MUSCLE User Guide

Multiple sequence comparison by log-expectation
by Robert C. Edgar

Version 3.52
November 2004

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

MUSCLE is updated regularly. Send me an e-mail if you would like to be notified of new releases.

Citation:

Edgar, Robert C. (2004), MUSCLE: multiple sequence alignment with high accuracy and high throughput, *Nucleic Acids Research* **32**(5), 1792-97.

For a complete description of the algorithm, see also:

Edgar, Robert C (2004), MUSCLE: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, **5**(1):113.

Table of Contents

1 Introduction 3
2 Quick Start 3
2.1 Installation 3
2.2 Making an alignment 3
2.3 Large alignments 3
2.4 Faster speed 4
2.5 Huge alignments 4
2.6 Accuracy: caveat emptor 4
2.7 Pipelining 4
2.8 Refining an existing alignment 4
2.9 Using a pre-computed guide tree 4
2.10 Profile-profile alignment 5
2.11 Sequence clustering 5
3 File Formats 5
3.1 Input files 5
3.1.1 Amino acid sequences 6
3.1.2 Nucleotide sequences 6
3.1.3 Determining sequence type 6
3.2 Output files 6
3.2.1 Sequence grouping 6
3.3 CLUSTALW format 6
3.4 MSF format 6
3.5 HTML format 7
4 Using MUSCLE 7
4.1 How the algorithm works 7
4.2 Command-line options 8
4.3 The maxiters option 8
4.4 The maxtrees option 8
4.5 The maxhours option 9
4.6 The maxmb option 9
4.7 The profile scoring function 9
4.8 Diagonal optimization 9
4.9 Anchor optimization 9
4.10 Log file 10
4.11 Progress messages 10
4.12 Running out of memory 10
4.13 Troubleshooting 11
4.14 Technical support 11
5 Command Line Reference 11

1 Introduction

MUSCLE is a program for creating multiple alignments of amino acid or nucleotide 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. Many command line options are provided to vary the internals of the algorithm; some of these will primarily be of interest to algorithm developers who wish to better understand which features of the algorithm are important in different circumstances.

2 Quick Start

The MUSCLE algorithm is delivered as a command-line program called *muscle*. If you are running under Linux or Unix you will 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 should get help from a local guru.

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 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 (a few thousand), 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 accuracy equal to T-Coffee and speeds much faster than CLUSTALW. This is done by the option *–maxiters 2*, as in the following example.

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

2.4 Faster speed

The *–diags* option enables an optimization for speed by finding common words (6-mers in a compressed amino acid alphabet) between the two sequences as seeds for diagonals. This is related to optimizations in programs such as BLAST and FASTA: you get faster speed, but sometimes lower average accuracy. For large numbers of closely related sequences, this option works very well.

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

```
muscle -in seqs.fa -out seqs.afa -maxiters 1 -diags -sv -distance1 kbit20_3
```

For nucleotides, use:

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

At the time of writing, *muscle* with these options is 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 Huge alignments

If you have a *very* large number of sequences (several thousand), or they are very long, then the *kbit20_3* option may cause problems because it needs a relatively large amount of memory. Better is to use the default distance measure, which is roughly 2× or 3× slower but needs less memory, like this:

```
muscle -in seqs.fa -out seqs.afa -maxiters 1 -diags1 -sv
```

2.6 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.

2.7 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
```

2.8 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
```

2.9 Using a pre-computed guide tree

The *–usetree* option allows you to provide your own guide tree. For example,

```
muscle -in seqs.fa -out seqs.afa -usetree mytree.phy
```

The tree must by in Newick format, as used by the Phylip package (hence the *.phy* extension). The Newick format is described here:

http://evolution.genetics.washington.edu/phylip/newicktree.html

WARNING. Do not use this option just because you believe that you have an accurate evolutionary tree for your sequences. The best guide tree for multiple alignment is *not* in general the correct evolutionary tree. This can be understood by the following argument. Alignment accuracy decreases with lower sequence identity. It follows that given a set of profiles, the two that can be aligned most accurately will tend to be the pair with the highest identity, i.e. at the shortest evolutionary distance. This is exactly the pair selected by the nearest-neighbor criterion which MUSCLE uses by default. When mutation rates are variable, the *evolutionary* neighbor may not be the *nearest* neighbor. This explains why a nearest-neighbor tree may be superior to the true evolutionary tree for guiding a progressive alignment.

You will get a warning if you use the *–usetree* option. To disable the warning, use *–usetree_nowarn* instead, e.g.:

```
muscle -in seqs.fa -out seqs.afa -usetree_nowarn mytree.phy
```

2.10 Profile-profile alignment

A fundamental step in the MUSCLE algorithm is aligning two multiple sequence alignments. This operation is sometimes called "profile-profile alignment". If you have two existing alignments of related sequences you can use the *–profile* option of MUSCLE to align those two sequences. Typical usage is:

```
muscle -profile -in1 one.afa -in2 two.afa -out both.afa
```

The alignments in *one.afa* and *two.afa*, which must be in aligned FASTA format, are aligned to each other, keeping input columns intact and inserting columns of gaps where needed. Output is stored in *both.afa*.

MUSCLE does not compute a similarity measure or measure of statistical significance (such as an E-value), so this option is not useful for discriminating homologs from unrelated sequences. For this task, I recommend Sadreyev & Grishin's COMPASS program.

2.11 Sequence clustering

The first stage in MUSCLE is a fast clustering algorithm. This may be of use in other applications. Typical usage is:

```
muscle -cluster -in seqs.fa -tree1 tree.phy
```

The sequences will be clustered, and a tree written to *tree.phy*. Options *–weight1*, *–distance1*, *–cluster1* and *–root1* can be applied if desired. Note that by default, UPGMA clustering is used. You can use *–neighborjoining* if you prefer, but note that this is substantially slower than UPGMA for large numbers of sequences, and is also slightly less accurate. See discussion of *–usetree* above.

3 File Formats

MUSCLE uses FASTA format for both input and output. For output only, it also offers CLUSTALW, MSF and HTML formats using the *–clw*, *–msf* and *–html* command-line options.

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 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 (this is new as of version 3.5), 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.

3.1.1 Amino acid sequences

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. U is understood to be the 21st amino acid Selenocysteine. 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 (e.g. for the *–refine* option).

3.1.2 Nucleotide sequences

The usual letters A, G, C, T and U stand for nucleotides. The letters T and U are equivalent as far as MUSCLE is concerned. N is the wildcard meaning "unknown nucleotide". R means A or G, Y means C or T/U. Other wildcards, such as those used by RFAM, are not understood in this version and will be replaced by Ns. If you would like support for other DNA / RNA alphabets, please let me know.

3.1.3 Determining sequence type

By default, MUSCLE looks at the first 100 letters in the input sequence data (excluding gaps). If 95% or more of those letters are valid nucleotides (AGCTUN), then the file is treated as nucleotides, otherwise as amino acids. This method almost always guesses correctly, but you can make sure by specifying the sequence type on the command line. This is done using the *–seqtype* option, which can take the following values:

–seqtype protein	Amino acid
–seqtype nucleo	Nucleotide
–seqtype auto	Automatic detection (default).

3.2 Output files

By default, output is also written in FASTA format. All letters are upper-case and gaps are represented by dashes "–".

3.2.1 Sequence grouping

By default, MUSCLE re-arranges sequences so that similar sequences are adjacent in the output file. (This is done by ordering sequences according to a prefix traversal of the guide tree). This makes the alignment easier to evaluate by eye. If you want to the sequences to be output in the same order as the input file, you can use the *–stable* option.

3.3 CLUSTALW format

You can request CLUSTALW output by using the *–clw* option. This should be compatible with CLUSTALW, with the exception of the program name in the file header. You can ask MUSCLE to impersonate CLUSTALW by writing "CLUSTAL W (1.81)" as the program name by using *–clwstrict*. Note that MUSCLE allows duplicate sequence labels, while CLUSTALW forbids duplicates. If you use the *–stable* option of *muscle*, then the order of the input sequences is preserved and sequences can be unambiguously identified even if the labels differ. If you have problems parsing MUSCLE output with scripts designed for CLUSTALW, please let me know and I'll do my best to provide a fix.

3.4 MSF format

MSF format, as used in the GCG package, is requested by using the *–msf* option. As with CLUSTALW format, this is easier for people to read than FASTA. Gaps are represented by a tilde (~). In MUSCLE 3.52, the MSF format has been tweaked to be more compatible with GCG. The following differences remain.

(a) MUSCLE truncates at the first white space or after 63 characters, which ever comes first. The GCG package apparently truncates after 10 characters. If this is a problem for you, please let me know and I'll add an option to truncate after 10 in a future version.

(b) MUSCLE allows duplicate sequence labels, while GCG forbids duplicates. If you use the *–stable* option of *muscle*, then the order of the input sequences is preserved and sequences can be unambiguously identified even if the labels differ.

Thanks to Eric Martel for help with improving GCG compatibility.

3.5 HTML format

I've added an experimental feature starting in version 3.4. To get a Web page as output, use the *–html* option. The alignment is colored using a color scheme from Eric Sonnhammer's Belvu editor, which is my personal favorite. A drawback of this option is that the Web page typically contains a very large number of HTML tags, which can be slow to display in the Internet Explorer browser. The Netscape browser works much better. If you have any ideas about good ways to make Web pages, please let me know.

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

I won't give a complete description of the MUSCLE algorithm here—for that, you will have to read the papers. (See citations on title page above). 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 $(1{,}000 \times 999)/2 = 499{,}500$ 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 unusual), 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, refinement may be rather slow.

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 maxmb option

If the amount of memory needed by MUSCLE exceeds available physical RAM, then the operating system will probably begin paging (i.e., swapping memory to and from hard disk), causing MUSCLE to run very slowly. This is especially problematic when MUSCLE is used for batch processing, where one or two very large alignments can cause a batch to effectively hang. Starting in version 3.52, MUSCLE attempts to limit the amount of memory used. If the limit is exceeded, MUSCLE quits, saving the best alignment so far produced (if any). MUSCLE attempts to determine the amount of physical RAM by making an appropriate operating system call. Under Linux and Windows, this works well. On other systems, particularly other flavors of Unix, MUSCLE doesn't know how to query the system and assumes that there is 500 Mb of RAM. To override this default, you can specify the maximum number of megabytes to allocate by using the *–maxmb* option, for example to set a limit of 1.5 Gb:

```
muscle -in huge.fa -out huge.afa -maxhours 1.0 -maxmb 1500
```

This feature has been hacked on top of code that wasn't really designed for it. So it doesn't always work perfectly, but is better than nothing. The ideal solution would be to implement linear space dynamic programming code (e.g., the Myers-Miller algorithm) for situations where memory is tight. One day I might do this if there is sufficient interest. If you are interested in contributing the code, e.g. for a class project, please let me know, I'll be glad to provide support.

4.7 The profile scoring function

Three different protein 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 better results on our tests, but is typically somewhere between two or three times slower than the sum-of-pairs score. For nucleotides, *–spn* is currently the only option (which is of course the default for nucleotide data, so you don't need to specify this option).

4.8 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. To enable diagonal optimization in the first iteration, use *–diags1*, to enable diagonal optimization in the second iteration, use *–diags2*. These are provided separately because it would be a reasonable strategy to enable diagonals in the first iteration but not the second (because the main goal of the first iteration is to construct a multiple alignment quickly in order to improve the distance matrix, which is not very sensitive to alignment quality; whereas the goal of the second iteration is to make the best possible progressive alignment).

4.9 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 *–noanchors* option is used to disable this feature. This option has no effect if *–maxiters 1* or *–maxiters 2* is specified. On benchmark tests, enabling anchors has little or no effect on accuracy, but if you want to be very conservative and are striving for the best possible accuracy then *–noanchors* is a reasonable choice.

4.10 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 or warnings that occurred.

4.11 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 (5%)  Iter   2  87.20%  Build guide tree
```

The fields are as follows.

`00:00:23`	Elapsed time since *muscle* started.
`25 Mb (5%)`	Peak memory use in megabytes (i.e., not the current usage, but the maximum amount of memory used since *muscle* started). The number in parentheses is the fraction of physical memory (see *–maxmb* option for more discussion).
`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.

4.12 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. See also the *–maxmb* option.

4.13 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*.

4.14 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.
`center`	Floating point	[1]	Center parameter. Should be negative.

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

`diagbreak`	Integer	1	Maximum distance between two diagonals that allows them to merge into one diagonal.
`diaglength`	Integer	`24`	Minimum length of diagonal.
`diagmargin`	Integer	`5`	Discard this many positions at ends of diagonal.

`distance1`	`kmer6_6`
	`kmer20_3`
	`kmer20_4`
		`Kmer6_6 (amino) or Kmer4_6`	Distance measure for iteration 1.

Value option	Legal values	Default	Description
	kbit20_3 kmer4_6	(nucleo)	
distance2	kmer6_6 kmer20_3 kmer20_4 kbit20_3 pctid_kimura pctid_log	pctid_kimura	Distance measure for iterations 2, 3 ...
gapopen	Floating point	[1]	The gap open score. Must be negative.
hydro	Integer	5	Window size for determining whether a region is hydrophobic.
hydrofactor	Floating point	1.2	Multiplier for gap open/close penalties in hydrophobic regions.
in	Any file name	standard input	Where to find the input sequences.
in1	Any file name	None	Where to find an input alignment.
in2	Any file name	None	Where to find an input alignment.
log	File name	None.	Log file name (delete existing file).
loga	File name	None.	Log file name (append to existing file).
maxhours	Floating point	None.	Maximum time to run in hours. The actual time may exceed the requested limit by a few minutes. Decimals are allowed, so 1.5 means one hour and 30 minutes.
maxiters	Integer 1, 2 ...	16	Maximum number of iterations.
maxmb	Integer	80% of Physical RAM, or 500 Mb if not known.	Maximum memory to allocate in Mb.
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.
minsmoothscore	Floating point	[1]	Minimum smoothed score a column must have to be an anchor.
objscore	sp ps dp xp spf spm	spm	Objective score used by tree dependent refinement. sp=sum-of-pairs score. spf=sum-of-pairs score (dimer approximation) spm=sp for < 100 seqs, otherwise spf dp=dynamic programming score. ps=average profile-sequence score. xp=cross profile score.

Value option	Legal values	Default	Description
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.
seqtype	protein nucleo auto	auto	Sequence type.
smoothscoreceil	Floating point	[1]	Maximum value of column score for smoothing purposes.
smoothwindow	Integer	7	Window used for anchor column smoothing.
spscore	File name		Compute SP objective score of multiple alignment.
SUEFF	Floating point value between 0 and 1.	0.1	Constant used in UPGMB clustering. Determines the relative fraction of average linkage (SUEFF) vs. nearest-neighbor linkage (1 – SUEFF).
tree1 tree2	File name	None	Save tree produced in first or second iteration to given file in Newick (Phylip-compatible) format.
usetree	File name	None	Use given tree as guide tree. Must by in Newick (Phyip-compatible) format.
weight1 weight2	none henikoff henikoffpb gsc clustalw threeway	clustalw	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`	yes	Use anchor optimization in tree dependent refinement iterations.
`brenner`	no	Use Steven Brenner's method for computing the root alignment.
`cluster`	no	Perform fast clustering of input sequences. Use the *–tree1* option to save the tree.
`dimer`	no	Use dimer approximation for the SP score (faster, slightly less accurate).
`clw`	no	Write output in CLUSTALW format (default is FASTA).
`clwstrict`	no	Write output in CLUSTALW format with the "CLUSTAL W (1.81)" header rather than the MUSCLE version. This is useful when a post-processing step is picky about the file header.
`core`	yes in muscle, no in muscled.	Do not catch exceptions.
`diags`	no	Use diagonal optimizations. Faster, especially for closely related sequences, but may be less accurate.
`diags1`	no	Use diagonal optimizations in first iteration.
`diags2`	no	Use diagonal optimizations in second iteration.
`fasta`	yes	Write output in FASTA format. Alternatives include *–clw*, *–clwstrict*, *–msf* and *–html*.
`group`	yes	Group similar sequences together in the output. This is the default. See also *–stable*.
`html`	no	Write output in HTML format (default is FASTA).
`le`	maybe	Use log-expectation profile score (VTML240). Alternatives are to use *–sp* or *–sv*. This is the default for amino acid sequences.
`msf`	no	Write output in MSF format (default is FASTA). Designed to be compatible with the GCG package.
`noanchors`	no	Disable anchor optimization. Default is *–anchors*.
`nocore`	no in muscle, yes in muscled.	Catch exceptions and give an error message if possible.
`profile`	no	Compute profile-profile alignment. Input alignments must be given using *–in1* and *–in2* options.
`quiet`	no	Do not display progress messages.
`refine`	no	Input file is already aligned, skip first two iterations and begin

Flag option	Set by default?	Description
		tree dependent refinement.
`sp`	no	Use sum-of-pairs protein profile score (PAM200). Default is *–le*.
`spscore`	no	Compute alignment score of profile-profile alignment. Input alignments must be given using *–in1* and *–in2* options. These must be pre-aligned with gapped columns as needed, i.e. must be of the same length (have same number of columns).
`spn`	maybe	Use sum-of-pairs nucleotide profile score. This is the only option for nucleotides, and is therefore the default. The substitution scores and gap penalty scores are "borrowed" from BLASTZ.
`stable`	no	Preserve input order of sequences in output file. Default is to group sequences by similarity (*–group*).
`sv`	no	Use sum-of-pairs profile score (VTML240). Default is *–le.*
`termgaps4`	yes	Use 4-way test for treatment of terminal gaps. (Cannot be disabled in this version).
`termgapsfull`	no	Terminal gaps penalized with full penalty. [1] Not fully supported in this version.
`termgapshalf`	yes	Terminal gaps penalized with half penalty. [1] Not fully supported in this version.
`termgapshalflonger`	no	Terminal gaps penalized with half penalty if gap relative to longer sequence, otherwise with full penalty. [1] Not fully supported in this version.
`verbose`	no	Write parameter settings and progress messages to log file.
`version`	no	Write version string to stdout and exit.

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