PROGRAM NAME: # maskedfastacoords.pl

AUTHOR: **Josep F. Abril** .... jabril@imim.es

LAST UPDATE: **September 25, 2001**

DESCRIPTION: Retrieving positions for masked regions of masked sequences in fasta format. Program can also output those coords in GFF format, where you can define several fields from command-line.

Genome Informatics Research Lab
Grup de Recerca en Infomàtica Biomèdica
Institut Municipal d'Investigació Mèdica
Universitat Pompeu Fabra

Contents

List of Tables

List of Figures

`< Id: maskedfastacoords.nw,v 1.2 2001/09/19 19:45:27 jabril Exp >`

1 Introduction

1.1 Program description

Retrieving positions for masked regions of masked sequences in fasta format. Program can also output those coords in GFF format, where you can define several fields from command-line.

1.2 Input

The following example illustrates both, "classical" (using 'N') and soft (using lowercase letters), masking of fasta sequences as they can be obtained from a program such REPEATMASKER[1]. Sequence is splitted into 50 chararacter-length lines.

1 ⟨*masked fasta input file* 1⟩≡
```
>Masked_fasta_sequence
TATCCATGCCCTCACTAATGTCTCAGTTTTTAGAAATTTTACAGTGCTTA
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
TAAACTTTCTTCTGATGTTGTCTAACCtgtatgttctttgaggacaagga
gagtacacagtaggaactcaatgaatacttgctgatgaNNNNNNNNNNNN
CTCTTAACACTcctcgactcagaatgttcccgtgttctcctctagTGTGA
CCTCTTTTGTGTCTTGAAaggaannnnnatcgctCATCAATCACCAAATA
TCAGTCCCTGTTGGGTGTAGGTCTG
```

The sequence '`Masked_fasta_sequence`' is 325 nucleotides long. Masking is distributed in the following coords:

51	100	N-masking
128	188	soft-masking
189	200	N-masking
212	245	soft-masking
269	284	soft-masking
274	278	n-masking (soft)

1.3 Output

The program can produce three different outputs returning the coords of the sequence masked segments: only "classical" masking ('N' or 'n'), only soft-masking ('[a-z]'), both. From the previous example we must obtain the following results:

Basic-masking

51	100	N-masking
189	200	N-masking
274	278	n-masking

Soft-masking

128	188	soft-masking
212	245	soft-masking
269	284	soft-masking

Merged-masking

51	100	N-masking
128	188	soft-masking
189	200	N-masking
212	245	soft-masking
269	273	soft-masking
274	278	n-masking (soft)
279	284	soft-masking

1.4 To Do

- This is a first draft of the maskedfastacoords.pl. . . . . [*Section* 2.1, *page* 2]
- We can define mask checking as three different subroutines that will be called by reference, so that a variable name (called for instance `$test`) will be set depending on command-line options.[*Section* 3.2, *page* 7]

[1]http://www.genome.washington.edu/UWGC/analysistools/repeatmask.htm

2 Implementation

2.1 Program outline

TO DO

- This is a first draft of the maskedfastacoords.pl.

2a ⟨*maskedfastacoords* 2a⟩≡
```
  ⟨PERL shebang 9a⟩
  #
  # MODULES
  #
  ⟨Use Modules 2f⟩
  #
  # VARIABLES
  #
  ⟨Global Vars 2g⟩
  #
  # MAIN LOOP
  #
  ⟨Main Loop 2h⟩
  #
  # FUNCTIONS
  #
  ⟨Functions 3b⟩
```

2b ⟨*Program Info* 2b⟩≡
```
  my $PROGRAM = 'getfastamasked.pl';
  my $VERSION = '0.1_alpha';
```

2c ⟨*Prog USAGE* 2c⟩≡
```
  #$PROGRAM [options] < fasta_file > coords_file
```

2d ⟨*Prog DESC* 2d⟩≡
```
  #Retrieving masked regions coords from fasta files.
```

2e ⟨*Program Description* 2e⟩≡
```
  #
  #   ⟨Prog USAGE 2c⟩
  #
  #   ⟨Prog DESC 2d⟩
  #
```

2f ⟨*Use Modules* 2f⟩≡
```
  ⟨Use Modules - Benchmark 9c⟩
  ⟨Use Modules - Getopt 3e⟩
  ⟨Use Modules - Bio::Seq 3c⟩
```

2g ⟨*Global Vars* 2g⟩≡
```
  ⟨Boolean 9b⟩
  ⟨Stderr subs vars 11b⟩
  my ($id,$seq) = ("","");
```

2h ⟨*Main Loop* 2h⟩≡
```
&parse_cmdline(); # PROG-START

&main();

&report('PROG-FINISH',&timing($T));

exit(0);
```

3a ⟨*messages: program running* 3a⟩≡
```
'PROG-START'  => "$line$s\n$s Running $PROGRAM\n$s".
                 "$s HOST: $host".
                 "$s USER: $USER".
                 "$s DATE: $DATE\n$s\n$line$s" ,
'PROG-FINISH' => "$s\n$line$s\n$s $PROGRAM FINISHED\n$s".
                 "$s TOTAL TIME: \%s\n$line" ,
```

3b ⟨*Functions* 3b⟩≡
⟨*Parsing command line options* 4a⟩
⟨*Main program function* 6f⟩
⟨*Common PERL subs - Benchmark* 9e⟩
⟨*Common PERL subs - STDERR* 10f⟩

2.2 BIOPERL modules

`Bio::Seq`[2] is the BIOPERL[3] main sequence object while `Bio::SeqIO`[4] is the BIOPERL support for sequence input/output into files. BIOPERL is also available from CPAN[5].

3c ⟨*Use Modules - Bio::Seq* 3c⟩≡
```
use Bio::Seq;
use Bio::SeqIO;
```

3d ⟨*perl requires - BioPerl* 3d⟩≡
```
"Bio::Seq"
"Bio::SeqIO" - BioPerl modules to handle sequence objects. (*)
               You can download directly from CPAN or
               from BioPerl web site at "http://bioperl.org/".
```

3 Program functions

3.1 Processing command-line options

3e ⟨*Use Modules - Getopt* 3e⟩≡
```
use Getopt::Long;
Getopt::Long::Configure qw/ bundling /;
```

3f ⟨*perl requires - Getopt* 3f⟩≡
```
"Getopt::Long" - processing command-line options.
```

[2]http://bioperl.org/Core/POD/Bio/Seq.html
[3]http://bioperl.org/
[4]http://bioperl.org/Core/POD/Bio/SeqIO.html
[5]http://search.cpan.org/search?dist=bioperl

See ’`man Getopt::Long`’ for further info about this package.

4a ⟨*Parsing command line options* 4a⟩≡
```
sub parse_cmdline() {

    $SIG{__WARN__} = sub { &warn('UNKNOWN_CL_OPTION',$T,$_[0]) };
    GetOptions(
               ⟨command-line options - masking 5c⟩
               ⟨command-line options - fasta 6b⟩
               ⟨command-line options - GFF 6d⟩
               ⟨command-line options with exit 4c⟩
               ) || (&warn('CMD_LINE_ERROR',$T), exit(1));
    $SIG{__WARN__} = 'DEFAULT';

    &report("PROG-START");
} # parse_cmdline
```

4b ⟨*warnings: command-line* 4b⟩≡
```
'UNKNOWN_CL_OPTION' =>
  $Warn."Error trapped while processing command-line:\n".(" "x16)."\%s\n",
'CMD_LINE_ERROR' =>
  $spl.$spw." Please, check your command-line options!!!\n".$Error."\n".
  $spw." ".("."x12)." Type \"maskedfastacoords.pl -h\" for help.\n".$spl,
```

3.1.1 Defining help and auxiliary code chunks

The following command-line checkings look for those options exiting the program: ’`help`’ and ’`version`’. Both need to output to screen without any other message/warning being displayed at the same time.

4c ⟨*command-line options with exit* 4c⟩≡
```
"version"   => \&prt_version,
"h|help|?"  => \&prt_help,
```

4d ⟨*command-line help - help* 4d⟩≡
```
-h, -help            Shows this help.
-version             Shows current version and exits.
```

4e ⟨*Parsing command line options* 4a⟩+≡
```
sub prt_version() {
    &report('SHOW_VERSION',$PROGRAM,$VERSION);
    exit(1);
} # prt_version
```

4f ⟨*messages - parsing command-line* 4f⟩≡
```
'SHOW_VERSION' => $sp."### \%s - \%s\n".$sp,
```

Printing command-line help to `STDERR`:

4g ⟨*Parsing command line options* 4a⟩+≡
```
sub prt_help() {
    print STDERR «"+++EndOfHelp+++";
PROGRAM:
                          $PROGRAM - $VERSION

      ⟨Prog DESC 2d⟩

USAGE:    ⟨Prog USAGE 2c⟩


DESCRIPTION:
    Retrieving positions for masked regions of masked sequences
    in fasta format. Program can also output those coords in GFF
    format, where you can define several fields from command-line.


  REQUIRES:
```

⟨*perl requires help* 5a⟩

```
  COMMAND-LINE OPTIONS:
```

⟨*command-line help* 5b⟩

```
  BUGS:    Report any problem to 'jabril\@imim.es'.

  AUTHOR:  $PROGRAM is under GNU-GPL (C) 2000 - Josep F. Abril

  +++EndOfHelp+++
    exit(1);
  } # prt_help
```

5a ⟨*perl requires help* 5a⟩≡

```
  $PROGRAM needs the following Perl modules
  installed in your system, we used those available
  from the standard Perl distribution. Those that
  are not in the standard distribution are marked
  with an '(*)', in such cases make sure that you
  already have downloaded them from CPAN
  (http://www.perl.com/CPAN) and installed.
```

⟨*perl requires - BioPerl* 3d⟩
⟨*perl requires - Getopt* 3f⟩
⟨*perl requires - Benchmark* 9f⟩

5b ⟨*command-line help* 5b⟩≡

```
  A double dash on itself "-" signals end of the options
  and start of file names (if present). After double dash,
  you can use a single dash "-" as STDIN placeholder.
  Available options and a short description are listed here:

  + General options:
```

⟨*command-line help - help* 4d⟩

⟨*command-line help - masking* 6a⟩

⟨*command-line help - fasta* 6c⟩

```
  + GFF output:
```

⟨*command-line help - GFF* 6e⟩

3.1.2 Defining command-line options

5c ⟨*command-line options - masking* 5c⟩≡

```
  "s|soft-masked"  => ,
  "m|merge-masked" => ,
```

6a ⟨*command-line help - masking* 6a⟩≡
```
  -s, -soft-masked      Default is looking for "N" masked sequence
                         segments. This option distinguish between
                         upper/lower-case and assumes lower-case to
                         define masked regions too (so called soft-masking).
  -m, -merge-masked     Default is differentiating classical-masking
                         (with "N"s) from soft-masking. This option merges
                         both as if they were the same masking type
                         (it also enables soft-masking search so previous
                         option is not required when passing this one).
```

6b ⟨*command-line options - fasta* 6b⟩≡
```
  "l|large-fasta"  => ,
```

6c ⟨*command-line help - fasta* 6c⟩≡
```
  -l, -large-fasta      For large genomic sequence you can enable
                         Bio::SeqIO to work with temporary files
                         avoiding memory overload (though it makes
                         the script to run slowly).
```

6d ⟨*command-line options - GFF* 6d⟩≡
```
  "g|gff"         => ,
  "seq-name=s"    => ,
  "source=s"      => ,
  "f|feature=s"   => ,
  "strand=s"      => ,
  "group=s"       => ,
```

6e ⟨*command-line help - GFF* 6e⟩≡
```
  -g, -gff              Output in GFF format (default coords).

  -seq-name <string>    Sets sequence GFF field (default "noname").
  -source <string>      Sets source GFF field (default "masked").
  -f, -feature <string> Sets feature GFF field (default "masked").
  -strand <string>      Sets strand GFF field (default ".").
  -group  <string>      Sets group GFF field (default none).
```

3.2 Main program function

6f ⟨*Main program function* 6f⟩≡
```
  sub main() {
      my $seqin = Bio::SeqIO->new(-format => 'FASTA', -fh => \*STDIN);
      while (my $sequence = $seqin->next_seq()) {
          my ($sid,$len,$seq,@nuc,@coords,$masked_flg,$match,$msk_num);
          @coords = ();
          ⟨Setting sequence variables from fasta record 6g⟩
          ⟨Finding masked regions coords 7a⟩
          ⟨Writing masked regions coords in GFF 7b⟩
      }; # while
  } # main
```

6g ⟨*Setting sequence variables from fasta record* 6g⟩≡
```
  print STDERR "### READING FASTA............\n";
  $sid  = $sequence->display_id();
  $len  = $sequence->length();
  $seq  = $sequence->seq();
```

TO DO

- We can define mask checking as three different subroutines that will be called by reference, so that a variable name (called for instance `$test`) will be set depending on command-line options.

7a ⟨*Finding masked regions coords* 7a⟩≡
```
print STDERR "###         PARSING SEQUENCE: $sid ($len bp)\n";
@nuc = split //og, $seq;
($masked_flg,$match) = ($F,$F) ;
for (my $n = 0; $n <= $#nuc; $n++) {
    $match = ( $nuc[$n] =~ /[NnXx]/o ) ? $T : $F;
    ( !$masked_flg && $match ) && do {
        $masked_flg = $T ;
        # $n contains the last non-masked nucleotide
        push @coords, ($n + 1);
        next;
    };
    $masked_flg && do {
        $match && (next);
        $masked_flg = $F ;
        # $n contains the last masked nucleotide now
        push @coords, $n;
    };
}; # for nuc in $seq
# if last nucleotide is masked, previous loop not includes its coord.
$masked_flg &&( push @coords, $len);
```

7b ⟨*Writing masked regions coords in GFF* 7b⟩≡
```
$msk_num = scalar(@coords) / 2;
print STDERR "###         WRITING GFF COORDS: $msk_num masked regions found.\n";
for (my $n = 0; $n <= $#coords; $n+=2) {
    my $GFFstring = ("\%s\t" x 5).(".\t" x 3).".\n";
    printf STDOUT $GFFstring, $sid, "masked", "masked", @coords[$n..($n + 1)];
}; # for coords in @coords
```

A empty appendix section

A.1 empty appendix subsection

B Common code blocks

B.1 PERL scripts

9a ⟨*PERL shebang* 9a⟩≡
```
#!/usr/bin/perl -w
# This is perl, version 5.005_03 built for i386-linux
#
⟨Program Description 2e⟩
#
⟨GNU License 12⟩
#
⟨Version Control Id Tag 11f⟩
#
use strict;
#
# BEGIN {
    ⟨Program Info 2b⟩
    my $DATE = localtime;
    my $USER = defined($ENV{USER}) ? $ENV{USER} : 'Child Process';
    my $host = `hostname`;
    chomp($host);
# } # BEGIN
```

9b ⟨*Boolean* 9b⟩≡
```
my ($T,$F) = (1,0); # for 'T'rue and 'F'alse
```

B.1.1 Timing our scripts

The '`Benchmark`' module encapsulates a number of routines to help to figure out how long it takes to execute a piece of code and the whole script.

9c ⟨*Use Modules - Benchmark* 9c⟩≡
```
use Benchmark;
  ⟨Timer ON 9d⟩
```

See '`man Benchmark`' for further info about this package. We set an array to keep record of timing for each section.

9d ⟨*Timer ON* 9d⟩≡
```
my @Timer = (new Benchmark);
```

9e ⟨*Common PERL subs - Benchmark* 9e⟩≡
```
sub timing() {
    push @Timer, (new Benchmark);
    # partial time
    $_[0] ||
        (return timestr(timediff($Timer[$#Timer],$Timer[($#Timer - 1)])));
    # total time
    return timestr(timediff($Timer[$#Timer],$Timer[0]));
} # timing
```

9f ⟨*perl requires - Benchmark* 9f⟩≡
```
"Benchmark" - checking and comparing running times of code.
```

B.1.2 Printing complex Data Structures

With '`Data::Dumper`' we are able to pretty print complex data structures for debugging them.

9g ⟨*Use Modules - Dumper* 9g⟩≡
```
use Data::Dumper;
local $Data::Dumper::Purity = 0;
local $Data::Dumper::Deepcopy = 1;
```

B.1.3 Common functions

10a ⟨*Skip comments and empty records* 10a⟩≡
```
next if /^\#/o;
next if /^\s*$/o;
chomp;
```

10b ⟨*Common PERL subs - Min Max* 10b⟩≡
```
#
sub max() {
    my $z = shift @_;
    foreach my $l (@_) { $z = $l if $l > $z };
    return $z;
} # max
sub min() {
    my $z = shift @_;
    foreach my $l (@_) { $z = $l if $l < $z };
    return $z;
} # min
```

10c ⟨*Common PERL subs - Text fill* 10c⟩≡
```
#
sub fill_right() { $_[0].($_[2] x ($_[1] - length($_[0]))) }
sub fill_left()  { ($_[2] x ($_[1] - length($_[0]))).$_[0] }
sub fill_mid()   {
    my $l = length($_[0]);
    my $k = int(($_[1] - $l)/2);
    ($_[2] x $k).$_[0].($_[2] x ($_[1] - ($l+$k)));
} # fill_mid
```

These functions are used to report to STDERR a single char for each record processed (useful for reporting parsed records).

10d ⟨*Common PERL subs - Counter* 10d⟩≡
```
#
sub counter { # $_[0]~current_pos++ $_[1]~char
    print STDERR "$_[1]";
    (($_[0] % 50) == 0) && (print STDERR "[".&fill_left($_[0],6,"0")."]\n");
} # counter
#
sub counter_end { # $_[0]~current_pos   $_[1]~char
    (($_[0] % 50) != 0) && (print STDERR "[".&fill_left($_[0],6,"0")."]\n");
} # counter_end
```

10e ⟨*Global Vars - Counter* 10e⟩≡
```
my ($n,$c); # counter and char (for &counter function)
```

B.1.4 Common functions for reporting program processes

Function ’`report`’ requires that a hash variable ’`%Messages`’ has been set, such hash contains the strings for each report message we will need. The first parameter for ’`report`’ is a key for that hash, in order to retrieve the message string, the other parameters passed are processed by the `sprintf` function on that string.

10f ⟨*Common PERL subs - STDERR* 10f⟩≡
```
sub report() { print STDERR sprintf($Messages{ shift @_ },@_) }
```

The same happens to ’`warn`’ function which also uses the hash variable ’`%Messages`’ containing the error messages.

11a ⟨*Common PERL subs - STDERR* 10f⟩+≡
```
sub warn() { print STDERR sprintf($Messages{ shift @_ }, @_) }
```

Those are accessory variables for the messages strings:

11b ⟨*Stderr subs vars* 11b⟩≡
```
my $line = ('#' x 80)."\n";
my $s = '### ';
my $Error = "\<\<\<  ERROR  \>\>\> ";
my $Warn  = "\<\<\< WARNING \>\>\> ";
my $spl   = "\<\<\<\-\-\-\-\-\-\-\-\-\>\>\>\n";
my $spw   = "\<\<\<         \>\>\> ";
```

And here the main messages hash:

11c ⟨*Stderr subs vars* 11b⟩+≡
```
my %Messages = (
    # ERROR MESSAGES
    ⟨warnings: command-line 4b⟩
    # WORKING MESSAGES
    ⟨messages: program running 3a⟩
   ); # %Messages
```

B.2 BASH scripts

11d ⟨*BASH shebang* 11d⟩≡
```
#!/usr/bin/bash
# GNU bash, version 2.03.6(1)-release (i386-redhat-linux-gnu)
⟨Version Control Id Tag 11f⟩
#
SECONDS=0 # Reset Timing
# Which script are we running...
L="####################"
{ echo "$L$L$L$L";
  echo "### RUNNING [$0]";
  echo "### Current date:‘date‘";
  echo "###"; } 1>&2;
```

11e ⟨*BASH script end* 11e⟩≡
```
{ echo "###"; echo "### Execution time for [$0] : $SECONDS secs";
  echo "$L$L$L$L";
  echo ""; } 1>&2;
#
exit 0
```

B.3 Version control tags

This document is under Revision Control System (RCS). The version you are currently reading is the following:

11f ⟨*Version Control Id Tag* 11f⟩≡
```
# $Id: maskedfastacoords.nw,v 1.2 2001/09/19 19:45:27 jabril Exp $
```

B.4 GNU General Public License

12 ⟨*GNU License* 12⟩≡

```
# #------------------------------------------#
# #                          maskedfastacoords                         #
# #------------------------------------------#
#
#    Remember to put a short description of your script here...
#
#     Copyright (C) 2001 - Josep Francesc ABRIL FERRANDO
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
#
# #------------------------------------------#
```

C Extracting code blocks from this document

From this file we can obtain both the code and the documentation. The following instructions are needed:

C.1 Extracts Script code chunks from the NOWEB file

Remember when tangling that '-L' option allows you to include program line-numbering relative to original NOWEB file. Then the first line of the executable files is a comment, not a shebang, and must be removed to make scripts runnable.

13a ⟨*tangling* 13a⟩≡

```
# showing line numbering comments in program
notangle -L -R"maskedfastacoords" $WORK/$nwfile.nw | \
    perl -ne '$.>1 && print' | cpif $BIN/maskedfastacoords.pl ;
chmod a+x $BIN/maskedfastacoords.pl ;
```

13b ⟨*tangling* 13a⟩+≡

```
# reformating program with perltidy
notangle -R"maskedfastacoords" $WORK/$nwfile.nw | \
    perltidy - | cpif $BIN/maskedfastacoords.pl ;
# html pretty-printing program with perltidy
notangle -R"maskedfastacoords" $WORK/$nwfile.nw | \
    perltidy -html - | cpif $DOCS/html/maskedfastacoords.html ;
#
```

C.2 Extracting different Config Files

13c ⟨*tangling* 13a⟩+≡

```
notangle -R"root" $WORK/$nwfile.nw | \
         cpif $DATA/root_config ;
```

C.3 Extracting documentation and LaTeX'ing it

13d ⟨*tangling* 13a⟩+≡

```
notangle -Rweaving  $WORK/$nwfile.nw | cpif $WORK/nw2tex ;
notangle -RLaTeXing $WORK/$nwfile.nw | cpif $WORK/ltx ;
chmod a+x $WORK/nw2tex $WORK/ltx;
```

13e ⟨*tangling complementary LaTeX files* 13e⟩≡

```
notangle -R"HIDE: LaTeX new definitions" $WORK/$nwfile.nw | cpif $DOCS/defs.tex ;
notangle -R"HIDE: TODO" $WORK/$nwfile.nw | cpif $DOCS/todo.tex ;
```

13f ⟨*weaving* 13f⟩≡

```
⟨BASH shebang 11d⟩
# weaving and LaTeXing
⟨BASH Environment Variables 14b⟩
⟨tangling complementary LaTeX files 13e⟩
noweave -v -t4 -delay -x -filter 'elide "HIDE: *"' \
        $WORK/$nwfile.nw | cpif $DOCS/$nwfile.tex ;
# noweave -t4 -delay -index $WORK/$nwfile.nw > $DOCS/$nwfile.tex
pushd $DOCS/ ;
#
latex $nwfile.tex ;
dvips $nwfile.dvi -o $nwfile.ps -t a4 ;
#
popd;
⟨BASH script end 11e⟩
```

14a ⟨*LaTeXing* 14a⟩≡

```
⟨BASH shebang 11d⟩
# only LaTeXing
⟨BASH Environment Variables 14b⟩
pushd $DOCS/ ;
#
echo "### RUNNING LaTeX on $nwfile.tex" 1>&2 ;
latex $nwfile.tex ;
latex $nwfile.tex ;
latex $nwfile.tex ;
dvips $nwfile.dvi -o $nwfile.ps -t a4 ;
#
# pdflatex $nwfile.tex ;
echo "### CONVERTING PS to PDF: $nwfile" 1>&2 ;
ps2pdf $nwfile.ps $nwfile.pdf ;
#
popd ;
⟨BASH script end 11e⟩
```

C.4 Defining working shell variables for the current project

14b ⟨*BASH Environment Variables* 14b⟩≡

```
#
# Setting Global Variables
WORK="/home/ug/jabril/development/softjabril/maskedfastacoords" ;
BIN="$WORK/bin" ;
PARAM="$BIN/param" ;
DOCS="$WORK/docs" ;
DATA="$WORK/data" ;
nwfile="maskedfastacoords" ;
export WORK BIN PARAM DOCS DATA nwfile ;
#
```

14c ⟨*tangling* 13a⟩+≡

```
#
# BASH Environment Variables
notangle -R'BASH Environment Variables' $WORK/$nwfile.nw | \
         cpif $WORK/.bash_VARS ;
source $WORK/.bash_VARS ;
#
```