie or beat a T-Coffee alignment on our tests. However, on average, we find that it can be improved by proceeding through the following steps.

From the multiple alignment, we can now compute the pair-wise identities of each pair of sequences. This gives us a new distance matrix, from which we estimate a new tree. We compare the old and new trees, and re-align subgroups where needed to produce a progressive multiple alignment from the new tree. If the two trees are identical, there is nothing to do; if there are no subtrees that agree (very unlikely), then the whole progressive alignment procedure must be repeated from scratch. Typically we find that the tree is pretty stable near the leaves, but some re-alignments are needed closer the root. This procedure (compute pair-wise identities, estimate new tree, compare trees, re-align) is iterated until the tree stabilizes or until a specified maximum number of iterations has been done. We call this process "tree refinement", although it also tends to improve the alignment.

We now keep the tree fixed and move to a new procedure which is designed to improve the multiple alignment. The set of sequences is divided into two subsets (i.e., we make a bipartition on the set of sequences). A profile is constructed for each of the two subsets based on the current multiple alignment.

These two profiles are then re-aligned to each other using the same pair-wise alignment algorithm as used in the progressive stage. If this improves an "objective score" that measures the quality of the alignment, then the new multiple alignment is kept, otherwise it is discarded. By default, the objective score is the classic sum-of-pairs score that takes the (sequence weighted) average of the pair-wise alignment score of every pair of sequences in the alignment. Bipartitions are chosen by deleting an edge in the guide tree, each of the two resulting subtrees defines a subset of sequences. This procedure is called "tree dependent refinement". One iteration of tree dependent refinement tries bipartitions produced by deleting every edge of the tree in depth order moving from the leaves towards the center of the tree. Iterations continue until convergence or up to a specified maximum.

For convenience, the major steps in MUSCLE are described as "iterations", though the first three iterations all do quite different things and may take very different lengths of time to complete. The tree-dependent refinement iterations 3, 4 ... are true iterations and will take similar lengths of time.

Iteration	**Actions**
1	Distance matrix by *k*-mer clustering, estimate tree, progressive alignment according to this tree.
2	Distance matrix by pair-wise identities from current multiple alignment, estimate tree, progressive alignment according to new tree, repeat until convergence or specified maximum number of times.
3, 4 ...	Tree-dependent refinement. One iteration visits every edge in the tree one time.

4.2 Command-line options

There are two types of command-line options: value options and flag options. Value options are followed by the value of the given parameter, for example *–in <filename>*; flag options just stand for themselves, such as *–msf*. All options are a dash (not two dashes!) followed by a long name; there are no single-letter equivalents. Value options must be separated from their values by white space in the command line. Thus, *muscle* does not follow Unix, Linux or Posix standards, for which we apologize. The order in which options are given is irrelevant unless two options contradict, in which case the right-most option silently wins.

4.3 The maxiters option

You can control the number of iterations that MUSCLE does by specifying the *–maxiters* option. If you specify 1, 2 or 3, then this is exactly the number of iterations that will be performed. If the value is greater than 3, then *muscle* will continue up to the maximum you specify or until convergence is reached, which ever happens sooner. The default is 16. If you have a large number of sequences, tree-dependent refinement may be very slow (see later section Large Alignments).

4.4 The maxtrees option

This option controls the maximum number of new trees to create in iteration 2. Our experience suggests that a point of diminishing returns is typically reached after the first tree, so the default value is 1. If a larger value is given, the process will repeat until convergence or until this number of trees has been created, which ever comes first.

4.5 The maxhours option

If you have a large alignment, *muscle* may take a long time to complete. It is sometimes convenient to say "I want the best alignment I can get in 24 hours" rather than specifying a set of options that will take an unknown length of time. This is done by using *–maxhours*, which specifies a floating-point number of hours. If this time is exceeded, *muscle* will write out current alignment and stop. For example,

```
   muscle -in huge.fa -out huge.afa -maxiters 9999 -maxhours 24.0
```

Note that the actual time may exceed the specified limit by a few minutes while *muscle* finishes up on a step. It is also possible for no alignment to be produced if the time limit is too small.

4.6 The profile scoring function

Three different profile scoring functions are supported, the log-expectation score (*–le* option) and a sum of pairs score using either the PAM200 matrix (*–sp*) or the VTML240 matrix (*–sv*). The log-expectation score is the default as it gives consistently better results on our tests, but is typically somewhere between two or three times slower than the sum-of-pairs score.

4.7 Diagonal optimization

Creating a pair-wise alignment by dynamic programming requires computing an $L_1 \times L_2$ matrix, where L_1 and L_2 are the sequence lengths. A trick used in algorithms such as BLAST is to reduce the size of this matrix by using fast methods to find "diagonals", i.e. short regions of high similarity between the two sequences. This speeds up the algorithm at the expense of some reduction in accuracy. MUSCLE uses a technique called *k*-mer extension to find diagonals. It is disabled by default because of the slight reduction in average accuracy and can be turned on by specifying the *–diags* option.

4.8 Anchor optimization

Tree-dependent refinement (iterations 3, 4 ...) can be speeded up by dividing the alignment vertically into blocks. Block boundaries are found by identifying high-scoring columns (e.g., a perfectly conserved column of Cs or Ws would be a candidate). Each vertical block is then refined independently before reassembling the complete alignment, which is faster because of the L^2 factor in dynamic programming (e.g., suppose the alignment is split into two vertical blocks, then $2 \times 0.5^2 = 0.5$, so the dynamic programming time is roughly halved). The *–anchors* option is used to enable this feature. As with diagonal optimization, my tests show that anchors result in a very small reduction in average accuracy, so are disabled by default. This option has no effect if *–maxiters 1* or *–maxiters 2* is specified.

4.9 Log file

You can specify a log file by using *–log <filename>* or *–loga <filename>*. Using *–log* causes any existing file to be deleted, *–loga* appends to any existing file. A message will be written to the log file when *muscle* starts and stops. Error and warning messages will also be written to the log. If *–verbose* is specified, then more information will be written, including the command line used to invoke *muscle*, the resulting internal parameter settings, and also progress messages. The content and format of verbose log file output is subject to change in future versions.

The use of a log file may seem contrary to Unix conventions for using standard output and standard error. I like these conventions, but never found a fully satisfactory way to use them. I like progress messages (see below), but they mess up a file if you re-direct standard error and there are errors or warning messages too. I could try to detect whether a standard file handle is a *tty* device or a disk file and change behavior accordingly, but I regard this as too complicated and too hard for the user to understand. On Windows it can be hard to re-direct standard file handles, especially when working in a GUI debugger. Maybe one day I will figure out a better solution (suggestions welcomed).

I highly recommend using *–verbose* and *–log[a]*, especially when running *muscle* in a batch mode. This enables you to verify whether a particular alignment was completed and to review any errors and warning that occurred.

4.10 Progress messages

By default, *muscle* writes progress messages to standard error periodically so that you know it's doing something and get some feedback about the time and memory requirements for the alignment. Here is a typical progress message.

```
00:00:23     25 Mb  Iter   2  87.20%  Build guide tree
```

The fields are as follows.

`00:00:23`	Elapsed time since *muscle* started.
`25 Mb`	Peak memory use in megabytes (i.e., not the current usage, but the maximum amount of memory used since *muscle* started).
`Iter 2`	Iteration currently in progress.
`87.20%`	How much of the current step has been completed (percentage).
`Build...`	A brief description of the current step.

The *–quiet* command-line option disables writing progress messages to standard error. If the *–verbose* command-line option is specified, a progress message will be written to the log file when each iteration completes. So *–quiet* and *–verbose* are not contradictory.

4.11 Recommended usage

If I had to recommend a single set of options for all input, it would be these:

```
muscle -in infile -out outfile -maxiters 2 -verbose -loga logfile
```

The default parameters give the best average accuracy, but for very large sets of sequences the tree dependent refinement iterations can be too slow to be practical. The first two iterations should complete in reasonable time and memory for just about any set of sequences that *muscle* is capable of aligning at all. The average accuracy obtained by these parameters is better than CLUSTALW, but alignments are produced in much less time. For example, on my 2.5 GHz Pentium 4 PC, these options align 1,000 sequences of average length 280 in 140 seconds.

4.12 Large alignments

Suppose you have a large number of sequences, and/or they are very long, and you want to try to do more than two iterations. There are no hard and fast rules about what to do, and what works best will depend on your particular sequences, but here are some suggestions.

Use the following options:

```
muscle -in infile -out outfile -maxiters 9999 -maxhours 24.0
   -verbose -log logfile
```

Here, I've assumed 24 hours to be the length of time that you are prepared to allocate for this experiment, you should of course use your own preferred value. See how many iterations *muscle* completes in the given time and how you like the quality of the alignment. If *muscle* failed to finish the third iteration, here are some things that might speed up the process. Even if it did manage to do three or more, it might be better to do more iterations with these options enabled versus fewer refinement iterations.

If there are groups of closely related sequences in your set, then using *–diags* may give you a significant speed improvement without degrading accuracy.

It may be worth trying *–anchors*, especially if sequences are long. This can degrade accuracy, but only rarely, especially when sequences are closely related.

For large numbers of sequences, it can give a significant speed improvement to use *–objscore dp*. This reduces the cost of computing the objective score from O(N^2) to O(N), where N is the number of sequences. This option affects only the tree dependent refinement iterations (3, 4, ...).

To test the possible speed improvements due to *–diags*, *–anchors* or *–objscore dp*, you could save the output from two iterations, then use *–refine <savedalignment> –maxiters 1* to request one tree dependent refinement iteration. You could compare the results of different combinations of parameters for speed and alignment quality, or just try all three together. If you're happy with the time/accuracy trade-off, and want more iterations, then you can run *–refine* on the output from the first *–refine*; you don't need to start from scratch.

4.13 Global alignment

Like many other multiple alignment programs, including CLUSTALW, MUSCLE constructs global alignments. This means that there is a built-in assumption that sequences are related over all or most of their lengths. There is some confusion on this point, and some nonsense has been written about it (even in the peer-reviewed literature), so I will make some brief comments on it here.

Protein structures are generally constructed from relatively independent units called domains which range in length from a few tens to a few hundreds of amino acids. Similar domains are often found in different proteins, in which case the sequences are often believed to be related through evolution by descent from a common ancestor. I will consider three domains A, B and C, and will represent the sequences for these three domains as AAAA, BBBB and CCCC. These sequences should be understood as being similar but not necessarily identical in each protein (with any differences assumed to be due to substitutions, deletions and insertions accumulated over evolutionary time). A real domain will of course be longer than four letters. The easiest case is a global alignment of two single-domain proteins, like this.

```
AAAAA
AAAAA
```

If two multi-domain proteins have the same domains in the same order, this case is also straightforward for a global alignment algorithm.

```
AAAAABBBBBCCC
AAAAABBBBBCCC
```

Aligning a single domain protein to a multi-domain protein typically works reasonably well:

```
----BBBBB----
AAAABBBBBCCCC
```

It has been suggested that global alignment programs can't cope with this because the gap penalties become very large due to the long missing regions. But when there is a big difference in lengths, the gaps have to go somewhere, and the correct alignment will often be the one that gets the highest score due to the high similarity of the domain that is present in the shorter sequence.

Other cases may not be so easy. Suppose domain A is missing in one of the proteins. The correct alignment is then as follows.

```
-----BBBBBCCC
AAAAABBBBBCCC
```

Now suppose B is missing.

```
AAAAA-----CCC
AAAAABBBBBCCC
```

Proving that the sequences for the common domains are closely related, these cases are not too hard for a global alignment algorithm. However, there is an important design issue to consider, which is how terminal gaps (i.e., gaps extending from the beginning or end of a sequence) are treated by the program. Some programs allow terminal gaps for free, i.e., no penalties are applied (this is what CLUSTALW does). Then there is a strong incentive to put gaps at the end rather than in the middle. This can lead to bad alignments like this.

```
---AAAACCC---
```

```
AAAAABBBBBCCC
```

On the plus side, the terminal-gaps-are-free design is ideal for the case where domain A is deleted in one sequence and domain C in the other:

```
-----BBBBBCCC
AAAAABBBBB---
```

If there are penalties of any kind for terminal gaps, this encourages more compact alignments—which is generally a good thing, but in this situation may go wrong something like this:

```
BBBBBCCC
AAAABBBB
```

If typical affine penalties (gap open + gap length) are applied to terminal gaps, this can lead to a different kind of mistake by encouraging the fewest possible gaps, regardless of their location. So it may tend to produce alignments with a gap at one end but not the other:

```
-------BBBBB
AAAAABBBBCCC
```

One way to prepare input that is suitable for a global alignment algorithm is to collect data using a local method (e.g., BLAST or PSI-BLAST), collect locally aligned regions and re-align them using a global method such as MUSCLE.

By default, *muscle* penalizes terminal gaps with half the usual penalty, which we believe is a reasonable design compromise and shows a small improvement over both free and full gaps in our tests. However, as the above examples show, in particular situations it may be more appropriate to apply different penalties to terminal gaps. The default can be changed by specifying the *–termgapsfull* option, which applies full penalties to terminal gaps. Because of certain technical complications in the MUSCLE algorithm, there is no option to make terminal gaps free. An option for free terminal gaps may be added in future versions—let me know if you would find that feature useful.

The moral of this story is that it helps to understand (a) your sequences, (b) something about gap penalties and the way global and local alignment algorithms work, and (c) some of the more obscure options in the alignment programs you use.

4.14 Running out of memory

The *muscle* code tries to deal gracefully with low-memory conditions by using the following technique. A block of "emergency reserve" memory is allocated when *muscle* starts. If a later request to allocate memory fails, this reserve block is made available, and *muscle* attempts to save the current alignment. With luck, the reserved memory will be enough to allow *muscle* to save the alignment and exit gracefully with an informative error message.

4.15 Troubleshooting

Here is some general advice on what to do if *muscle* fails and you don't understand what happened. The code is designed to fail gracefully with an informative error message when something goes wrong, but there will no doubt be situations I haven't anticipated (not to mention bugs).

Check the MUSCLE web site for updates, bug reports and other relevant information.

http://www.drive5.com/muscle

Check the input file to make sure it is in valid FASTA format. Try giving it to another sequence analysis program that can accept large FASTA files (e.g., the NCBI *formatdb* utility) to see if you get an informative error message. Try dividing the file into two halves and using each half individually as input. If one half fails and the other does not, repeat until the problem is localized as far as possible.

Use *–log* or *–loga* and *–verbose* and check the log file to see if there are any messages that give you a hint about the problem. Look at the peak memory requirements (reported in progress messages) to see if you may be exceeding the physical or virtual memory capacity of your computer.

If *muscle* crashes without giving an error message, or hangs, then you may need to refer to the source code or use a debugger. A "debug" version, *muscled*, may be provided. This is built from the same source code but with the DEBUG macro defined and without compiler optimizations. This version runs much more slowly (perhaps by a factor of three or more), but does a lot more internal checking and may be able to catch something that is going wrong in the code. The *–core* option specifies that *muscle* should not catch exceptions. When *–core* is specified, an exception may result in a debugger trap or a core dump, depending on the execution environment. The *–nocore* option has the opposite effect. In *muscle*, *–nocore* is the default, *–core* is the default in *muscled*.

4.16 Technical support

I am happy to provide support. But I am busy, and am offering this program at no charge, so I ask you to make a reasonable effort to figure things out for yourself before contacting me.

5 Command Line Reference

Value option	Legal values	Default	Description
`anchorspacing`	Integer	`32`	Minimum spacing between anchor columns.

`cluster1`
`cluster2`
	`upgma`
	`upgmb`
	`neighborjoining`
		`upgmb`	Clustering method. cluster1 is used in iteration 1 and 2, cluster2 in later iterations.

`distance1`	`kmer6_6`
	`kmer20_3`
	`kmer20_4`
		`Kmer6_6`	Distance measure for iteration 1.

`distance2`	`kmer6_6`
	`kmer20_3`
	`kmer20_4`
	`pctid_kimura`
	`pctid_log`
		`pctid_kimura`	Distance measure for iterations 2, 3 ...

`gapopen`	Floating point	[1]	The gap open score. Must be negative.
`in`	Any file name	standard input	Where to find the input sequences.
`log`	File name	None.	Log file name (delete existing file).
`loga`	File name	None.	Log file name (append to existing file).
`maxiters`	Integer 1, 2 ...	`16`	Maximum number of iterations.
`maxtrees`	Integer	`1`	Maximum number of new trees to build in iteration 2.
`minbestcolscore`	Floating point	[1]	Minimum score a column must have to be an anchor.

`objscore`	`sp`
	`ps`
	`dp`
		`sp`	Objective score used by tree dependent refinement.

Value option	Legal values	Default	Description

	xp		sp=sum-of-pairs score.
			dp=dynamic programming score.
			ps=average profile-sequence score.
			xp=cross profile score.

out	File name	standard output	Where to write the alignment.
root1 root2	pseudo midlongestspan minavgleafdist	psuedo	Method used to root tree; root1 is used in iteration 1 and 2, root2 in later iterations.
smoothscoreceil	Floating point	[1]	Maximum value of column score for smoothing purposes.
smoothwindow	Integer	7	Window used for anchor column smoothing.
SUEFF	Floating point value between 0 and 1.	0.1	Constant used in UPGMB clustering.
weight1 weight2	none henikoff henikoffpb gsc clustalw threeway	henikoffpb	Sequence weighting scheme. weight1 is used in iterations 1 and 2. weight2 is used for tree-dependent refinement. none=all sequences have equal weight. henikoff=Henikoff & Henikoff weighting scheme. henikoffpb=Modified Henikoff scheme as used in PSI-BLAST. clustalw=CLUSTALW method. threeway=Gotoh three-way method.

Flag option	Set by default?	Description
anchors	no	Use anchor optimization in tree dependent refinement iterations.
core	yes in muscle, no in muscled.	Do not catch exceptions.
le	yes	Use log-expectation profile score (VTML240). Alternatives are to use *–sp* or *–sv*.
msf	no	Write output in MSF format (default is to use FASTA).
nocore	no in muscle, yes in muscled.	Catch exceptions and give an error message if possible.
quiet	no	Do not display progress messages.
refine	no	Input file is already align, skip first two iterations and begin tree dependent refinement.
sp	no	Use sum-of-pairs profile score (PAM200). Default is *–le.*
sv	no	Use sum-of-pairs profile score (VTML240). Default is *–le.*

Flag option	Set by default?	Description
`termgapsfull`	no	Terminal gaps penalized with full penalty.
`termgapshalf`	yes	Terminal gaps penalized with half penalty.
`termgapshalflonger`	no	Terminal gaps penalized with half penalty if gap relative to longer sequence, otherwise with full penalty.
`verbose`	no	Write parameter settings and progress messages to log file.

Notes
[1] Default depends on the profile scoring function. To determine the default, use *–verbose –log* and check the log file.